PAR Laboratory Assignment

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

Teacher:

Jordi Tubella

Students:

Eric Hurtado (par2114)

Jordi Pitarch (par2118)

Index:

1. Introduction

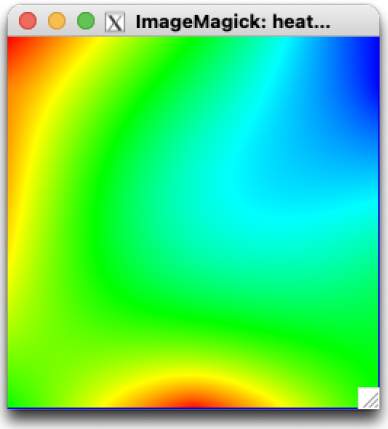
In this last laboratory assignment we will work on the parallelisation of a sequential code that simulates the diffusion of heat in a solid body using two different solvers for the heat equation, Jacobi and Gauss-Seidel.

Firstly, we will proceed to study the sequential code *heat.c* and we will do an analysis with Tareador, in the second section we will do the parallelisation of the two heat equation solvers using the OpenMP directives.

2. Sequential heat diffusion program and analysis with tareador

Firstly, we are going to do an analysis of two solvers, the Jacobi and gauss-seidel. In order to compile the program with both, we have executed the following commands with the following responses:

```
make heat
./heat test.dat -a 0 -o heat-jacobi.ppm
display 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: 4.363
Flops and Flops per second: (11.182 GFlop => 2562.55 MFlop/s)
Convergence to residual=0.000050: 15756 iterations

```
./heat test.dat -a 1 -o heat-gaus.ppm
display heat-gaus.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: 8.815
Flops and Flops per second: (8.806 GFlop => 999.04 MFlop/s)
Convergence to residual=0.000050: 12409 iterations

Next, we are going to see the graphs generated by the tareador using the two solvers mentioned above. We have used the program heat-tareador, which was given by the teachers.

If we see the script, we can see that the flag 0 is for Jacobi solver, while the flag 1 is for the gauss-seidel solver.

Firstly we have executed the Jacobi solver with the following command: (left)

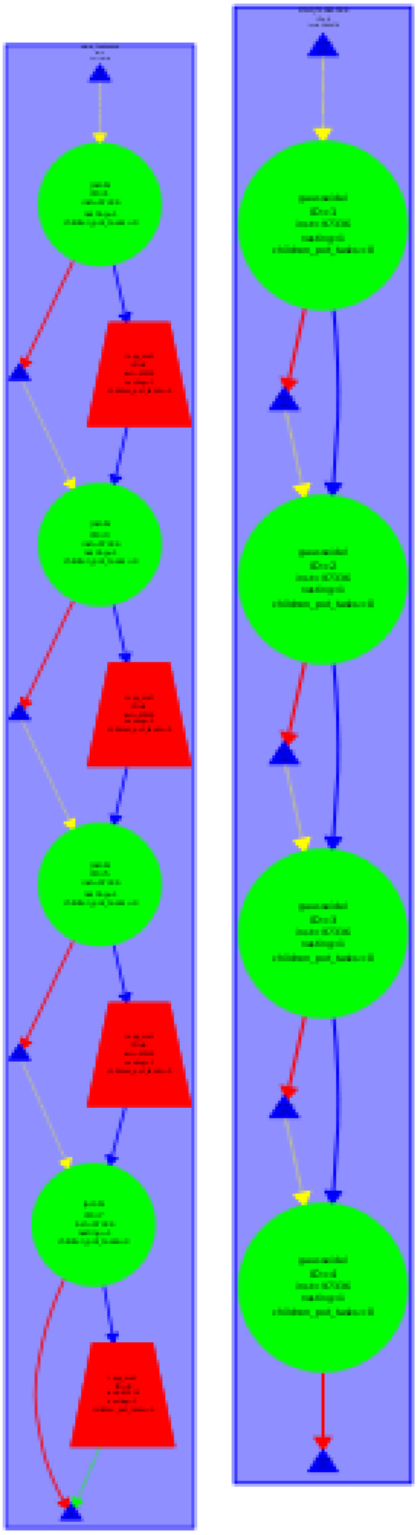
```
./run-tareador.sh heat-tareador 0
```

Resolution : 32
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: 0.008
Flops and Flops per second: (0.000 GFlop => 5.37 MFlop/s)
Convergence to residual=2.188666: 4 iterations

Next, we have executed for Gauss-seidel solver with: (right)

```
./run-tareador.sh heat-tareador 1
```

Resolution : 32
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: 0.005
Flops and Flops per second: (0.000 GFlop => 8.64 MFlop/s)
Convergence to residual=3.432559: 4 iterations

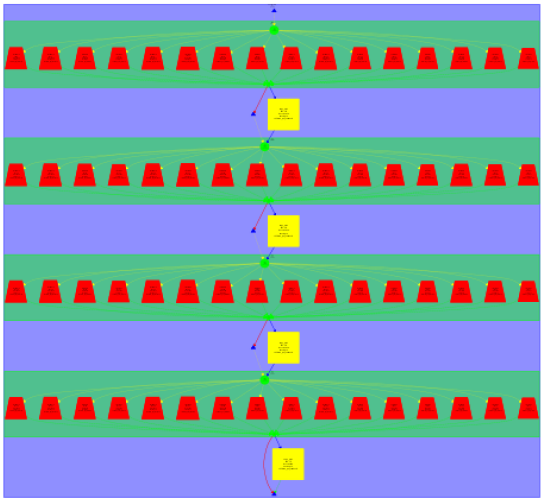


If we take a look at the function “solve” which is used for the calculation of both solver, we can see that this is the code that should be parallelized in order to get a different granularity. If we want to improve the parallelization, we should change the granularity to a finer one. Now, there is no parallelism with this granularity.

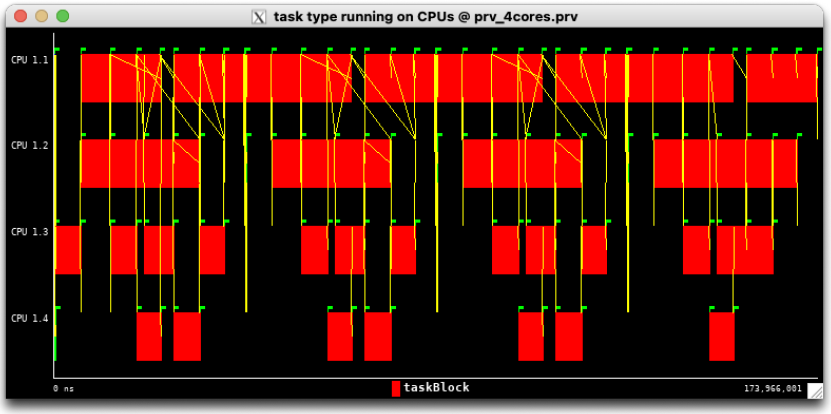
Now, we have modified the code to make one task per block. This change has been done using tasks inside the second “for”, because this is the level where the tasks generated are of the size required. The code has been modified like the following image:

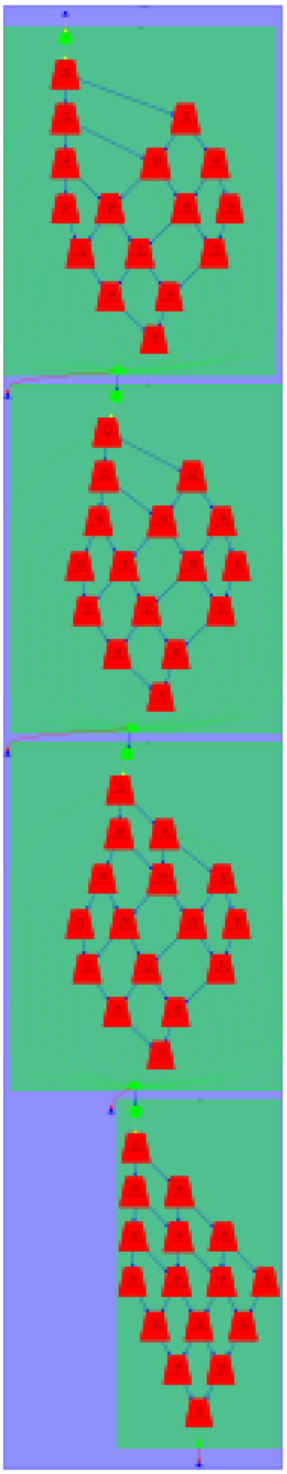
```
// 2D-blocked solver: one iteration step
double solve (double *u, double *unew, unsigned sizex, unsigned sizey) {
    double tmp, diff, sum=0.0;

    int nblocksi=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) {
        tareador_start_task("taskBlock");
        int j_start = lowerb(blockj, nblocksj, sizey);
        int j_end = upperb(blockj, nblocksj, sizey);
        for (int i=max(1, i_start); i<=min(sizex-2, i_end); i++) {
          for (int j=max(1, j_start); j<=min(sizey-2, j_end); j++) {
                tmp = 0.25 * ( u[ i*sizey          + (j-1) ] +  // left
                               u[ i*sizey      + (j+1) ] +  // right
                               u[ (i-1)*sizey + j     ] +  // top
                               u[ (i+1)*sizey + j     ] ); // bottom
                diff = tmp - u[i*sizey+ j];
                sum += diff * diff;
                unew[i*sizey+j] = tmp;
          }
        }
        tareador_end_task("taskBlock");
      }
    }
    //tareador_enable_object(&sum);

    return sum;
}
```

If we take a look at the tareador graphics of Jacobi solver, we get the following images:



In the other hand, if we execute with the gauss-seidel solver we get the next images:





As we can see in Jacobi solver, the tasks identified by the yellow color are from the function copy_mat which is causing serialization in our code. We are obtaining more parallelism when we are not protecting the variable sum as a result of the other threads do not have to wait until the variable finishes his protection. This protection can be done with OpenMP using the clauses:

“#pragma omp atomic”

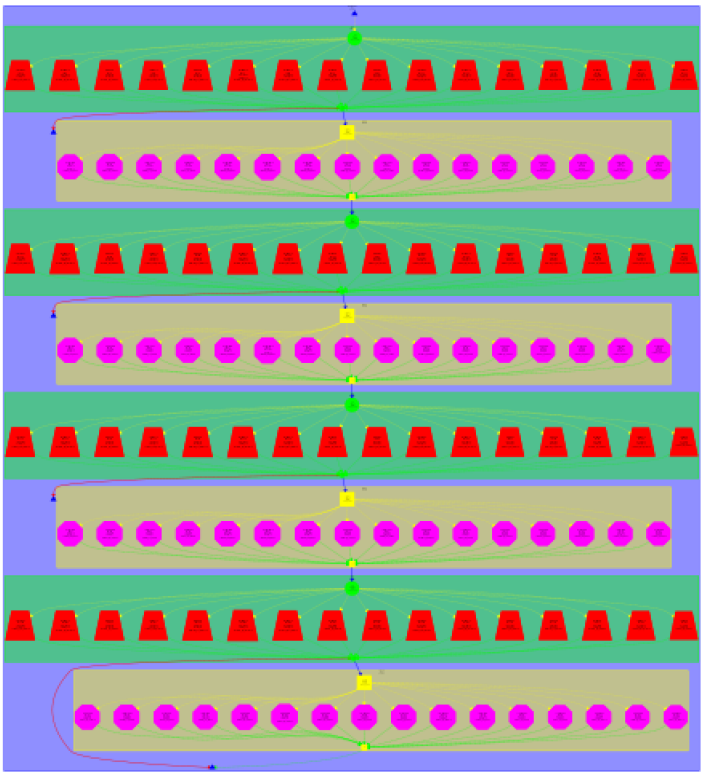
“#pragma omp critical”

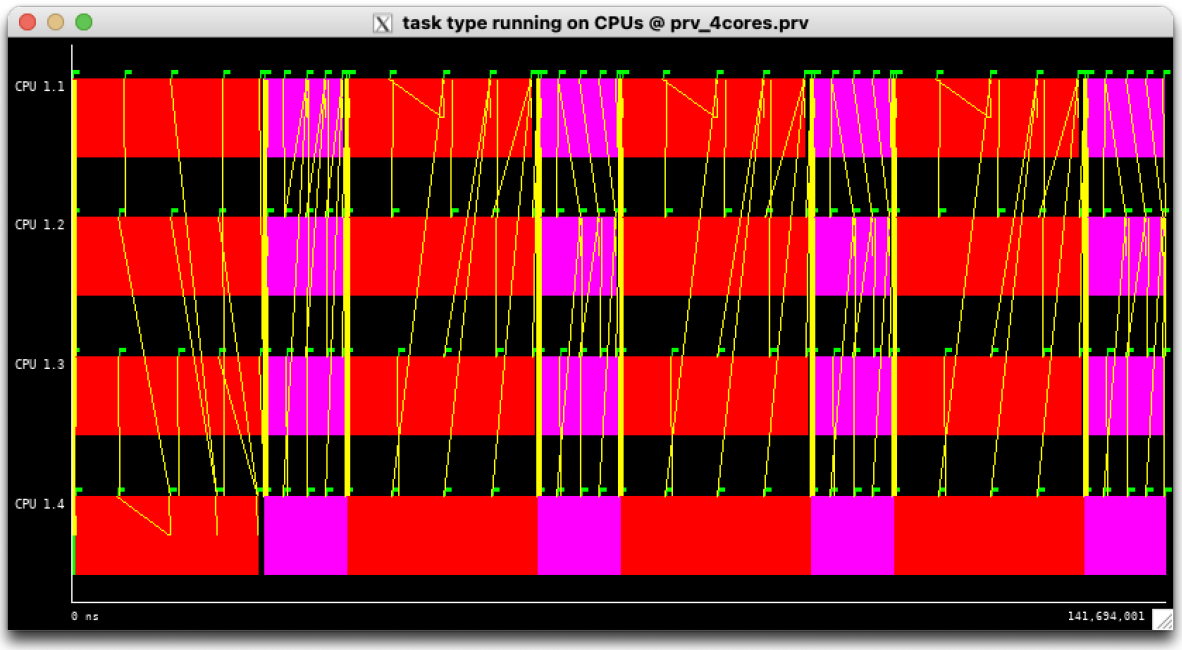
“#pragma omp reduction”

To solve it, we have modified the code of “solver-tareador.c” in order to parallelize the “copy_mat function”.

```
// Function to copy one matrix into another
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) {
        tareador_start_task("copyMatTask");
        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];
        tareador_end_task("copyMatTask");
      }
    }
}
```

If we execute this version in the tareador, we get the following graphs using Jacobi solver due to the fact that is which use more the copy_mat function:





As we can see in the graphs, we have now fully parallelized our program.

3. Parallelisation of the head equation solvers

3.1 Jacobi solver

In this section, we are going to parallelise the sequential code for the heat equation code considering the use of the Jacobi solver, using the implicit tasks generated in #pragma omp parallel, following a geometric block data decomposition by rows.

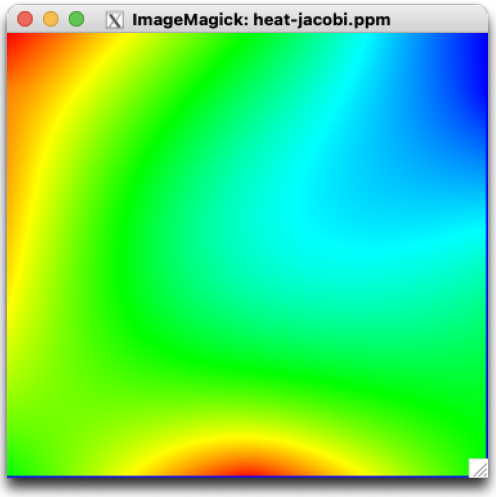
We started parallelising the solve function from the solver-omp.c in order to reduce the dependences that we have discovered previously. The code has been modified to the image shown below.

```
// 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)
    {
      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;
}
```

After the changes, in order to see if the execution is correct, we have executed the script submit-omp.sh with 1 and 8 threads. This has been possible as a result of the execution of the following commands:

make heat-omp

sbatch submit-omp.sh heat-omp 0 1



sbatch submit-omp.sh heat-omp 0 8

We can see that the images are the same, and as a result of this, we can assume that the code execution in terms of parallelisation is correct.

We have changed the code with the private(diff) because we need to have a correct value of diff value and the reduction to have the value of the sum of “sum” correct in the iterations in order to not have wrong values.

Next, we have executed the submit-strong-extrae.sh script in order to trace the execution for a different number of threads.

Overview of whole program execution metrics				
Number of processors	1	4	8	16
Elapsed time (sec)	2.82	2.16	2.07	2.01
Speedup	1.00	1.30	1.36	1.40
Efficiency	1.00	0.33	0.17	0.09

Table 1: Analysis done on Wed Nov 30 11:10:29 AM CET 2022, par2118

Overview of the Efficiency metrics in parallel fraction, ϕ=65.14%				
Number of processors	1	4	8	16
Global efficiency	99.64%	71.19%	61.15%	33.01%
Parallelization strategy efficiency	99.64%	83.48%	98.27%	97.86%
Load balancing	100.00%	85.70%	99.90%	99.80%
In execution efficiency	99.64%	97.41%	98.37%	98.06%
Scalability for computation tasks	100.00%	85.28%	62.22%	33.73%
IPC scalability	100.00%	85.53%	71.09%	45.17%
Instruction scalability	100.00%	99.97%	94.71%	83.58%
Frequency scalability	100.00%	99.73%	92.41%	89.35%

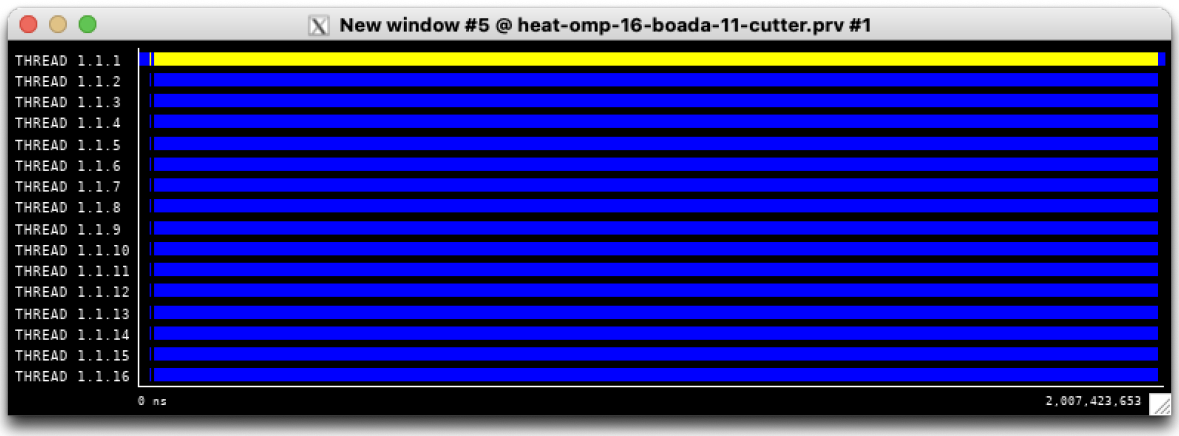
Table 2: Analysis done on Wed Nov 30 11:10:29 AM CET 2022, par2118

Statistics about explicit tasks in parallel fraction				
Number of processors	1	4	8	16
Number of implicit tasks per thread (average us)	1000.0	1000.0	1000.0	1000.0
Useful duration for implicit tasks (average us)	1830.52	536.64	367.73	339.18
Load balancing for implicit tasks	1.0	0.86	1.0	1.0
Time in synchronization implicit tasks (average us)	0	0	0	0
Time in fork/join implicit tasks (average us)	6.56	181.93	6.39	7.72

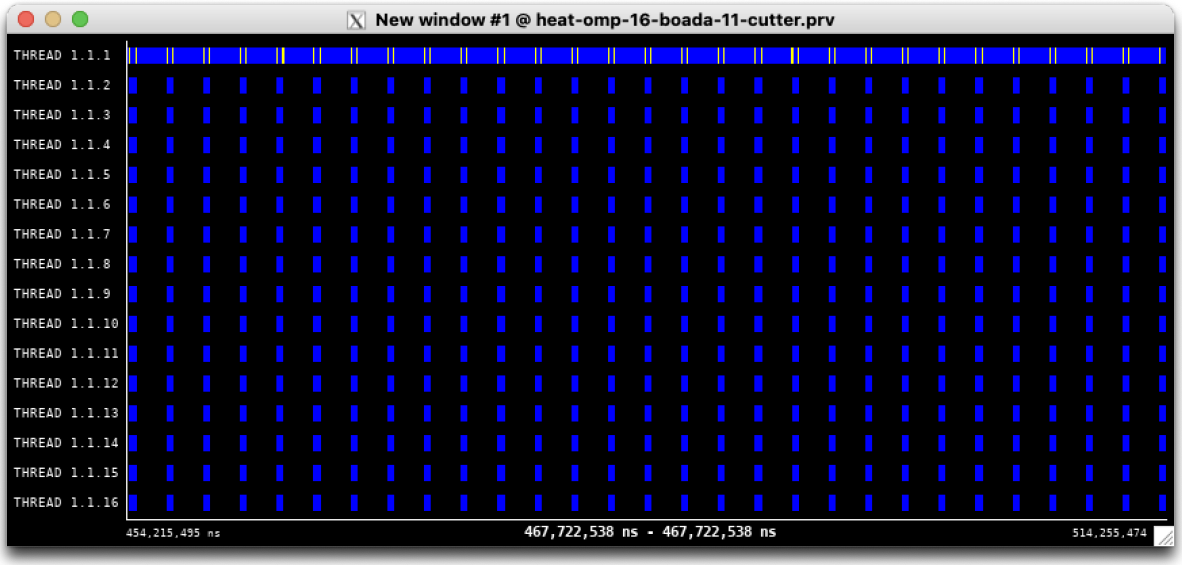
Table 3: Analysis done on Wed Nov 30 11:10:29 AM CET 2022, par2118

As we can see in the tables, the execution time is reduced if we increase the number of threads, this show us that the parallelisation is correct. By the other hand we can see that the speedup of the program increases so few in 4 to 8 threads. In the second table we can see that the parallel fraction is so low, this is due to the fact that there are parts of the code that are not parallelised. In the third table we can see that no time is spend synchonization implicit tasks.

Following, we have executed the “wxparaver” to see the traces of the execution with 16 threads.



If we take a look closely, we can see that the running time that is seen above is:



The region that is making this low parallel fraction is caused by the copy_mat function, which is not parallelized.

In order to maximize the parallel fraction of our code, we have modified the code as the following image:

```
// 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
    {
    //for (int blocki=0; blocki<nblocksi; ++blocki) {
      int blocki = omp_get_max_threads();
      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];
      }
    }
    //}
}
```

When we execute this new version, with the submit-omp.sh script, specifying the 16 threads, and the Jacobi solver, we get the following result:

```
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: 2.353
Flops and Flops per second: (11.182 GFlop => 4751.12 MFlop/s)
Convergence to residual=0.000050: 15756 iterations
```

If we execute the submit-strong-extrae.sh script, we can see the following tables:

Overview of whole program execution metrics				
Number of processors	1	4	8	16
Elapsed time (sec)	1.79	0.51	0.31	0.19
Speedup	1.00	3.51	5.86	9.60
Efficiency	1.00	0.88	0.73	0.60

Table 1: Analysis done on Tue Dec 13 10:42:13 AM CET 2022, par2118

Overview of the Efficiency metrics in parallel fraction, ϕ=98.34%				
Number of processors	1	4	8	16
Global efficiency	99.70%	91.88%	80.81%	72.01%
Parallelization strategy efficiency	99.70%	96.33%	96.31%	93.06%
Load balancing	100.00%	97.93%	99.68%	99.07%
In execution efficiency	99.70%	98.37%	96.62%	93.94%
Scalability for computation tasks	100.00%	95.37%	83.90%	77.38%
IPC scalability	100.00%	95.74%	91.46%	87.71%
Instruction scalability	100.00%	99.95%	98.91%	97.78%
Frequency scalability	100.00%	99.67%	92.75%	90.23%

Table 2: Analysis done on Tue Dec 13 10:42:13 AM CET 2022, par2118

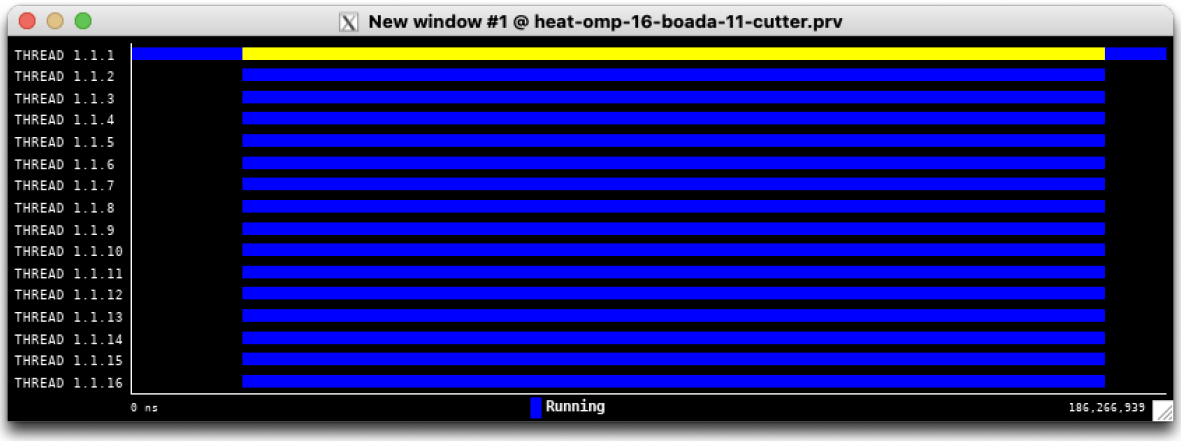
Statistics about explicit tasks in parallel fraction				
Number of processors	1	4	8	16
Number of implicit tasks per thread (average us)	2000.0	2000.0	2000.0	2000.0
Useful duration for implicit tasks (average us)	877.02	229.89	130.66	70.83
Load balancing for implicit tasks	1.0	0.98	1.0	0.99
Time in synchronization implicit tasks (average us)	0	0	0	0
Time in fork/join implicit tasks (average us)	2.63	11.87	5.3	5.73

Table 3: Analysis done on Tue Dec 13 10:42:13 AM CET 2022, par2118

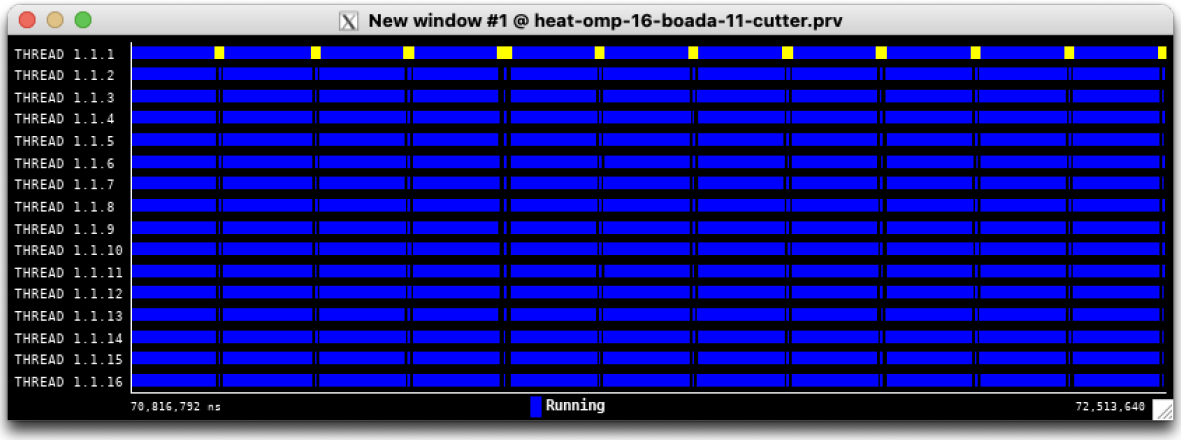
First of all, we have an increment of the parallel fraction as a result of the parallelization of the function copy_mat, also we have an execution time more reduced than before. The problem is the number of threads, as a result of the initialization of the variable “nblocksi =

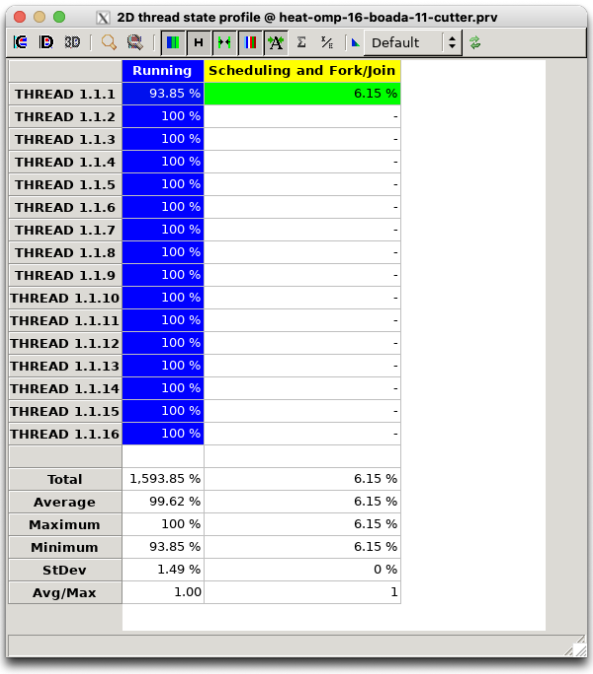
omp_get_max_threads()", the scalability will be better if we increment the number of threads, this can be seen in the first table, because the execution time with more threads is better.

If we execute the tareador, we can see the following timeline:



If we take a look closely, we can see that it is not spend a lot of time during the tasks, and only one thread is who makes new tasks.





	Running	Scheduling and Fork/Join
THREAD 1.1.1	93.85 %	6.15 %
THREAD 1.1.2	100 %	-
THREAD 1.1.3	100 %	-
THREAD 1.1.4	100 %	-
THREAD 1.1.5	100 %	-
THREAD 1.1.6	100 %	-
THREAD 1.1.7	100 %	-
THREAD 1.1.8	100 %	-
THREAD 1.1.9	100 %	-
THREAD 1.1.10	100 %	-
THREAD 1.1.11	100 %	-
THREAD 1.1.12	100 %	-
THREAD 1.1.13	100 %	-
THREAD 1.1.14	100 %	-
THREAD 1.1.15	100 %	-
THREAD 1.1.16	100 %	-
Total	1,593.85 %	6.15 %
Average	99.62 %	6.15 %
Maximum	100 %	6.15 %
Minimum	93.85 %	6.15 %
StDev	1.49 %	0 %
Avg/Max	1.00	1

The execution time for the invocation to function solve has changed as a result of the amount of time required to execute the function we have just modified. This new version is not spending time in synchronization as we can see in the table. As a result, this new version is much better and is performing better than the other.

3.2 Gauss–Seidel solver

In the second place we have changed the code to the Gauss-Seidel solver, in order to experiment with other ways to optimize the program. We have implemented this as 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=nblocksi;
    int mat[24] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
    #pragma omp parallel private(tmp, diff) reduction(+:sum)
    {
      int cont;
      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) {
           do {
                #pragma omp atomic read
                cont = mat[blocki-1];
           } while (cont <= blockj);
        }
        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) {
           #pragma omp atomic update
           mat[blocki] = mat[blocki] + 1;
        }
      }
    }

    return sum;
}
```

And if we execute the script submit-omp.sh we get the expected image that we had in previous versions of the program:

By the other hand, with the script submit-strong-extrae.sh, we can see the following tables:

Overview of whole program execution metrics				
Number of processors	1	4	8	16
Elapsed time (sec)	6.92	3.11	1.75	0.97
Speedup	1.00	2.22	3.96	7.17
Efficiency	1.00	0.56	0.49	0.45

Table 1: Analysis done on Wed Dec 21 11:00:05 AM CET 2022, par2118

Overview of the Efficiency metrics in parallel fraction, ϕ=99.20%				
Number of processors	1	4	8	16
Global efficiency	99.96%	55.78%	50.02%	46.06%
Parallelization strategy efficiency	99.96%	78.47%	99.73%	99.45%
Load balancing	100.00%	78.57%	99.98%	99.93%
In execution efficiency	99.96%	99.87%	99.76%	99.52%
Scalability for computation tasks	100.00%	71.08%	50.16%	46.32%
IPC scalability	100.00%	78.12%	76.92%	75.23%
Instruction scalability	100.00%	92.26%	70.30%	68.56%
Frequency scalability	100.00%	98.63%	92.76%	89.80%

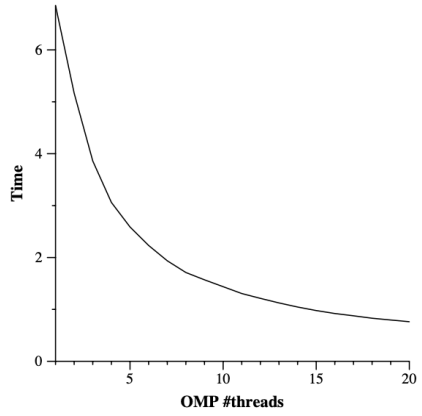
Table 2: Analysis done on Wed Dec 21 11:00:05 AM CET 2022, par2118

Statistics about explicit tasks in parallel fraction				
Number of processors	1	4	8	16
Number of implicit tasks per thread (average us)	1000.0	1000.0	1000.0	1000.0
Useful duration for implicit tasks (average us)	6860.42	2412.79	1709.71	925.75
Load balancing for implicit tasks	1.0	0.79	1.0	1.0
Time in synchronization implicit tasks (average us)	0	0	0	0
Time in fork/join implicit tasks (average us)	2.79	1323.13	4.69	5.35

Table 3: Analysis done on Wed Dec 21 11:00:05 AM CET 2022, par2118

If we take a look at the tables, we can see that the scalability is worse than the previous one, but we get that with more threads, we get a better performance of our code. In the second and third table, with 4 threads, the program parallelization strategy and load balancing is lower than we had expected, and the same for the fork/join time in the third, which is much bigger than it should be.

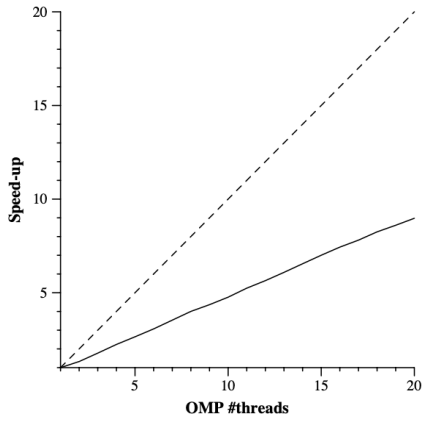
Another thing that we can do to see the parallelization is with the script “submit-omp.sh”, which shows us the strong and weak scalability. The graphics that we had got are the following:



par2118
Min elapsed execution time
Generated by par2118 on Wed Dec 21 11:12:05 AM CET 2022

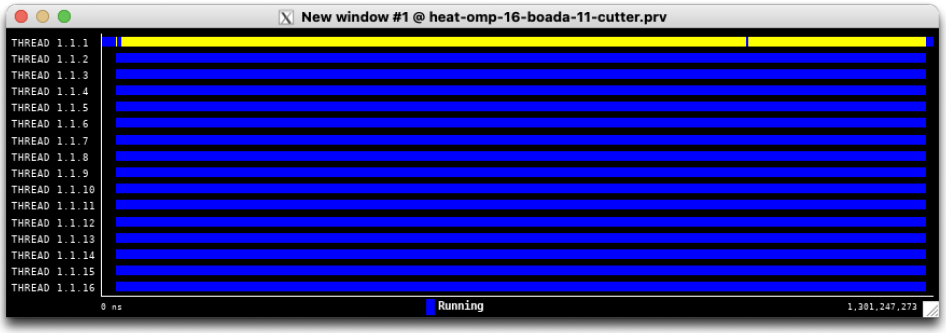
As we can see in these images the scalability of the program and the execution time is better with more threads, until we get 16 threads, this is due to the fact the overheads, that makes the execution time bigger due to synchronizations.

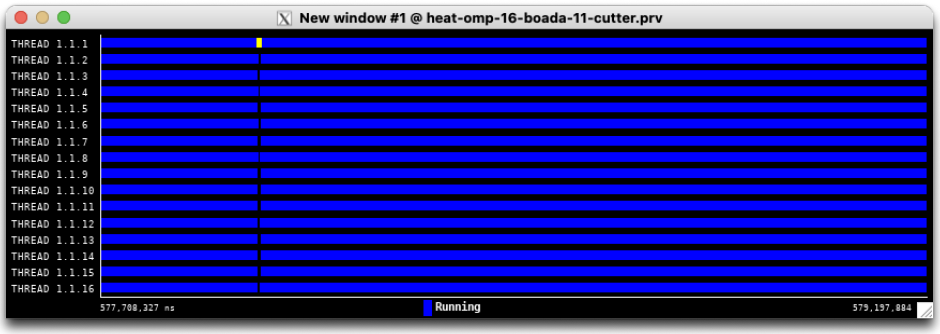
The parallelism could be increased using a better way to determine the size of the blocks considering the number of threads used in the execution.

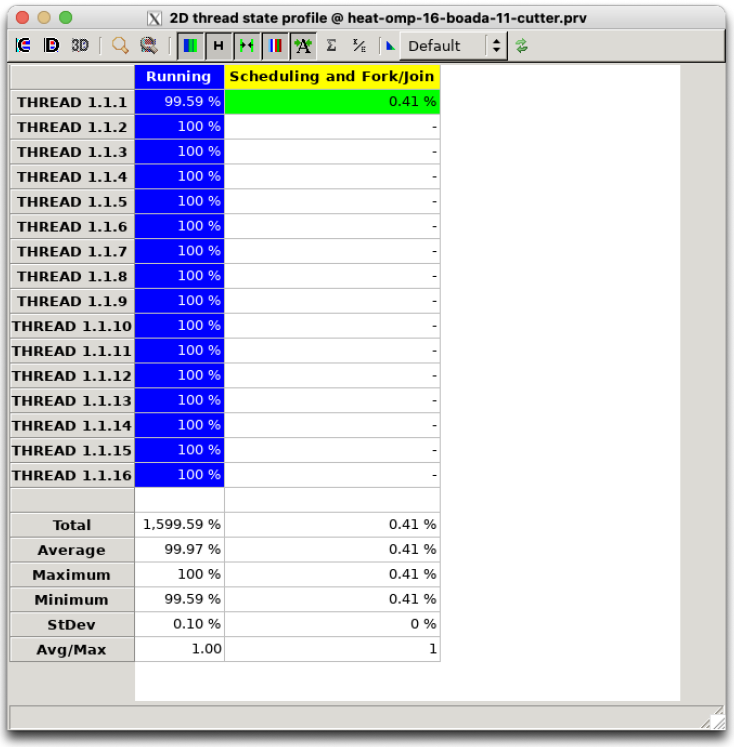


par2118
Speed-up wrt sequential time
Generated by par2118 on Wed Dec 21 11:12:05 AM CET 2022

With paraver, we have got the following plots:







	Running	Scheduling and Fork/Join
THREAD 1.1.1	99.59 %	0.41 %
THREAD 1.1.2	100 %	-
THREAD 1.1.3	100 %	-
THREAD 1.1.4	100 %	-
THREAD 1.1.5	100 %	-
THREAD 1.1.6	100 %	-
THREAD 1.1.7	100 %	-
THREAD 1.1.8	100 %	-
THREAD 1.1.9	100 %	-
THREAD 1.1.10	100 %	-
THREAD 1.1.11	100 %	-
THREAD 1.1.12	100 %	-
THREAD 1.1.13	100 %	-
THREAD 1.1.14	100 %	-
THREAD 1.1.15	100 %	-
THREAD 1.1.16	100 %	-
Total	1,599.59 %	0.41 %
Average	99.97 %	0.41 %
Maximum	100 %	0.41 %
Minimum	99.59 %	0.41 %
StDev	0.10 %	0 %
Avg/Max	1.00	1

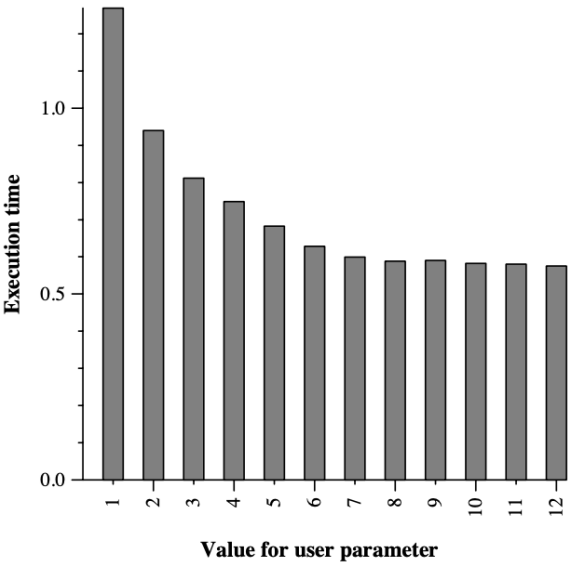
When studying the scalability of the Gauss-Seidel solution, we can see a lower scalability than in the Jacobi version. Examining the execution times and the speed-up of this new version and considering the execution times of the versions of Jacobi with parallelized and non-parallelized copy _mat, we conclude that, despite the Gauss-Seidel solver does not respond as well to parallelization as Jacobi, this is because of the copy_ mat function.

To improve the efficiency of the program, we have changed the values for the dimension j in order to test different values and see the result that we get. We have modified the previous code changing the variable nblocksj = userparam * nblocksi

```
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;
    int mat[24] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
    #pragma omp parallel private(tmp, diff) reduction(+:sum)
    {
```

With these modifications we change the value of the dimension j in order to get a different size of the blocks that we use next.

Nextly, we have executed the submit-userparam-omp.sh script with 12 threads and we got the following result:



par2118
Average elapsed execution time (heat difussion)
Wed Dec 21 11:22:45 AM CET 2022

As we can see in the graph from the second script, if we increase the user param it takes less time. This is produced as a result that by changing the userparam we are changing the size of the blocks, and making them bigger it increases the parallelism of the program and as a result the execution time gets reduced.

4. Conclusions

First of all, we have practiced with the heat dissipation scripts, which have been used during the rest of the laboratory assignment. Also, we have compiled and executed that were given by the teachers in order to see the images generated. Next we started studying the Jacobi and Gauss-Seidel solvers.

In the second session we started with the Jacobi solver, we parallelised and studied the scalability, first with the function “solve” and later with the function “copy_mat”. After this change we saw the big differences in terms of the parallelization fraction.

Finally in the last session we used the Gauss-Seidel solver and we done the same as in the other to see the difference between them. In addition to explore more the parallelization, we have used the user param in order to change the j dimension and see the differences and get a conclusion.

5. Final Survey

From our point of view, we think that modelfactors is a very useful tool to see the parallelization of a program and more information about it. It is easy to understand and it is very practical for some explanations. It would be very useful for the next generations of PAR students this tool for a better understanding of the laboratory assignments.

Modelfactors:
9.5

Tareador:
8

Extrae +Paraver:
5
(Due to the time required to do some actions in computer that are not provided by the university or using a eduroam in our laptops in class)