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

Víctor Asenjo Carvajal - par1302
Ferran De La Varga Antoja - par1305

May 2022
Semester of Spring 2021-2022

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Index

1 Introduction

In this 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).

Each solver has different numerical properties which are not relevant for the purposes of this laboratory assignment.

If we take a look at the source code of heat.c (where the solver is invoked) and solver.c (where the solvers are coded). We will soon realise that both solvers use the same function solve. The difference is that Jacobi uses a temporary matrix to store the new computed matrix (param.uhelp) while Gauss-Seidel directly updates the same matrix (param.u).

We have to notice that function solve is iteratively invoked inside a while loop that iterates while two different conditions are met:
1) the maximum number of iterations param.maxiter is not reached;
2) the value returned by the solver is larger than param.residual.

And also that at each iteration of the while loop Jacobi needs to copy the newly computed matrix into the original one in order to repeat the process, by simply invoking function copy mat also defined inside solver.c.

The result is a heat diffusion (Figures 2 and 4) when the two heat sources are placed in the borders of the 2D solid.

2 Task decomposition strategies

By executing *./heat test.dat -a 0 -o heat-jacobi.ppm* we obtain the output shown at Figure 1.

```
par1302@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: 4.664
Flops and Flops per second: (11.182 GFlop => 2397.25 MFlop/s)
Convergence to residual=0.000050: 15756 iterations
```

Figure 1: Output of `./heat test.dat -a 0 -o heat-jacobi.ppm`

And the picture associated to this output is the next one (we will keep it in mind to compare futures executions and in order to check later the correctness of the parallel versions you will program):

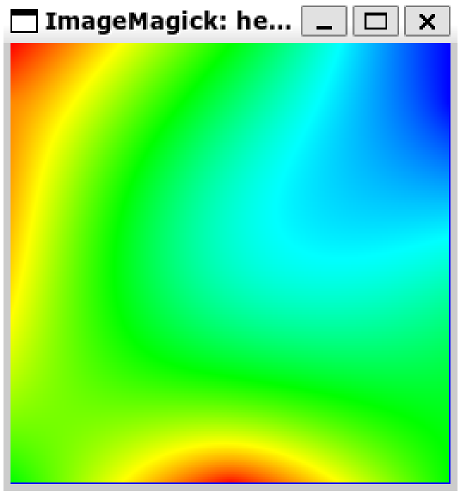


Figure 2: Heat map of Jacobi first execution

By executing the same as before but now with Gauss we obtain:

```
par1302@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: 6.038
Flops and Flops per second: (8.806 GFlop => 1458.51 MFlop/s)
Convergence to residual=0.000050: 12409 iterations
```

Figure 3: Output of `./heat test.dat -a 1 -o heat-gauss.ppm`

The heat map associated is the one shown at Figure 4.

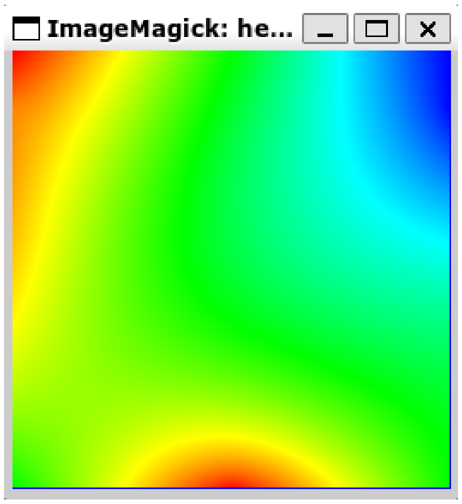


Figure 4: Heat map of Guass first execution

As we can see, the sequential version of Jacobi is quicker than the Gauss-Seidel because the execution time of the first one is 4.67 seconds and the second one 6.04 seconds.

2.1 Analysis of task granularities and dependences

Let's see the task dependence graph generated by the first version given of Tareador.

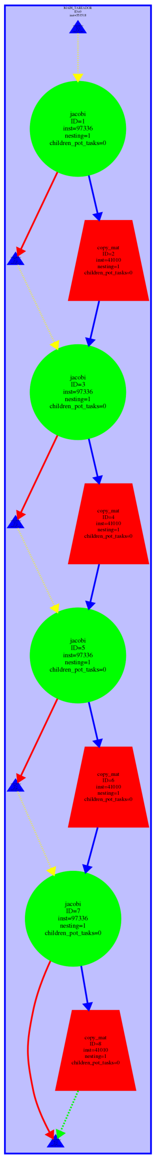


Figure 5: Dependence graph of Jacobi (original version given)

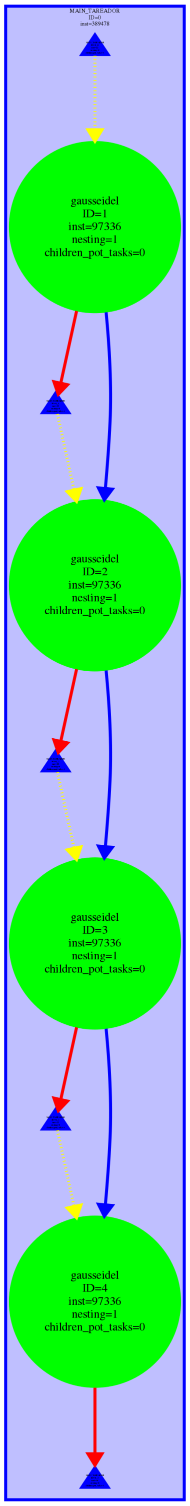


Figure 6: Dependence graph of Gauss-Seidel (original version given)

By observing only the task dependence generated in both cases we can say that there isn't any parallelism that can be exploited at this granularity level (by checking the simulation with any number of processors wanted we see it faster).

That's why we have changed the code, adding `tareador_start_task()` and `tareador_end_task()` inside `solve` and `copy_mat` functions for generating a task per block. These changes can be seen in solver-tareador-one-task-per-block.c delivered.

In `solve` function, we have put the clauses before the two fors. These fors iterate through a block which their size is defined by the minimum between the limits of the matrix and the size that we define for a block in the variables nblocksi and nblocksj. Furthermore, the two previous fors before the Tareador clauses declarations calculate the limits of the block. Figures 7 and 8 show the new dependence graph for Jacobi and Gauss-Seidel with one task per block implementation.

The Jacobi has two groups of tasks that repeats continuously. One calculates the result and the other corresponds to the function that copies the result matrix. However, Gauss-Seidel has different groups of tasks semi-sequential executed but with more dependencies.

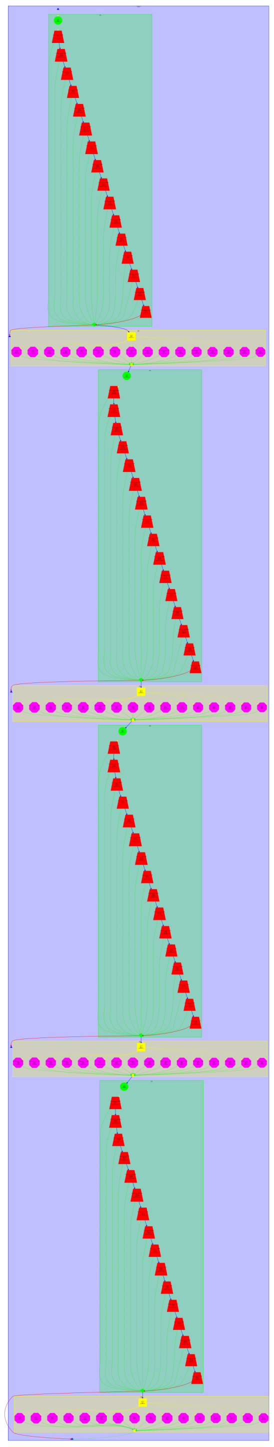


Figure 7: Dependence graph of Jacobi (one task per block)

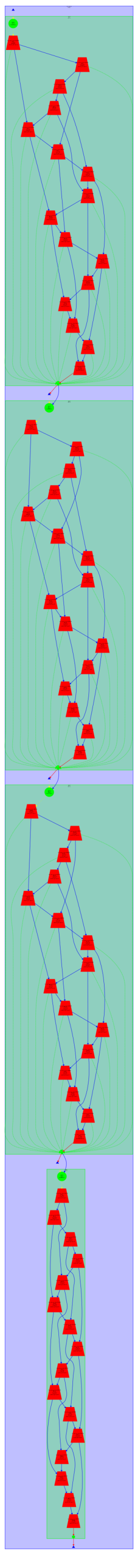


Figure 8: Dependence graph of Gauss-Seidel (one task per block)

After analyzing the code and the Dataview option in Tareador, we have found that variable sum is the variable that is causing the serialization of all the tasks.

In order to emulate the effect of protecting the dependences caused by this variable, we have used the `tareador_disable_object` and `tareador_enable_object` calls, already introduced in the code as comments. With these calls we are telling to Tareador to filter the dependences caused by the variable sum. The code modified is included in the delivery: solver-tareador-enable-disable-object.c.

Figures 9 and 10 show the dependence graphs. The number of tasks are the same for Jacobi and Gauss-Seidel methods. In Jacobi we have achieved the parallelisation of the red tasks which are the ones that calculate the results, which was sequential (viewed in Figure 7).

Furthermore, the Gauss-Seidel method has improved in parallelization even though it is not as good as Jacobi. That is because there are some dependencies between chunks of data because of writing the result in the same matrix.

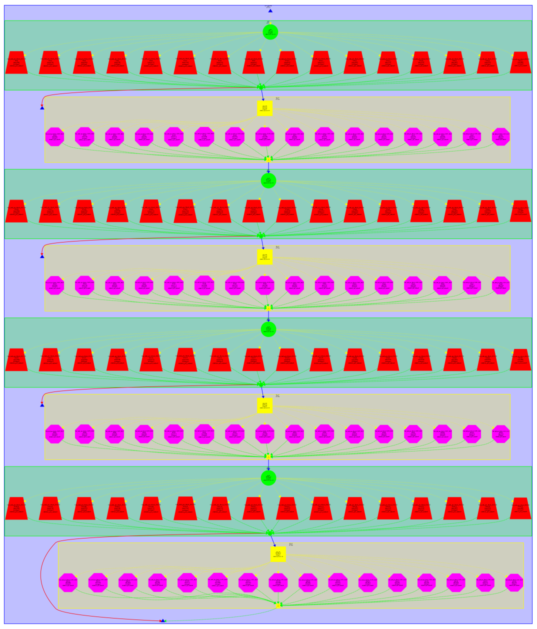


Figure 9: Dependence graph of Jacobi (one task per block), enable and disable object

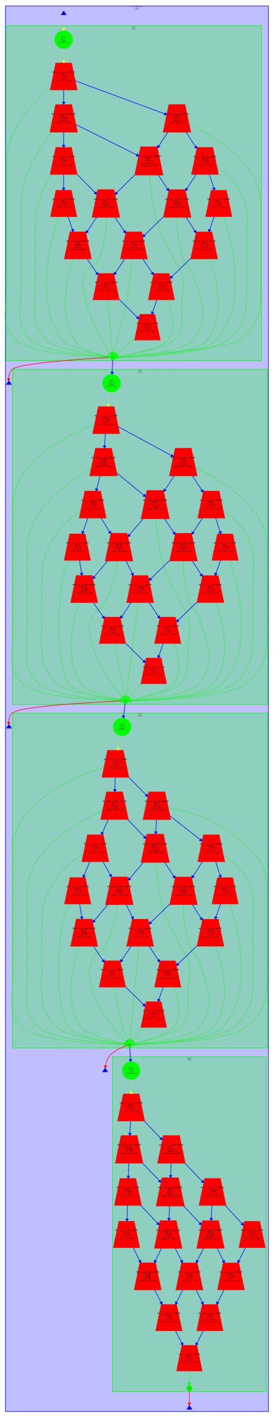


Figure 10: Dependence graph of Gauss-Seidel (one task per block), enable and disable object

Now we have simulated the execution with 4 processors (4 threads). In Jacobi method, each thread executes four tasks. Red tasks are the ones that calculates the results.
The Gauss-Seidel graph shows the same structure that the task dependence graph. At the beginning only one thread can execute a task. When it ends, it can be executed two more, after these two, three tasks can be now executed and what it happens is that when it arrives to four tasks, the number of tasks executing start decreasing until we arrive to one task; and then starts again.

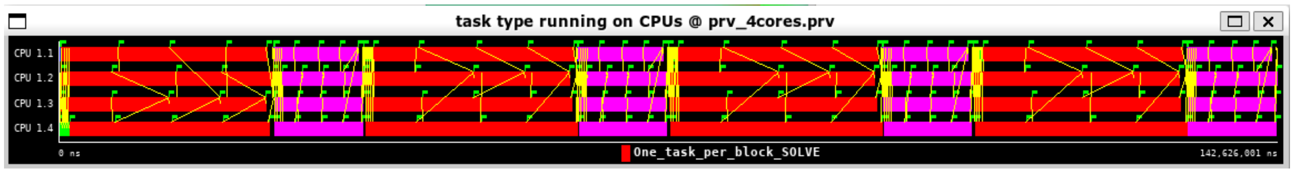


Figure 11: Tareador running simulator Jacobi with 4 processors

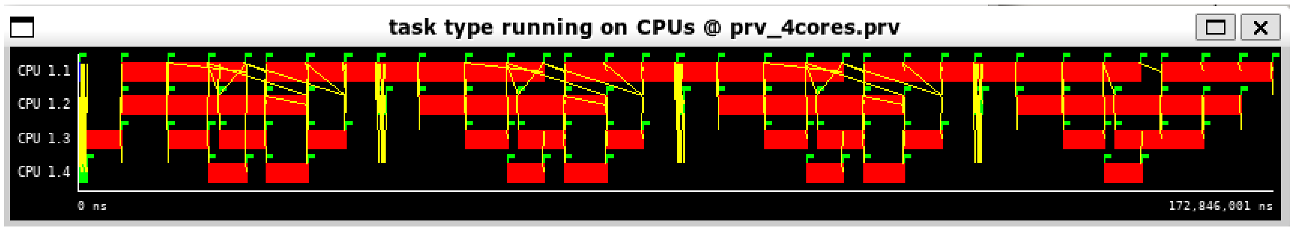


Figure 12: Tareador running simulator Gauss-Seidel with 4 processors

3 Implementation in *OpenMP* and performance analysis

3.1 Jacobi solver

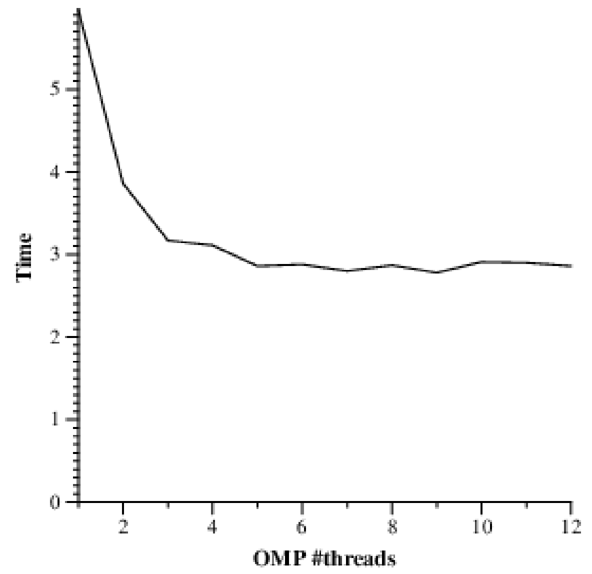
By only parallelizing the first solve function adding `#pragma_omp_parallel private(tmp, diff)reduction (+:sum)` (see solver-omp-Jacobi-v1.c) we have achieved an execution time of 2.954 seconds (see Figure 13). It has been an improvement because the sequential time seen in the first section was 4.664 seconds. Jacobi uses an auxiliary matrix and it doesn't have dependencies between computations. That's why we have privatizated the variables tmp and diff which are declared outside the area of the pragma. The reduction to the variable sum avoids the data race between threads.

The distribution of data blocks between threads consists on start dividing rows depending on the number of threads, and then, for each of them, is computed all the columns of the corresponding row.

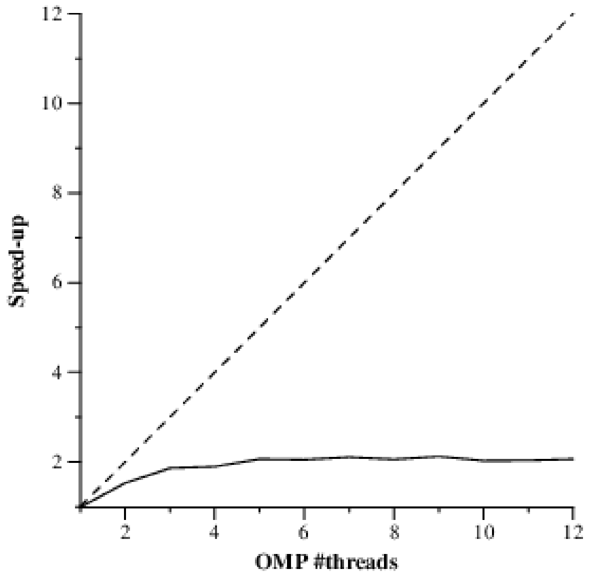
We have validated the parallelisation by visually inspecting the image generated and making a `diff` with the file generated with the original sequential version.

```
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: 2.954
Flops and Flops per second: (11.182 GFlop => 3784.89 MFlop/s)
Convergence to residual=0.000050: 15756 iterations
```

Figure 13: Output of `sbatch submit-omp.sh heat-omp 0 8` (v1)



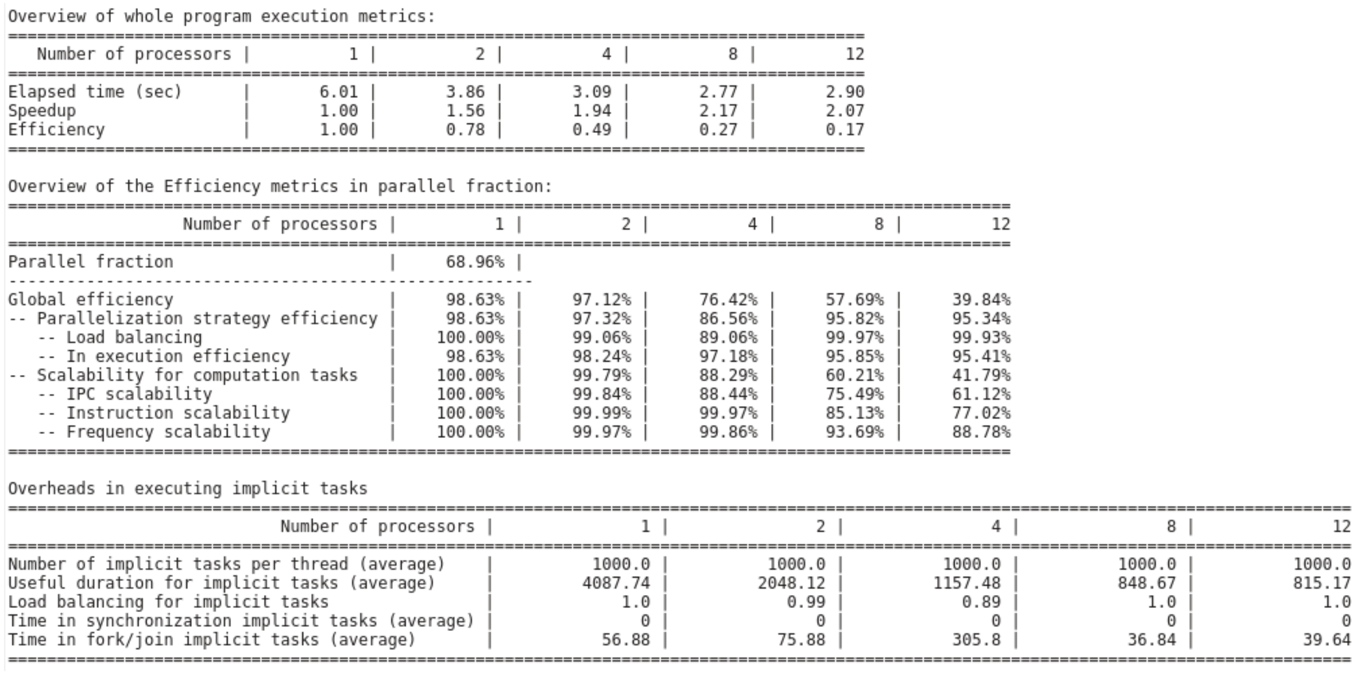
par1302
Average elapsed execution time
Fri May 27 08:44:01 CEST 2022



par1302
Speed-up wrt sequential time
Fri May 27 08:44:01 CEST 2022

Figure 14: Scalability plots for Jacobi v1

The scalability plots obtained are not appropriate because the speed-up increases a little bit but then doesn't increase. The output of mdelfactors.py shows that the execution time is divided in two if we put two processors; however, if we increase the number of processors, the elapsed time stays basically constant.



```
Overview of whole program execution metrics:
=================================================================================
   Number of processors |          1 |          2 |          4 |          8 |         12
=================================================================================
Elapsed time (sec)      |       6.01 |       3.86 |       3.09 |       2.77 |       2.90
Speedup                 |       1.00 |       1.56 |       1.94 |       2.17 |       2.07
Efficiency              |       1.00 |       0.78 |       0.49 |       0.27 |       0.17
=================================================================================

Overview of the Efficiency metrics in parallel fraction:
===============================================================================================
                  Number of processors |          1 |          2 |          4 |          8 |         12
===============================================================================================
Parallel fraction                      |     68.96% |
-----------------------------------------------------
Global efficiency                      |     98.63% |     97.12% |     76.42% |     57.69% |     39.84%
-- Parallelization strategy efficiency |     98.63% |     97.32% |     86.56% |     95.82% |     95.34%
   -- Load balancing                   |    100.00% |     99.06% |     89.06% |     99.97% |     99.93%
   -- In execution efficiency          |     98.63% |     98.24% |     97.18% |     95.85% |     95.41%
-- Scalability for computation tasks   |    100.00% |     99.79% |     88.29% |     60.21% |     41.79%
   -- IPC scalability                  |    100.00% |     99.84% |     88.44% |     75.49% |     61.12%
   -- Instruction scalability          |    100.00% |     99.99% |     99.97% |     85.13% |     77.02%
   -- Frequency scalability            |    100.00% |     99.97% |     99.86% |     93.69% |     88.78%
===============================================================================================

Overheads in executing implicit tasks
==============================================================================================================================
                            Number of processors |                 1 |                 2 |                 4 |                 8 |                12
==============================================================================================================================
Number of implicit tasks per thread (average)    |            1000.0 |            1000.0 |            1000.0 |            1000.0 |            1000.0
Useful duration for implicit tasks (average)     |           4087.74 |           2048.12 |           1157.48 |            848.67 |            815.17
Load balancing for implicit tasks                |               1.0 |              0.99 |              0.89 |               1.0 |               1.0
Time in synchronization implicit tasks (average) |                 0 |                 0 |                 0 |                 0 |                 0
Time in fork/join implicit tasks (average)       |             56.88 |             75.88 |             305.8 |             36.84 |             39.64
==============================================================================================================================
```

Figure 15: Output of modelfactors.py Jacobi v1

We have opened with Paraver the trace that has been generated for 8 threads and we have observed that there are some black parts:

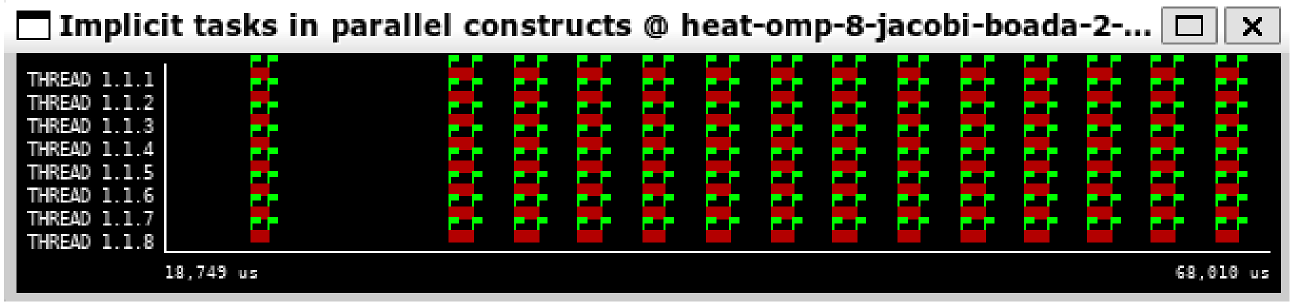
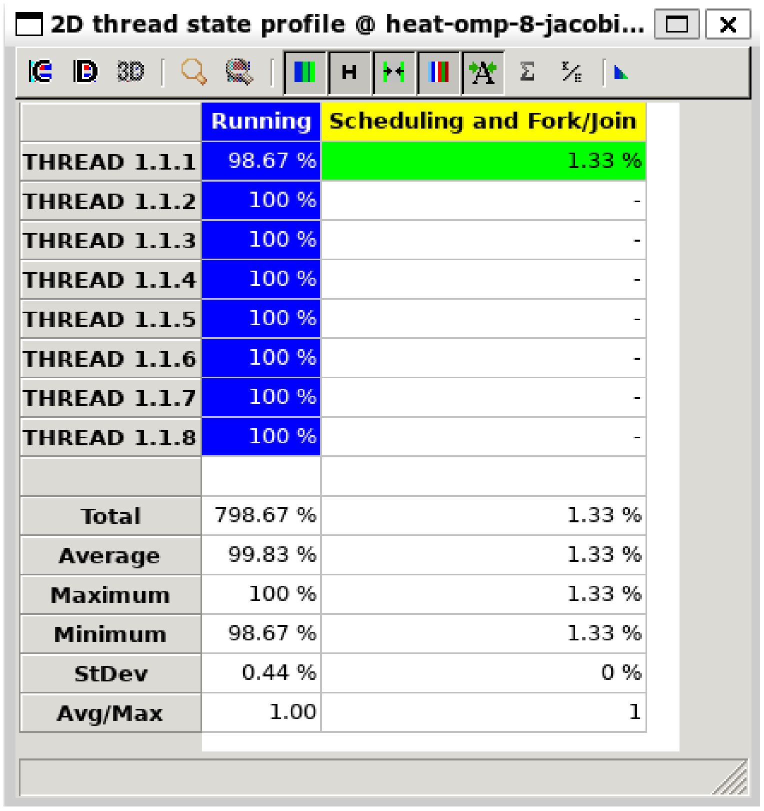


Figure 16: Implicit tasks in parallel constructs 8 processors, Jacobi v1 (zoomed in)

Also, the 2D thread state profile with 8 processors shows that there is only one thread that spends time in scheduling and fork/join.



2D thread state profile @ heat-omp-8-jacobi...

	Running	Scheduling and Fork/Join
THREAD 1.1.1	98.67 %	1.33 %
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 %	-
Total	798.67 %	1.33 %
Average	99.83 %	1.33 %
Maximum	100 %	1.33 %
Minimum	98.67 %	1.33 %
StDev	0.44 %	0 %
Avg/Max	1.00	1

Figure 17: 2D thread state profile Jacobi v1

Due to that we have an only a 68.96% of parallelisation, we have parallelized the region of the code that is provoking the low value for the parallel fraction: the `copy_mat` function. `solver-omp-Jacobi-v2.c` contains the same code of version 1 but `copy_mat` is parallelized using the same strategy seen before but without the privatization and the reduction.

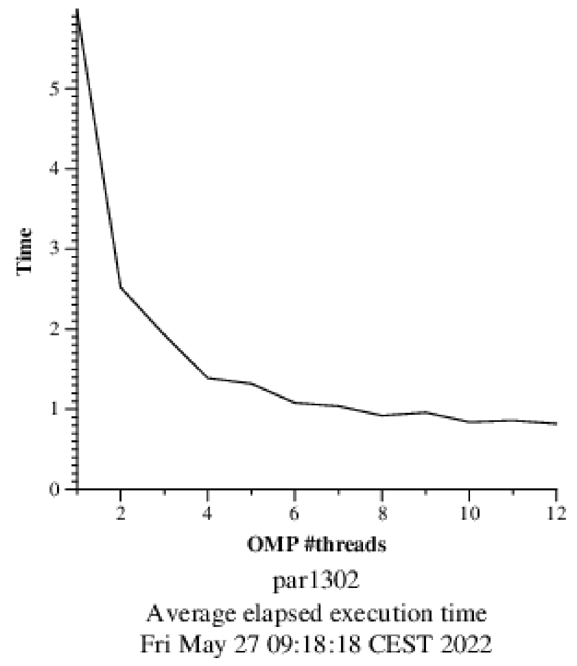
Now the execution time has reduced: from 2.954 seconds of version 1 to 0.753 seconds of this version 2 with 8 threads (see Figure 18).

```
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: 0.753
Flops and Flops per second: (11.182 GFlop => 14856.34 MFlop/s)
Convergence to residual=0.000050: 15756 iterations
```

Figure 18: Output of `sbatch submit-omp.sh heat-omp 0 8` (v2)

Scalability plots have improved (Figure 19). Now speed-up increases if we increase the number of threads. First it increases ideally but then from 5 threads it starts to deviate from the ideal line.

Modelfactors.py (Figure 20) shows that now there is a parallel fraction of 98.85% and the execution time decreases up to 0.79 seconds with 12 processors. Furthermore, the speed-up increases until arriving to a value of 7.63 with 12 processors. However, in version 1, the speedup with 12 processors was 2. With 2 and 4 processors, the efficiency has been above 1. Also, the average number of implicit tasks per tread is 2000 whereas in version 1 the average was 1000.



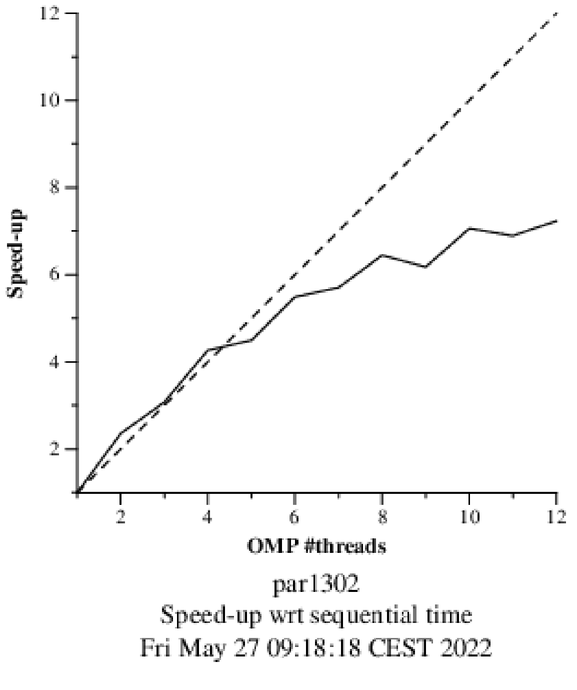


Figure 19: Scalability plots for Jacobi v2

```
Overview of whole program execution metrics:
===========================================================================
   Number of processors |          1 |         2 |         4 |         8 |        12
===========================================================================
Elapsed time (sec)      |       6.03 |      2.54 |      1.38 |      0.95 |      0.79
Speedup                 |       1.00 |      2.38 |      4.35 |      6.37 |      7.63
Efficiency              |       1.00 |      1.19 |      1.09 |      0.80 |      0.64
===========================================================================

Overview of the Efficiency metrics in parallel fraction:
==========================================================================================
                 Number of processors |          1 |         2 |         4 |         8 |        12
==========================================================================================
Parallel fraction                     |     98.85% |
---------------------------------------------------
Global efficiency                     |     98.96% |   118.76% |   110.55% |    82.90% |    66.77%
-- Parallelization strategy efficiency |     98.96% |    98.14% |    96.86% |    95.05% |    94.55%
   -- Load balancing                  |    100.00% |    99.24% |    99.39% |    98.68% |    98.19%
   -- In execution efficiency         |     98.96% |    98.90% |    97.45% |    96.33% |    96.30%
-- Scalability for computation tasks  |    100.00% |   121.01% |   114.13% |    87.21% |    70.62%
   -- IPC scalability                 |    100.00% |   121.01% |   114.78% |    94.51% |    79.83%
   -- Instruction scalability         |    100.00% |    99.98% |    99.95% |    97.75% |    97.95%
   -- Frequency scalability           |    100.00% |   100.02% |    99.48% |    94.40% |    90.31%
==========================================================================================

Overheads in executing implicit tasks
======================================================================================================================
                              Number of processors |            1 |            2 |            4 |            8 |           12
======================================================================================================================
Number of implicit tasks per thread (average)      |       2000.0 |       2000.0 |       2000.0 |       2000.0 |       2000.0
Useful duration for implicit tasks (average)       |      2947.25 |      1217.76 |       645.59 |       422.44 |        347.8
Load balancing for implicit tasks                  |          1.0 |         0.99 |         0.99 |         0.99 |         0.98
Time in synchronization implicit tasks (average)   |            0 |            0 |            0 |            0 |            0
Time in fork/join implicit tasks (average)         |        30.88 |        32.45 |        23.07 |        28.65 |        27.83
======================================================================================================================
```

Figure 20: Output of modelfactors.py Jacobi v2

The implicit tasks in parallel constructs have less black spaces compared to version 1.

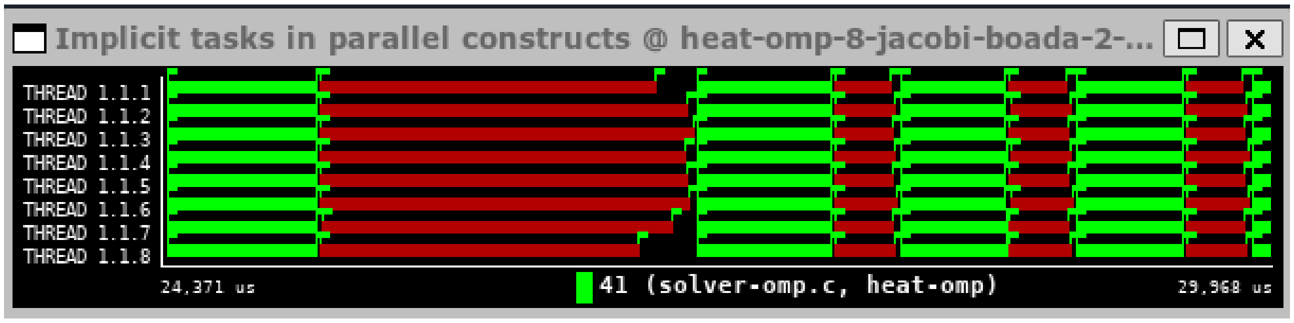
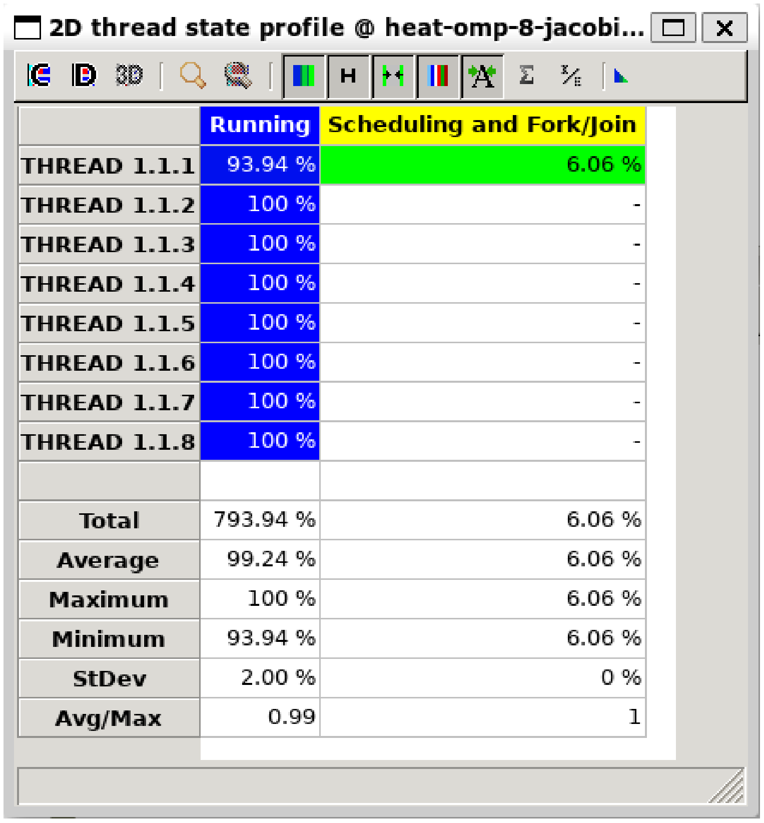


Figure 21: Implicit tasks in parallel constructs 8 processors, Jacobi v2 (zoomed in)

The 2D thread state profile is very similar to version 1, except that in our execution the thread that creates all threads spend a bit more in scheduling and fork/join instead in running state.



	Running	Scheduling and Fork/Join
THREAD 1.1.1	93.94 %	6.06 %
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 %	-
Total	793.94 %	6.06 %
Average	99.24 %	6.06 %
Maximum	100 %	6.06 %
Minimum	93.94 %	6.06 %
StDev	2.00 %	0 %
Avg/Max	0.99	1

Figure 22: 2D thread state profile Jacobi v2

To conclude this section we can say that it is when `copy_mat` function is also parallelised that we can achieve the full potential parallelism and better results.

3.2 Gauss-Seidel

As the statement says, we have read the annex. With the given hints, we have decided to create a shared matrix `control` containing the execution status information of each block executed by a thread. The positions of the matrix represents a block from the original matrix, where each position can have two values: 0 for not finished tasks and 1 for finished tasks. This changes can be seen in `solver-omp-Gauss-v1.c` delivered.

As we are working with implicit tasks, it's not possible to use dependency clauses.

We have added a verification in the parallel region of each thread, after the calculations needed to know which block is going to use the thread, to know if it can be solved if the upper position is 1 or not. If the corresponding position of `control` that represents that block is ready, it will be 1. Otherwise, this thread will be checking that condition all the time inside the do while.

We have added a `pragma omp atomic write` when writing 1 and a `pragma omp atomic read` when reading the control matrix inside the do while for avoiding possible data race. When a thread finish its task writes 1 in the control matrix so that other threads can start their work.

By looking at the implicit tasks in parallel constructs graph (Figures 23 and 24) we can see a little delay before the beginning of the execution of each thread due to the need of waiting to the execution of the upper block, if there is one, and also the execution of their left block.

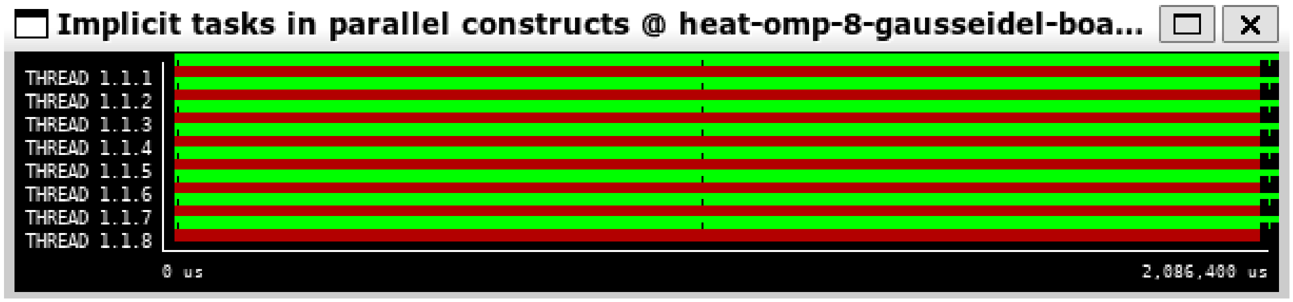


Figure 23: Implicit tasks in parallel constructs 8 processors, Gauss-Seidel

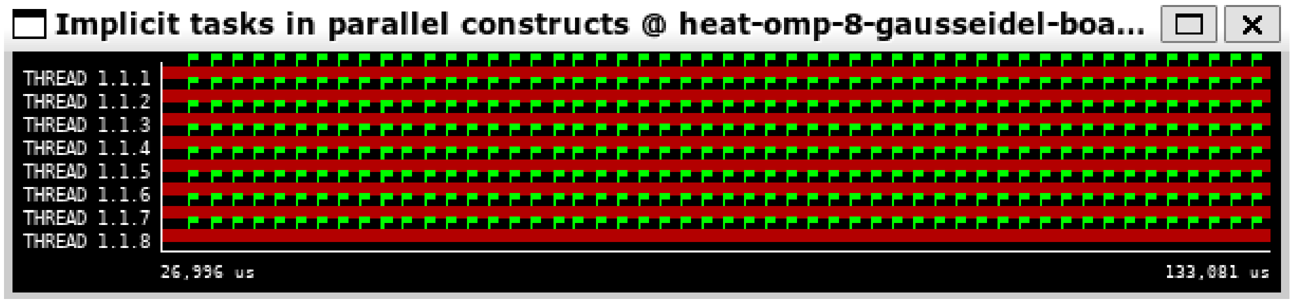
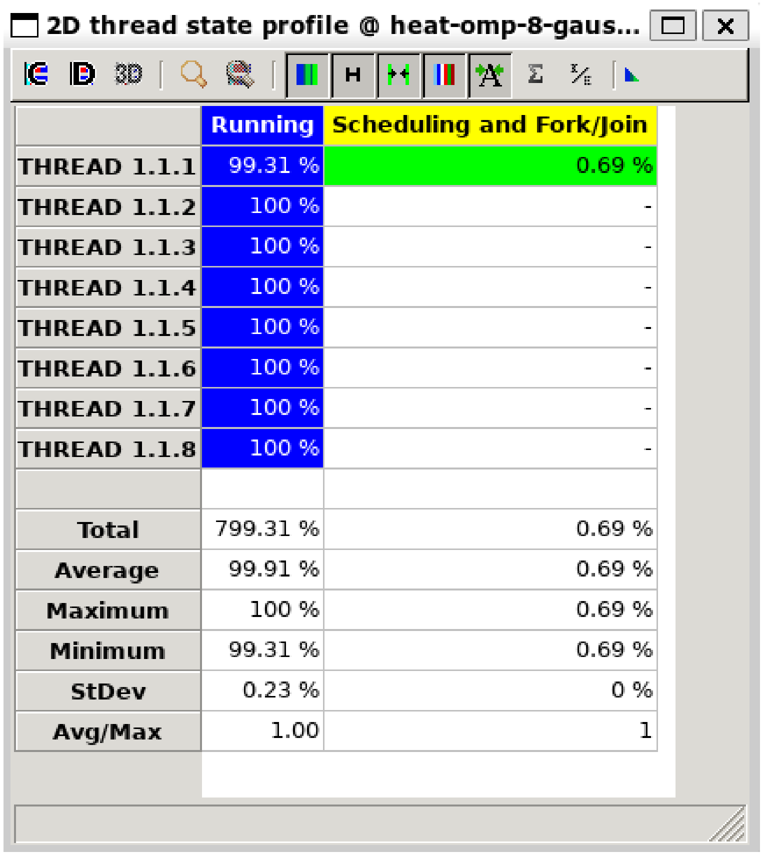


Figure 24: Implicit tasks in parallel constructs 8 processors, Gauss-Seidel (zoomed in)

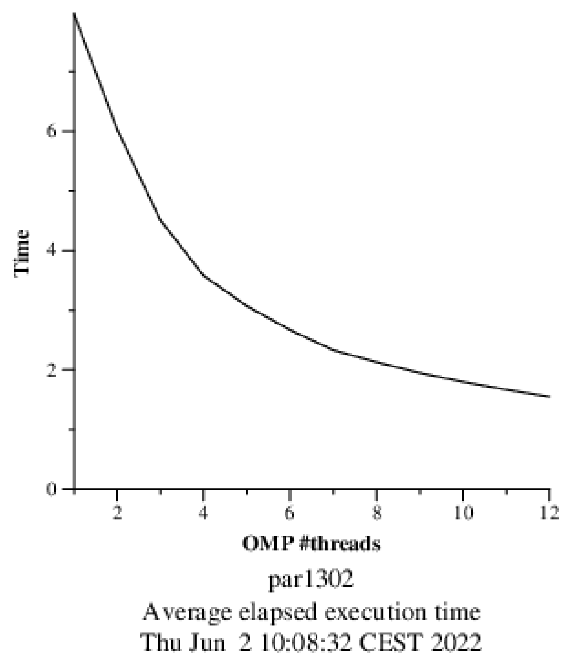
In Figure 25 we can see that the thread that creates other threads spends a few time in scheduling and Fork/Join state, whereas in Jacobi versions spent more time.



	Running	Scheduling and Fork/Join
THREAD 1.1.1	99.31 %	0.69 %
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 %	-
Total	799.31 %	0.69 %
Average	99.91 %	0.69 %
Maximum	100 %	0.69 %
Minimum	99.31 %	0.69 %
StDev	0.23 %	0 %
Avg/Max	1.00	1

Figure 25: 2D thread state profile Gauss-Seidel

Looking at speed-up and execution time plots for each number of threads (Figure 26), we can observe that the execution time decreases and the speed-up increases almost constant if we increase the number of processors. However, the speed-up line isn't close to the ideal speed-up line. These plots are better than Jacobi version 1 but worse than Jacobi version 2.



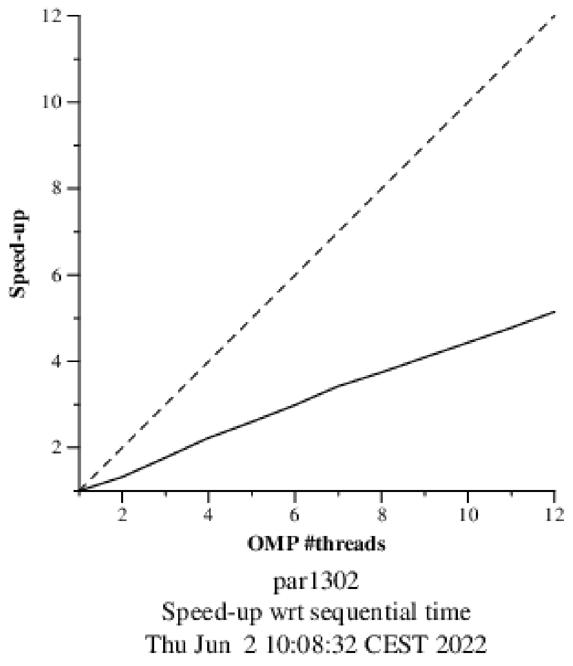
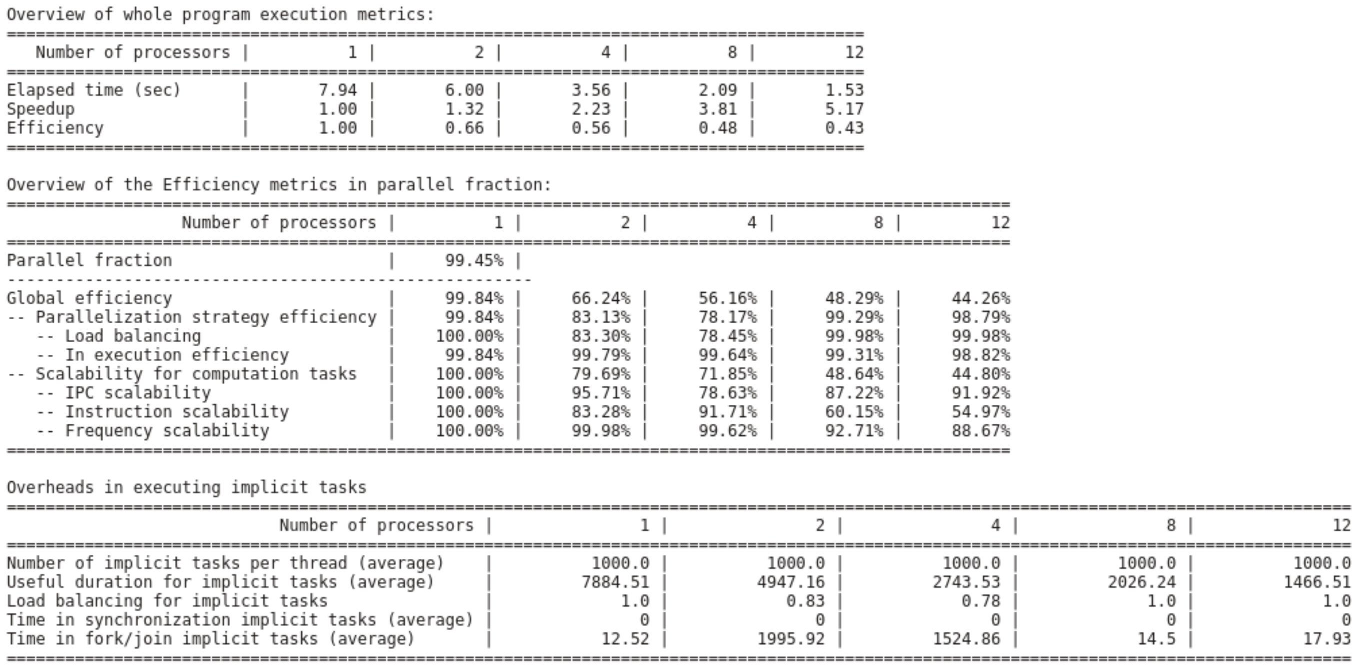


Figure 26: Scalability plots for Gauss-Seidel

If we take a look at modelfactors.out, we can see that for 12 processors we have a speed-up of 5.17 and a parallel efficiency of 0.43. This is less than Jacobi version 2. The sequential time is almost 8 seconds, but it is being reduced if the number of processors is increased.
The parallel fraction is about 99% and the average number of implicit tasks per thread are 1000 whereas in Jacobi version 2 was 2000.



```
Overview of whole program execution metrics:
=====================================================================================
   Number of processors |        1 |        2 |        4 |        8 |       12
=====================================================================================
Elapsed time (sec)      |     7.94 |     6.00 |     3.56 |     2.09 |     1.53
Speedup                 |     1.00 |     1.32 |     2.23 |     3.81 |     5.17
Efficiency              |     1.00 |     0.66 |     0.56 |     0.48 |     0.43
=====================================================================================

Overview of the Efficiency metrics in parallel fraction:
=====================================================================================================
                  Number of processors |        1 |        2 |        4 |        8 |       12
=====================================================================================================
Parallel fraction                      |   99.45% |
------------------------------------------------------
Global efficiency                      |   99.84% |   66.24% |   56.16% |   48.29% |   44.26%
-- Parallelization strategy efficiency |   99.84% |   83.13% |   78.17% |   99.29% |   98.79%
   -- Load balancing                   |  100.00% |   83.30% |   78.45% |   99.98% |   99.98%
   -- In execution efficiency          |   99.84% |   99.79% |   99.64% |   99.31% |   98.82%
-- Scalability for computation tasks   |  100.00% |   79.69% |   71.85% |   48.64% |   44.80%
   -- IPC scalability                  |  100.00% |   95.71% |   78.63% |   87.22% |   91.92%
   -- Instruction scalability          |  100.00% |   83.28% |   91.71% |   60.15% |   54.97%
   -- Frequency scalability            |  100.00% |   99.98% |   99.62% |   92.71% |   88.67%
=====================================================================================================

Overheads in executing implicit tasks
=======================================================================================================================================
                            Number of processors |                 1 |                 2 |                 4 |                 8 |                12
=======================================================================================================================================
Number of implicit tasks per thread (average)    |            1000.0 |            1000.0 |            1000.0 |            1000.0 |            1000.0
Useful duration for implicit tasks (average)     |           7884.51 |           4947.16 |           2743.53 |           2026.24 |           1466.51
Load balancing for implicit tasks                |               1.0 |              0.83 |              0.78 |               1.0 |               1.0
Time in synchronization implicit tasks (average) |                 0 |                 0 |                 0 |                 0 |                 0
Time in fork/join implicit tasks (average)       |             12.52 |           1995.92 |           1524.86 |              14.5 |             17.93
=======================================================================================================================================
```

Figure 27: Output of modelfactors.py Gauss-Seidel

In order to exploit more parallelism in `solver`'s execution, now `nblocksj` is declared as `userparam* nblocksi`. `userparam` is the global variable that is received through an argument in the command-line execution with -u option. This changes can be seen in `solver-omp-Gauss-with-userparam.c`

Here we have the output of the Gauss-Seidel execution with 8 threads. As we can see it takes 1.637 seconds to be executed; more than Jacobi v2 but less than v1:

```
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: 1.637
Flops and Flops per second: (8.806 GFlop => 5378.52 MFlop/s)
Convergence to residual=0.000050: 12409 iterations
```

Figure 28: Output of `sbatch submit-omp.sh heat-omp 1 8`

Next figures show the average execution time for these changes we've done. Time elapsed is expressed for different values of `userparam` and different number of threads: 2, 4, 8 and 12.

Through these plots, we can extract that when `nblocksj` is bigger, the execution time decreases. If we increase the number of threads used, the execution time is better and better. However, de execution time doesn't decrease constantly if the number of threads are increased. For example, the difference between 8 threads and 12 is not that much.

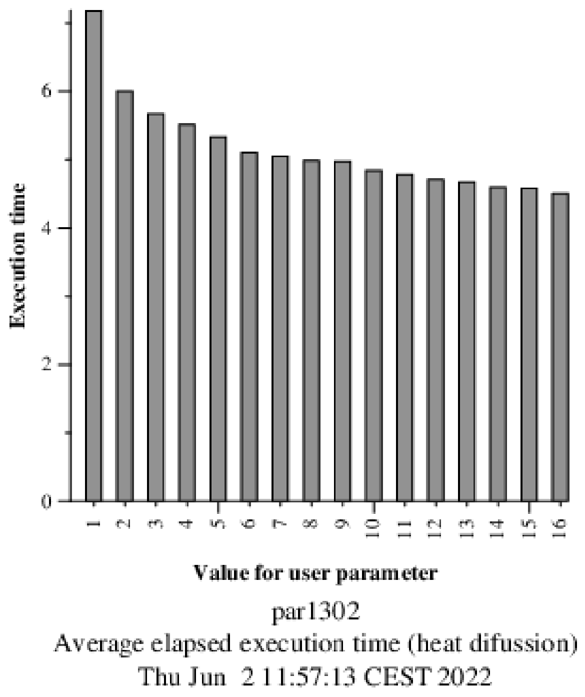


Figure 29: Gauss-Seidel execution time plot with 2 threads, using different values of userparam

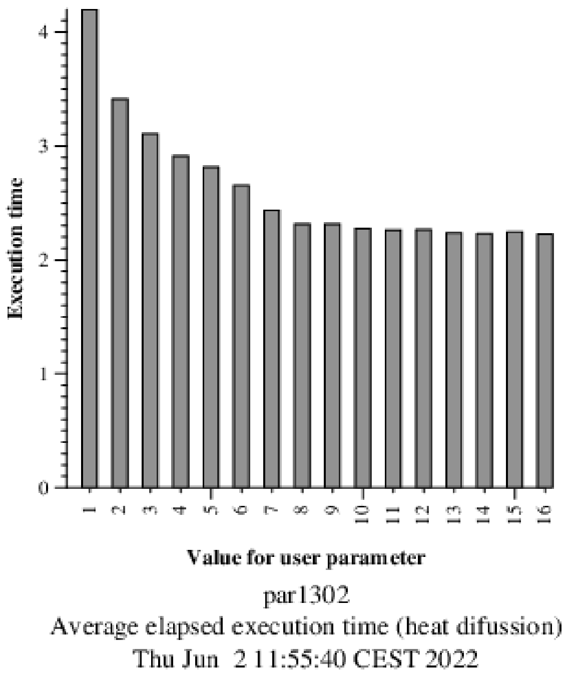


Figure 30: Gauss-Seidel execution time plot with 4 threads, using different values of userparam

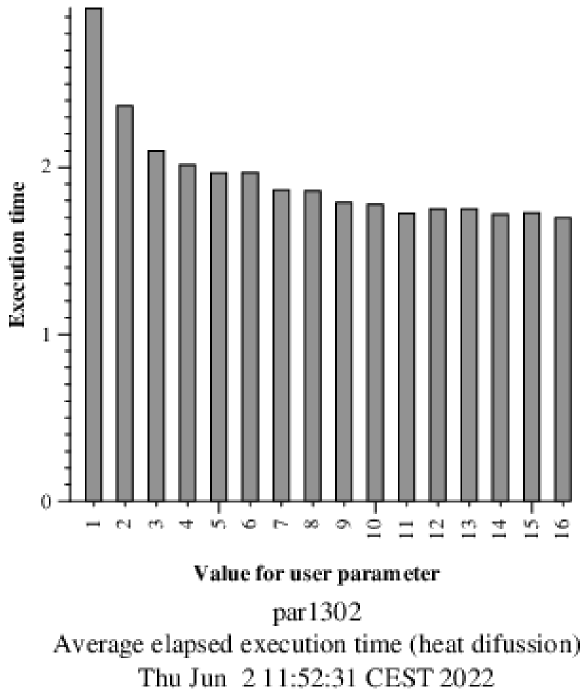


Figure 31: Gauss-Seidel execution time plot with 8 threads, using different values of userparam

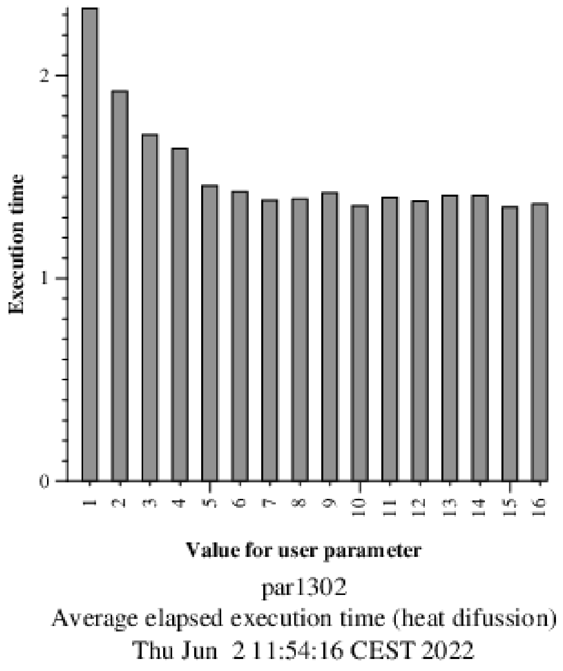


Figure 32: Gauss-Seidel execution time plot with 12 threads, using different values of userparam

4 Optionals

4.1 Optional 1

We have implemented the Jacobi solver using explicit tasks and following an iterative task decomposition (solver-omp-Optional-1.c). For achieving it, functions `solver` and `copy_mat` have been modified: we have replaced the initialization of `blocki` from a `omp_get_threads_num()` to a for that iterates over all the blocks. Furthermore, we added a `pragma_omp_single` after the parallel, for the further execution with one thread and then a `pragma_omp_taskloop` for parallelizing this loop. In `solver` function there are dependency clauses, whereas in `copy_mat` there aren't.

The execution time for this version is 1.89 seconds:

```
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: 1.890
Flops and Flops per second: (11.182 GFlop => 5917.00 MFlop/s)
Convergence to residual=0.000050: 15756 iterations
```

Figure 33: Output of `sbatch submit-omp.sh heat-omp 0 8`

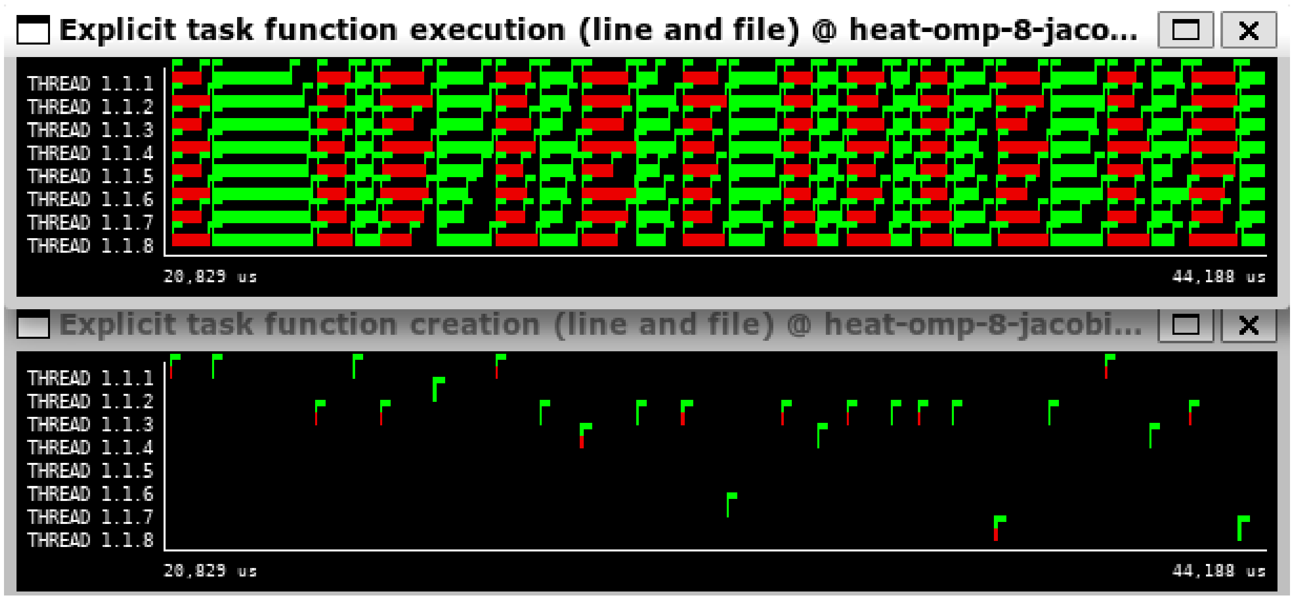


Figure 34: Explicit task function execution and creation 8 processors, Jacobi optional 1 (zoomed in)

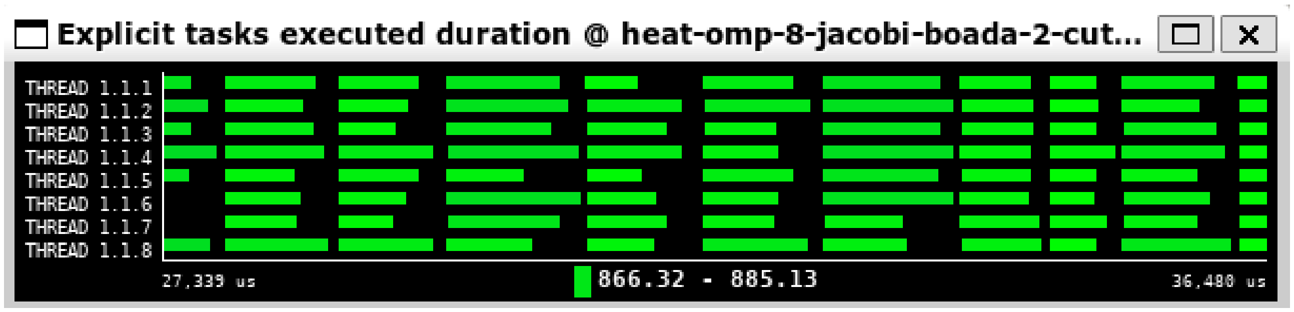
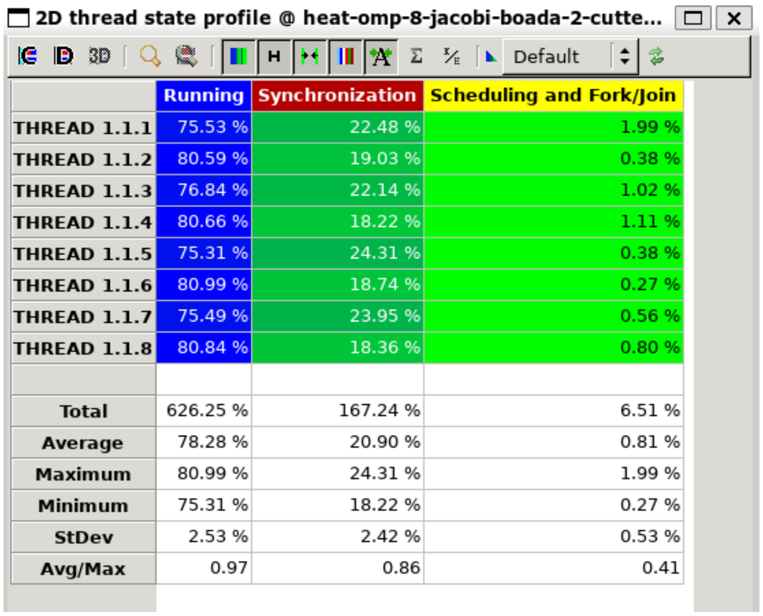


Figure 35: Explicit tasks executed duration 8 processors, Jacobi optional 1 (zoomed in)

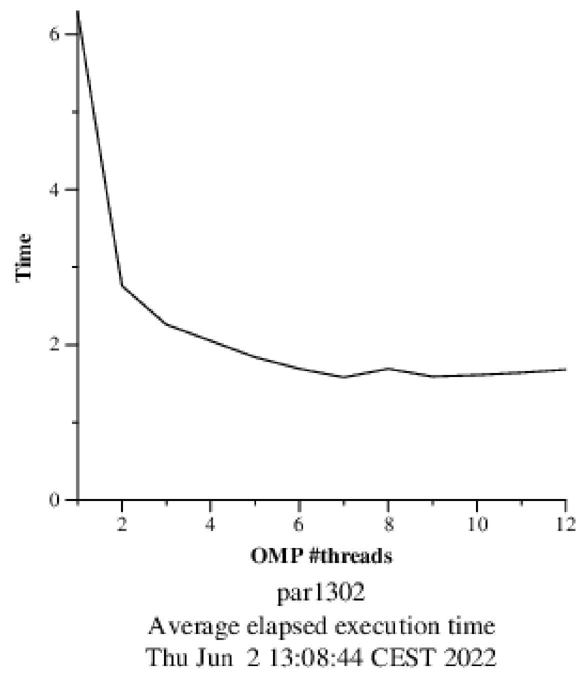
We can see in Figures 34 and 35 how tasks are generated, executed and their duration. It is not possible to compare these graphs with implicit tasks because threads work in a different way now. All threads spend more or less the same time working, seen also in Figure 36 (running state).
Because now we have explicit tasks, all threads spend time on scheduling and Fork/Join state. Also, in this version, there is an overhead that corresponds to the synchronization time.


2D thread state profile @ heat-omp-8-jacobi-boada-2-cutte...
Default

	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	75.53 %	22.48 %	1.99 %
THREAD 1.1.2	80.59 %	19.03 %	0.38 %
THREAD 1.1.3	76.84 %	22.14 %	1.02 %
THREAD 1.1.4	80.66 %	18.22 %	1.11 %
THREAD 1.1.5	75.31 %	24.31 %	0.38 %
THREAD 1.1.6	80.99 %	18.74 %	0.27 %
THREAD 1.1.7	75.49 %	23.95 %	0.56 %
THREAD 1.1.8	80.84 %	18.36 %	0.80 %
Total	626.25 %	167.24 %	6.51 %
Average	78.28 %	20.90 %	0.81 %
Maximum	80.99 %	24.31 %	1.99 %
Minimum	75.31 %	18.22 %	0.27 %
StDev	2.53 %	2.42 %	0.53 %
Avg/Max	0.97	0.86	0.41

Figure 36: 2D thread state profile Jacobi explicit tasks

In scalability plots and modelfactors.py (Figures 37 and 38) we can see that due to the overhead explained previously, the speed-up is worse than Jacobi with implicit tasks and the number of explicit tasks executed increases if we increase the number of processors.



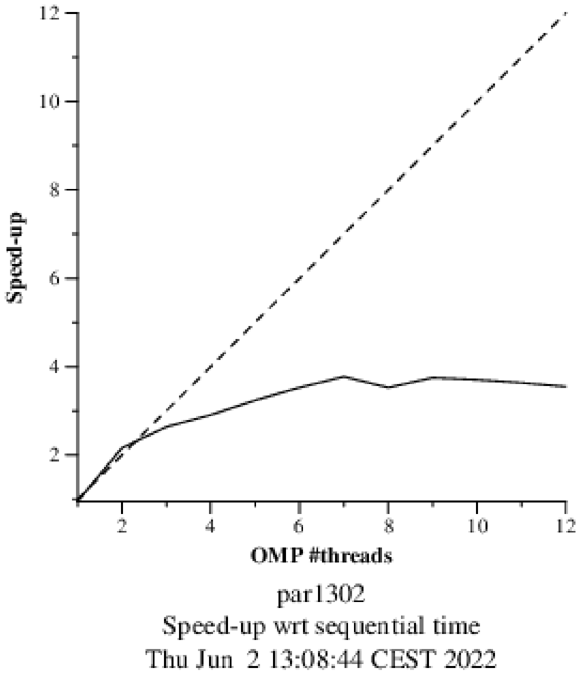
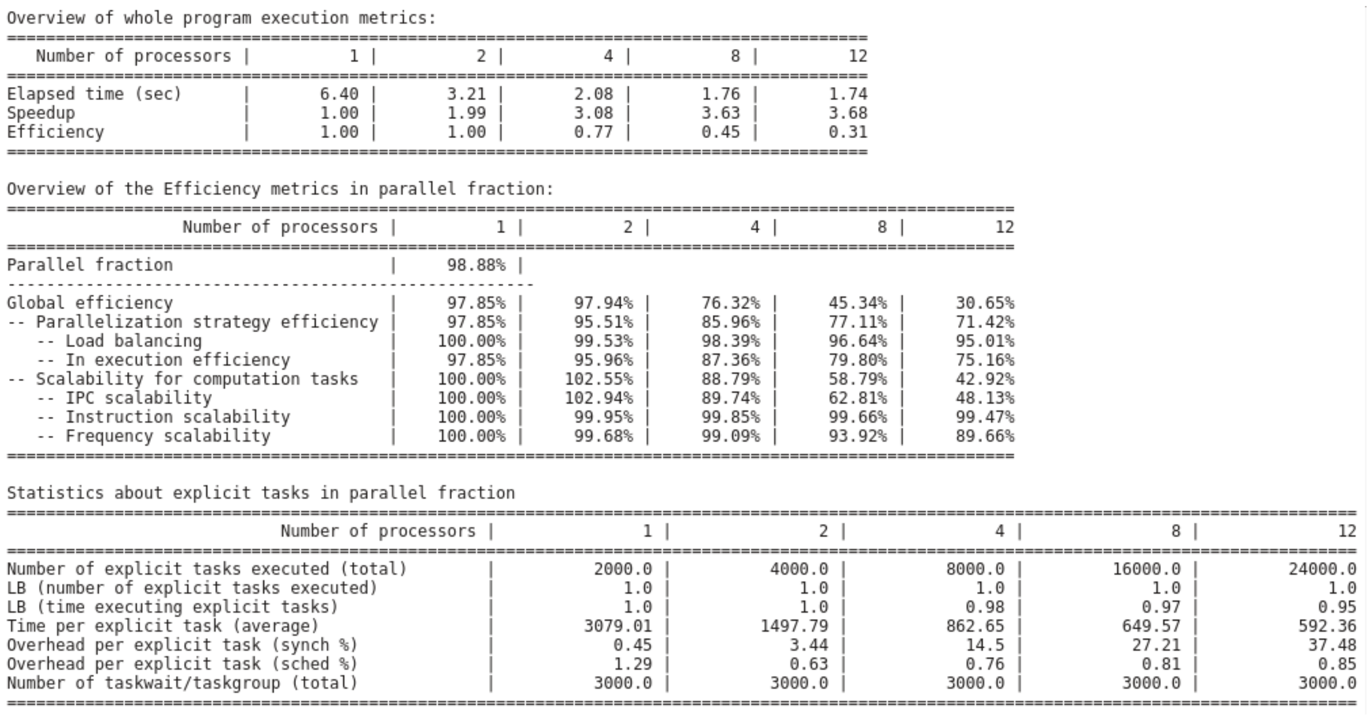


Figure 37: Scalability plots for Jacobi explicit tasks



```
Overview of whole program execution metrics:
==========================================================================================
   Number of processors |          1 |          2 |          4 |          8 |         12
==========================================================================================
Elapsed time (sec)      |       6.40 |       3.21 |       2.08 |       1.76 |       1.74
Speedup                 |       1.00 |       1.99 |       3.08 |       3.63 |       3.68
Efficiency              |       1.00 |       1.00 |       0.77 |       0.45 |       0.31
==========================================================================================

Overview of the Efficiency metrics in parallel fraction:
===========================================================================================================
                  Number of processors |          1 |          2 |          4 |          8 |         12
===========================================================================================================
Parallel fraction                      |     98.88% |
------------------------------------------------------
Global efficiency                      |     97.85% |     97.94% |     76.32% |     45.34% |     30.65%
-- Parallelization strategy efficiency |     97.85% |     95.51% |     85.96% |     77.11% |     71.42%
   -- Load balancing                   |    100.00% |     99.53% |     98.39% |     96.64% |     95.01%
   -- In execution efficiency          |     97.85% |     95.96% |     87.36% |     79.80% |     75.16%
-- Scalability for computation tasks   |    100.00% |    102.55% |     88.79% |     58.79% |     42.92%
   -- IPC scalability                  |    100.00% |    102.94% |     89.74% |     62.81% |     48.13%
   -- Instruction scalability          |    100.00% |     99.95% |     99.85% |     99.66% |     99.47%
   -- Frequency scalability            |    100.00% |     99.68% |     99.09% |     93.92% |     89.66%
===========================================================================================================

Statistics about explicit tasks in parallel fraction
=========================================================================================================================================
                            Number of processors |                 1 |                 2 |                 4 |                 8 |                12
=========================================================================================================================================
Number of explicit tasks executed (total)        |            2000.0 |            4000.0 |            8000.0 |           16000.0 |           24000.0
LB (number of explicit tasks executed)           |               1.0 |               1.0 |               1.0 |               1.0 |               1.0
LB (time executing explicit tasks)               |               1.0 |               1.0 |              0.98 |              0.97 |              0.95
Time per explicit task (average)                 |           3079.01 |           1497.79 |            862.65 |            649.57 |            592.36
Overhead per explicit task (synch %)             |              0.45 |              3.44 |              14.5 |             27.21 |             37.48
Overhead per explicit task (sched %)             |              1.29 |              0.63 |              0.76 |              0.81 |              0.85
Number of taskwait/taskgroup (total)             |            3000.0 |            3000.0 |            3000.0 |            3000.0 |            3000.0
=========================================================================================================================================
```

Figure 38: Output of modelfactors.py Jacobi explicit tasks

4.2 Optional 2

We have implemented the Gauss-Seidel solver using explicit tasks and task dependences, following an iterative task decomposition because we wanted to compare these performance results with the ones obtained with the data decomposition strategy.

We have added in `solve` and `copy_mat` functions a `#pragma_omp_parallel` and a `#pragma_omp_single`. Also, we have added a `#pragma omp taskloop` before the for. These changes are reflected in solver-omp-Optional-2.c delivered.

The execution time for this version is 2.183 seconds, more than the implicit tasks version (1.637 seconds).

```
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: 2.183
Flops and Flops per second: (8.806 GFlop => 4033.95 MFlop/s)
Convergence to residual=0.000050: 12409 iterations
```

Figure 39: Output of `sbatch submit-omp.sh heat-omp 1 8`

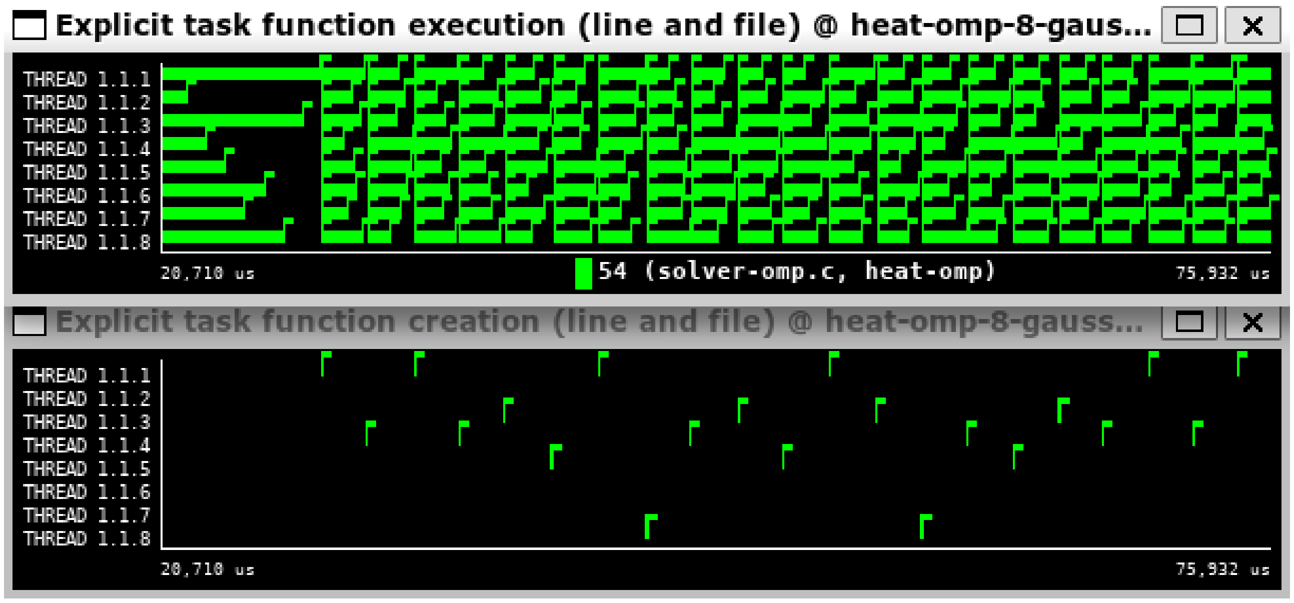


Figure 40: Explicit task function execution and creation 8 processors, Gauss-Seidel optional 2 (zoomed in)

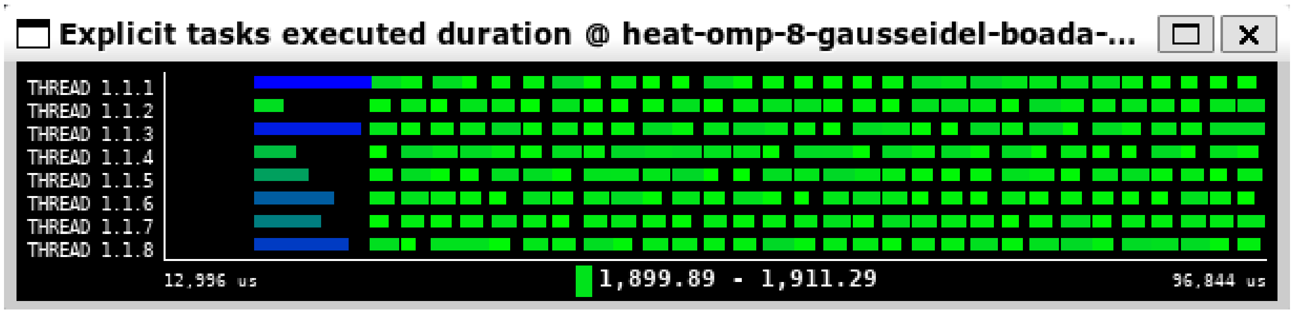


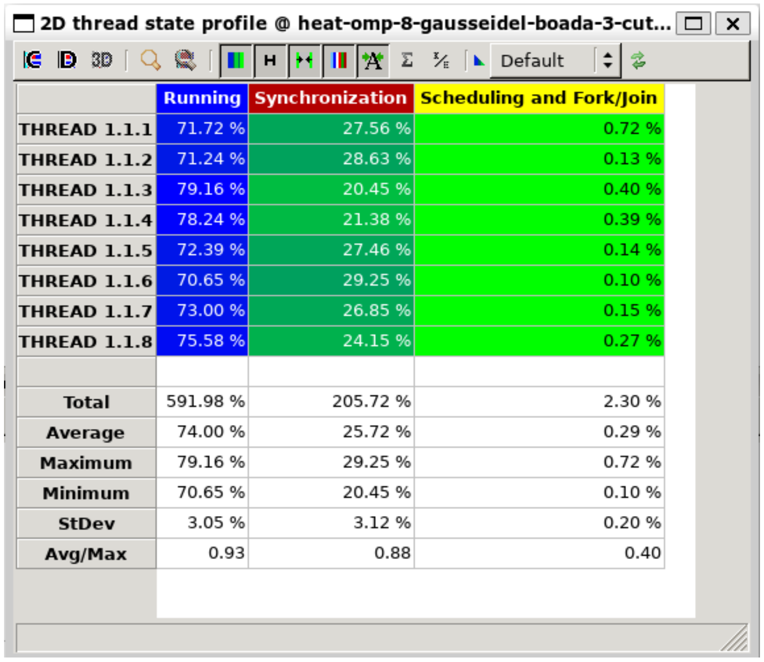
Figure 41: Explicit tasks executed duration 8 processors, Gauss-Seidel optional 2 (zoomed in)

We can see in Figure 41, different tasks has different durations. Figures 42 and 40 show the creation of the tasks and the execution with 8 processors.

The 2D thread state profile shows that now all tasks are in Scheduling and Fork/Join (Figure 42).

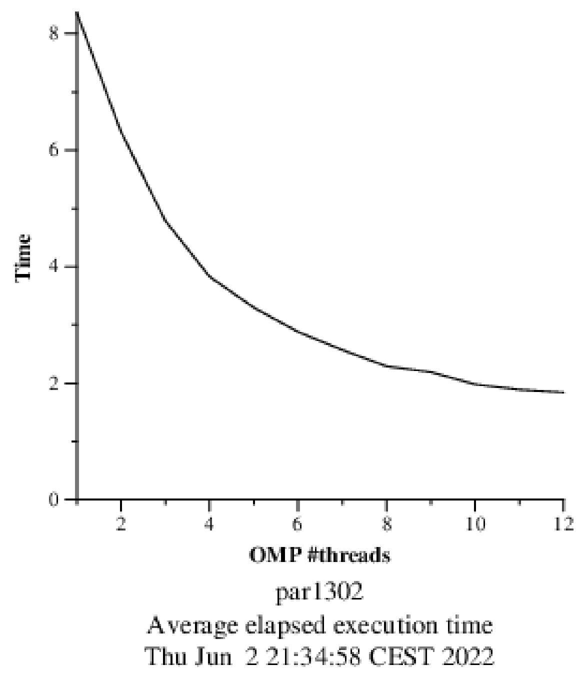
The speed-up plot is equal or a bit less than the version with implicit tasks.

Furthermore, modelfactors.out shows that the number of explicit tasks executed increases achieving 24000 explicit tasks with 12 processors.


2D thread state profile @ heat-omp-8-gausseidel-boada-3-cut...

	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	71.72 %	27.56 %	0.72 %
THREAD 1.1.2	71.24 %	28.63 %	0.13 %
THREAD 1.1.3	79.16 %	20.45 %	0.40 %
THREAD 1.1.4	78.24 %	21.38 %	0.39 %
THREAD 1.1.5	72.39 %	27.46 %	0.14 %
THREAD 1.1.6	70.65 %	29.25 %	0.10 %
THREAD 1.1.7	73.00 %	26.85 %	0.15 %
THREAD 1.1.8	75.58 %	24.15 %	0.27 %
Total	591.98 %	205.72 %	2.30 %
Average	74.00 %	25.72 %	0.29 %
Maximum	79.16 %	29.25 %	0.72 %
Minimum	70.65 %	20.45 %	0.10 %
StDev	3.05 %	3.12 %	0.20 %
Avg/Max	0.93	0.88	0.40

Figure 42: 2D thread state profile Gauss-Seidel explicit tasks



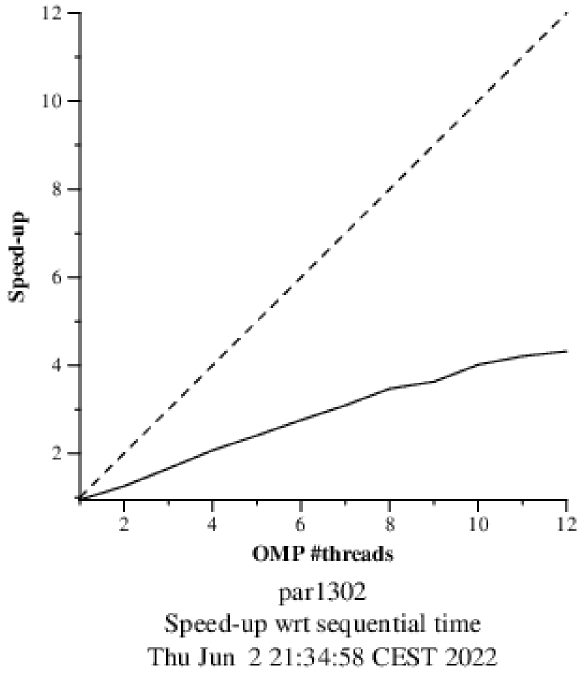
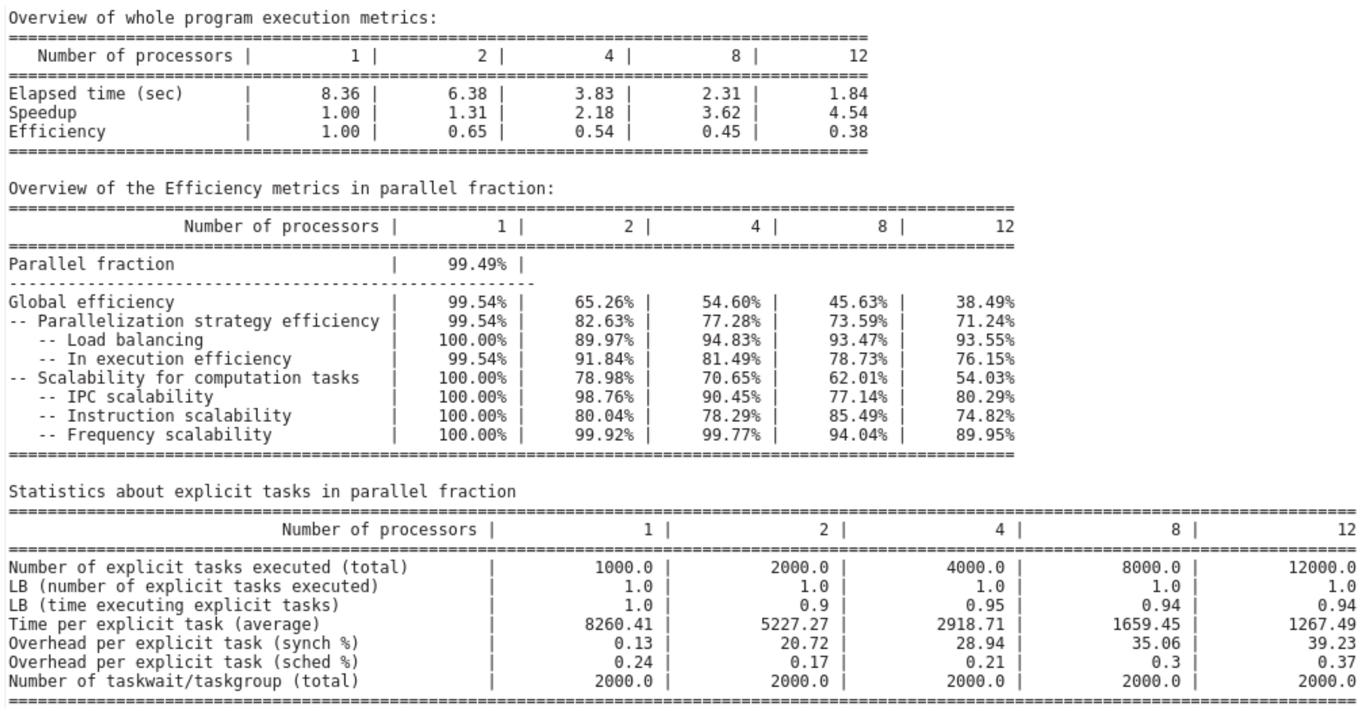


Figure 43: Scalability plots for Gauss-Seidel explicit tasks



```
Overview of whole program execution metrics:
=====================================================================================
   Number of processors |          1 |          2 |          4 |          8 |         12
=====================================================================================
Elapsed time (sec)      |       8.36 |       6.38 |       3.83 |       2.31 |       1.84
Speedup                 |       1.00 |       1.31 |       2.18 |       3.62 |       4.54
Efficiency              |       1.00 |       0.65 |       0.54 |       0.45 |       0.38
=====================================================================================

Overview of the Efficiency metrics in parallel fraction:
====================================================================================================
                 Number of processors |          1 |          2 |          4 |          8 |         12
====================================================================================================
Parallel fraction                     |     99.49% |
------------------------------------------------------
Global efficiency                     |     99.54% |     65.26% |     54.60% |     45.63% |     38.49%
-- Parallelization strategy efficiency |     99.54% |     82.63% |     77.28% |     73.59% |     71.24%
   -- Load balancing                  |    100.00% |     89.97% |     94.83% |     93.47% |     93.55%
   -- In execution efficiency         |     99.54% |     91.84% |     81.49% |     78.73% |     76.15%
-- Scalability for computation tasks  |    100.00% |     78.98% |     70.65% |     62.01% |     54.03%
   -- IPC scalability                 |    100.00% |     98.76% |     90.45% |     77.14% |     80.29%
   -- Instruction scalability         |    100.00% |     80.04% |     78.29% |     85.49% |     74.82%
   -- Frequency scalability           |    100.00% |     99.92% |     99.77% |     94.04% |     89.95%
====================================================================================================

Statistics about explicit tasks in parallel fraction
=====================================================================================================================================
                           Number of processors |                 1 |                 2 |                 4 |                 8 |                12
=====================================================================================================================================
Number of explicit tasks executed (total)       |            1000.0 |            2000.0 |            4000.0 |            8000.0 |           12000.0
LB (number of explicit tasks executed)          |               1.0 |               1.0 |               1.0 |               1.0 |               1.0
LB (time executing explicit tasks)              |               1.0 |               0.9 |              0.95 |              0.94 |              0.94
Time per explicit task (average)                |           8260.41 |           5227.27 |           2918.71 |           1659.45 |           1267.49
Overhead per explicit task (synch %)            |              0.13 |             20.72 |             28.94 |             35.06 |             39.23
Overhead per explicit task (sched %)            |              0.24 |              0.17 |              0.21 |               0.3 |              0.37
Number of taskwait/taskgroup (total)            |            2000.0 |            2000.0 |            2000.0 |            2000.0 |            2000.0
=====================================================================================================================================
```

Figure 44: Output of modelfactors.py Gauss-Seidel explicit tasks

5 Conclusions

To conclude, extracting the results obtained through strong-scalability plots, modelfactors, Paraver and various graphs of Tareador, we can close the laboratory assignment concluding that implicit tasks are useful for parallelizing these methods, and it is quite curious how changes the scalability plots depending on the number of dependences.

Furthermore, we could see in Jacobi that a parallelization with implicit tasks is better than using explicit tasks. Also, we have been able to learn the importance of the dependencies and how to treat them in a correct way. Moreover, we believe that we have acknowledged how to use some very useful tools such as OpenMP, Paraver and Tareador.

6 Final Survey

From 0 to 10, how would you rate:

- modelfactors: **9**
- Tareador: **9**
- Extrae + Paraver: **8,5**

`Modelfactors` has been very useful to understand the performance of our parallel applications. With this, we have had the opportunity to have all the relevant information about a parallel program, all together. We strongly advice its use in the laboratory assignments of this course in future editions.

With `Extrae` and `Paraver` we encountered problems with them sometimes related to delays in action requests, making it a bit awkward to work with these tools.

In general terms, we are very happy with the work carried out and with the organization of the laboratory by the teaching staff. It is noted that the laboratory scripts have been thought out and have gone through a refinement process before reaching the student, which is appreciated.

Finally, to close the last deliverable, we would like to thank Josep for his dedication to the subject and to the students: from the dedication shown in class, the shared values and passion for teaching, to the quick responses to emails even being very late. Learning from teachers like this is an honor and a privilege.

Please keep teaching like this.

See you around campus.