PAR Laboratory Assignment

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

Marc Duran Lopez: par1311
Jesús Pérez Bermejo: par1317
Spring 2022-23
May 19, 2023

INDEX

In this final laborator assignment we are going to work on the parallelization of a sequential code that simulates heat diffusion in a solid body using 2 different solvers for the heat equation (Jacobi and Guass-Seidel).

1.Sequential heat diffusion program and analysis with Tareador

1.1 Analysis of Jacobi solver

Let's begin by analyzing the simulation with the Jacobi solver. After examining the initial implementation with coarse-grained task definitions, we have obtained the following Task Dependency Graph (TDG) for the Jacobi solver:

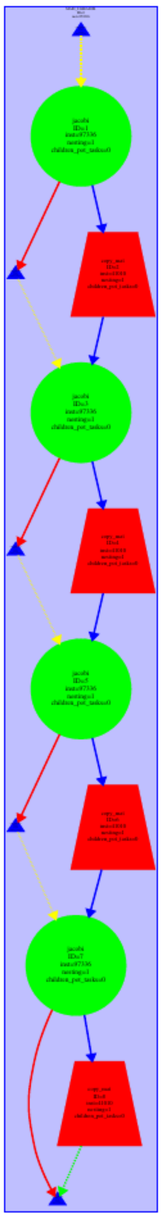


Fig. 1. TDG of the program with Jacobi solver and coarse-grain granularity

The green tasks represent the solve function, while the red tasks correspond to the copy_mat function. Upon inspecting the TDG, it becomes evident that there exist dependencies between the tasks, and at the current granularity level, there is no potential for parallelism. Therefore, it is necessary to explore finer granularities in order to exploit parallelism.
To achieve finer granularities, we made modifications to the solve function in the program by incorporating task creation per block. As a result, we achieved a granularity of one task per block.

```
// 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("jacobi_block");
        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("jacobi_block");
      }
    }
        stareador_enable_object(&sum);

    return sum;
}
```

Fig. 2. Function solve of the solver-tareador.c

After modifying the code, we have generated the new TDG. In this updated TDG, we observe an increase in the number of tasks created. However, we have identified a variable, namely "sum," that is causing the serialization of tasks. Upon further investigation of the code, we have determined that the variable "sum" is the root cause of this issue.

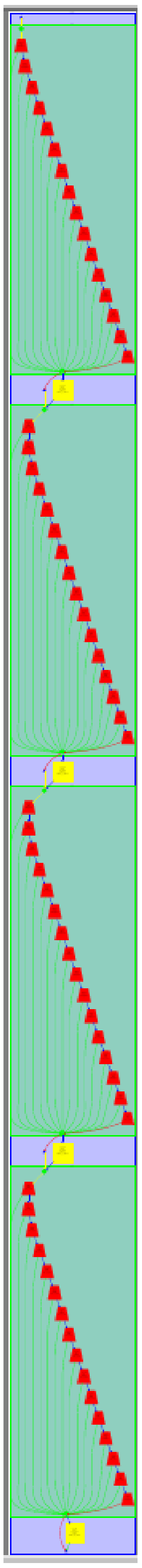


Fig. 3. TDG of the program with Jacobi solver and finer-grain granularity

To emulate the effect of protecting the dependencies caused by the variable, you have utilized the tareador_disable_object and tareador_enable_object calls. By uncommenting these calls, recompiling, and executing the code again, you obtained a new TDG. The updated TDG indicates that more parallelism has been achieved. In your OpenMP implementations, you can protect the variable by incorporating a reduction clause in the #pragma omp parallel construct.

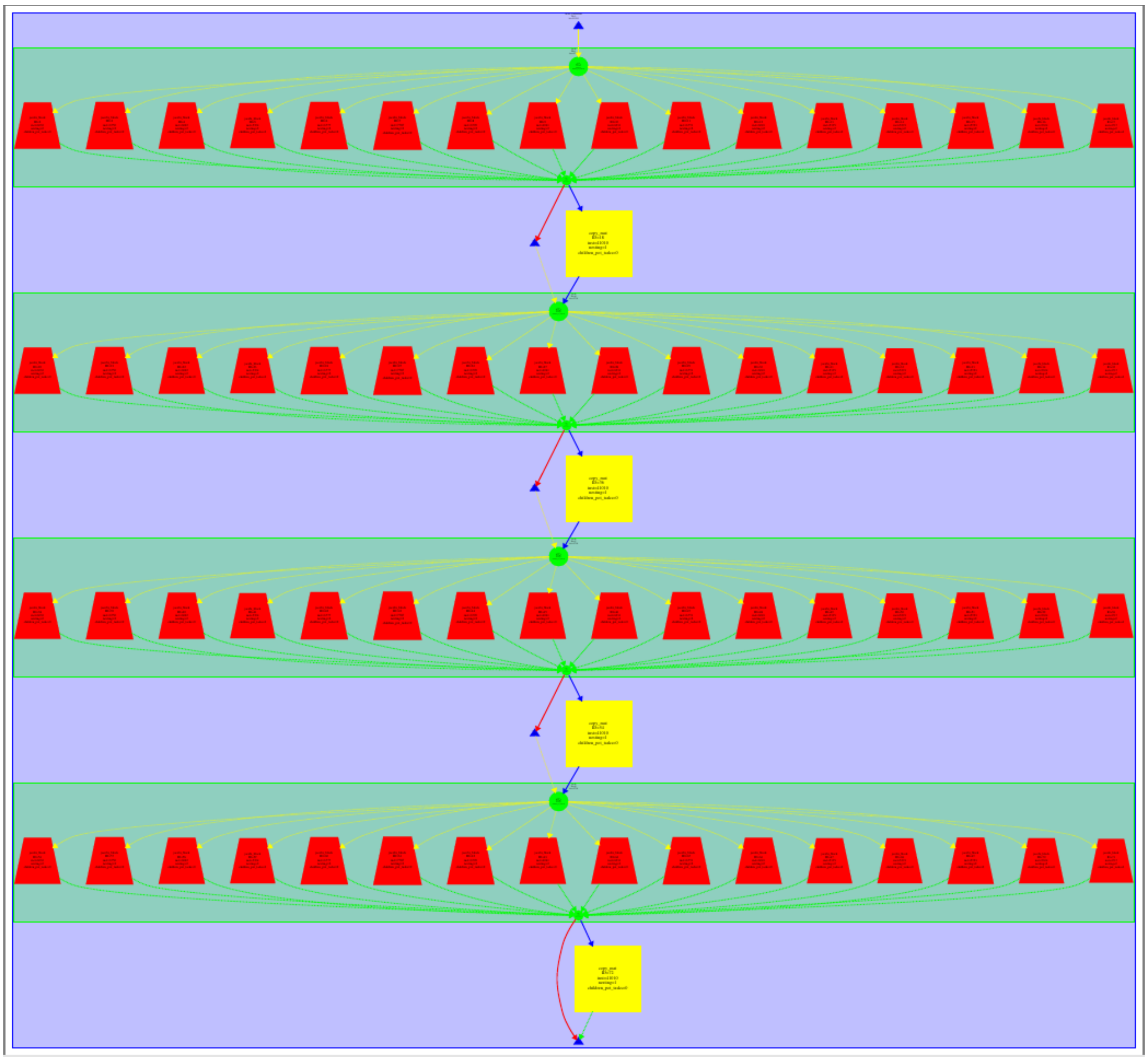


Fig. 4. TDG of the program with Jacobi solver and finer-grain granularity

Finally, we simulated the execution using 4 processors:

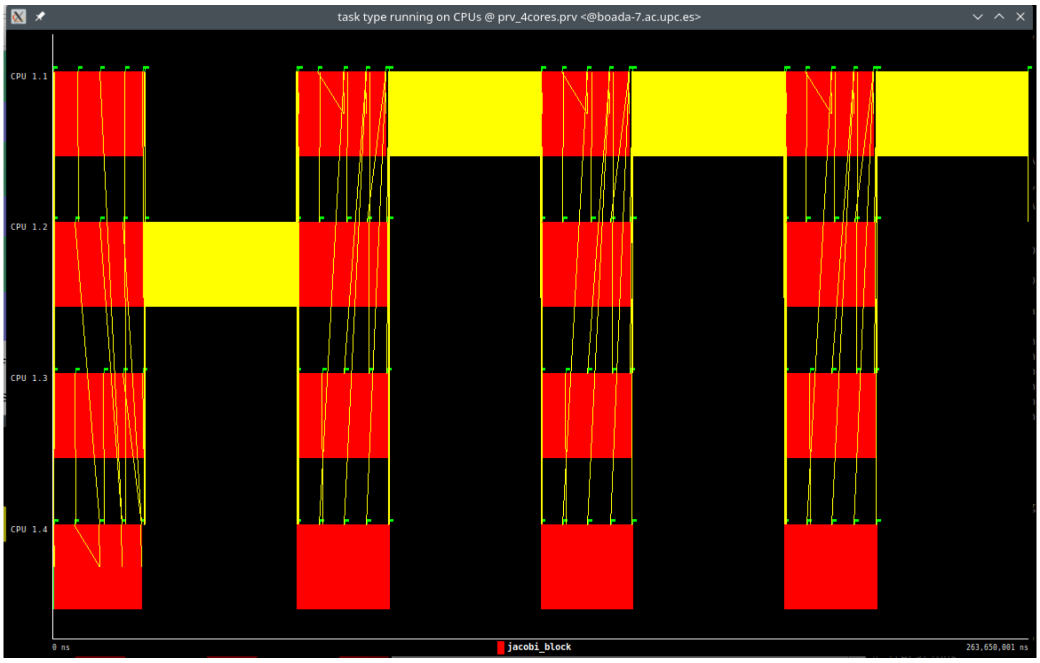

task type running on CPUs @ prv_4cores.prv <@boada-7.ac.upc.es>
CPU 1.1
CPU 1.2
CPU 1.3
CPU 1.4
0 ns
jacobi_block
263,650,001 ns

Fig. 5. Simulation of the program with 4 threads

The red regions in the simulation correspond to the parallel region of the solve function, indicating successful parallelization. However, you have identified a yellow region representing the copy_mat function, and maybe this can also be parallelized.

```
// 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("copy_mat");
        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("copy_mat");
              }
    }
}
```

Fig. 6. function copy_mat of the solver-tareador.c

After modifying the code to parallelize the copy_mat function, we have obtained a new TDG. In this updated TDG, we can observe that more tasks have been created, indicating that we have achieved a higher level of parallelism.

This is a positive outcome as it suggests that the modifications we have made to parallelize the copy_mat function have effectively increased the potential for parallel execution in our program.

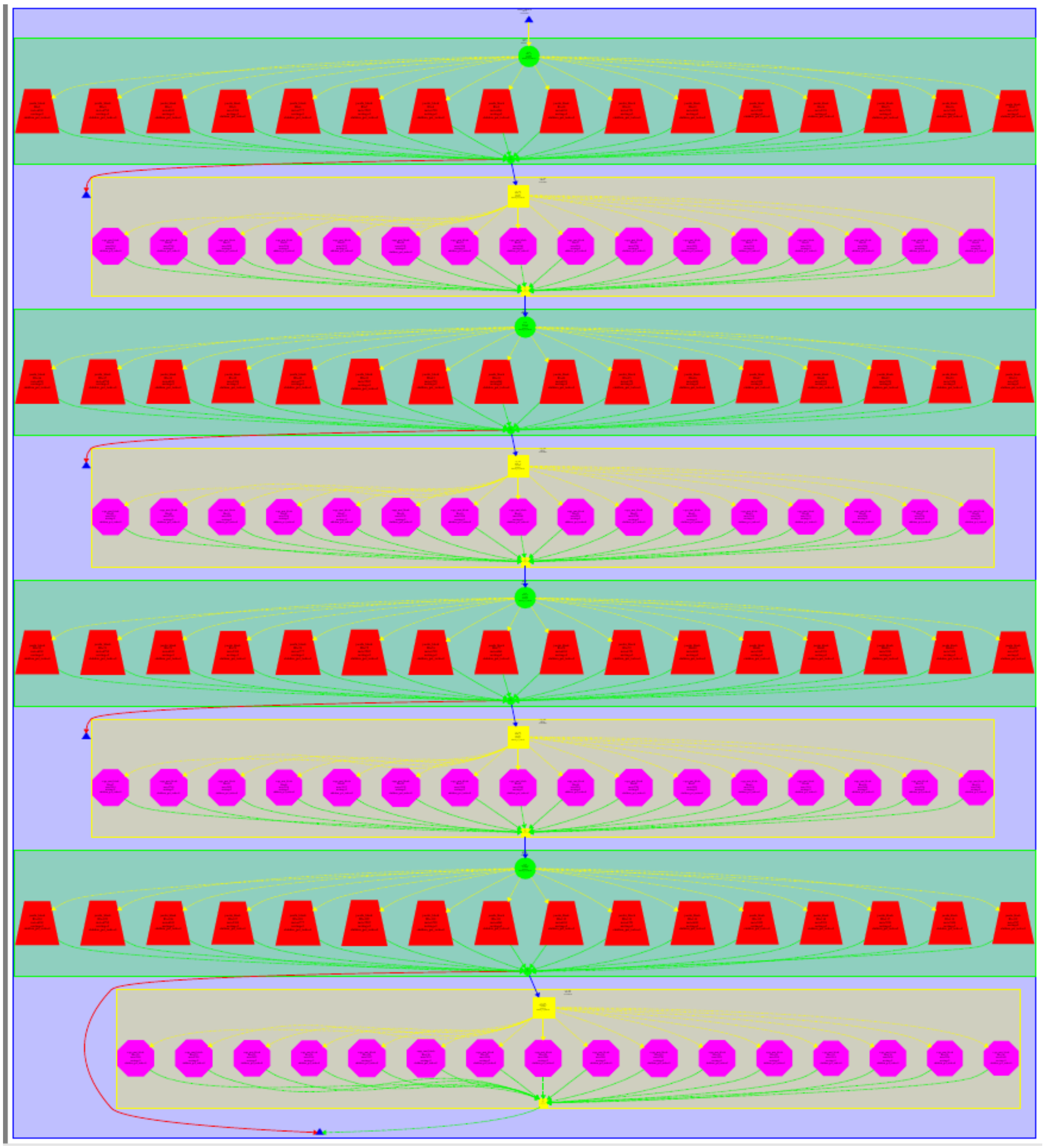


Fig. 7. TDG of the program with Jacobi solver and finer-grain granularity for both functions

Finally, we simulated the execution again with 4 processors, and we can now observe that all regions of the program are well-parallelized. This indicates that the modifications made to parallelize both the solve and copy_mat functions have been successful.

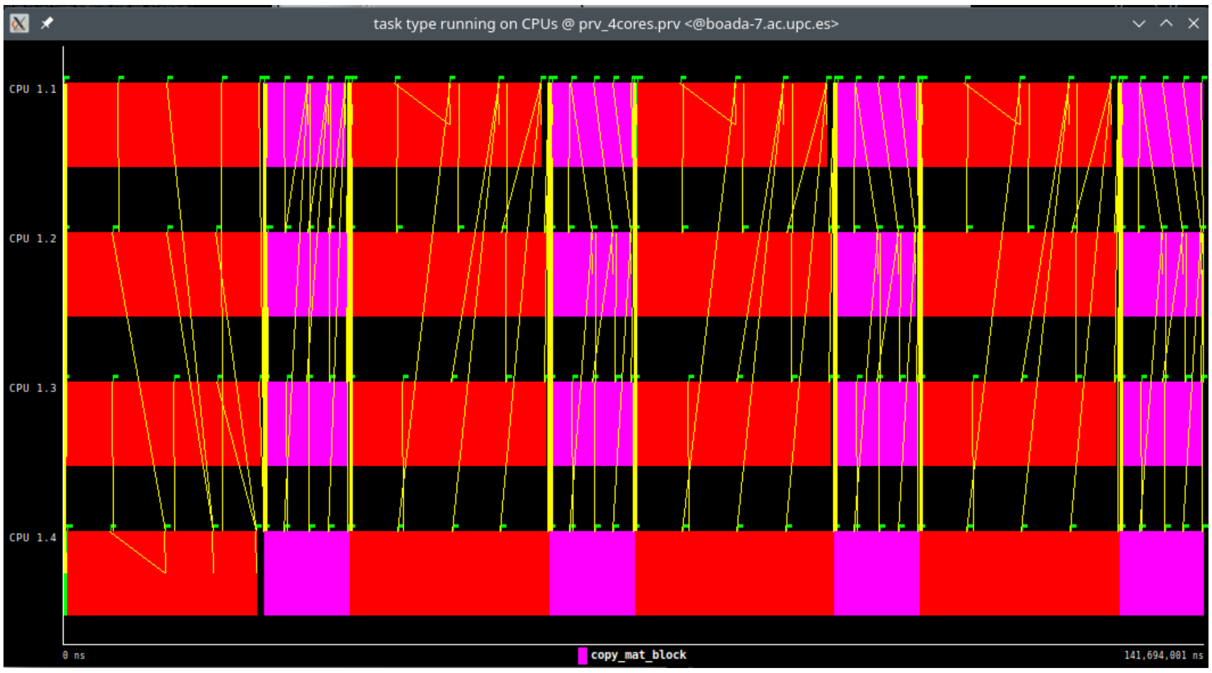

task type running on CPUs @ prv_4cores.prv <@boada-7.ac.upc.es>
CPU 1.1
CPU 1.2
CPU 1.3
CPU 1.4
0 ns
copy_mat_block
141,694,001 ns

Fig. 8. Simulation of the program with 4 threads

1.2 Analysis of Gauss solver

After analyzing the JacobI solver, run the same simulation, Gauss-Seidel solver.
From the task dependency diagram obtained with the Tareador task defined by default:
Again, parallelism is clearly not possible, as shown in Figure 9. This level of granularity can be exploited. When all tasks are executed, all tasks are executed in order. Use the same data. In contrast to the Jacobi solver, the Gauss-Seidel solver saves the results. Since the data are computed on the same matrix, no matrix copy task is performed.

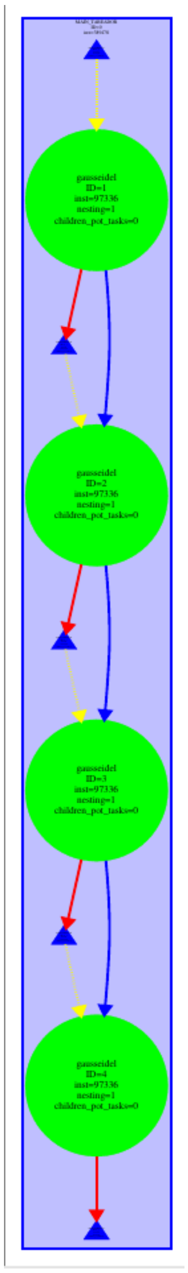


Fig. 9. TDG of the program with Guass-Seidel solver and coarse-grain granularity

Then try a finer level of granularity. It also consists of one task per block. Here is the dependency graph obtained:

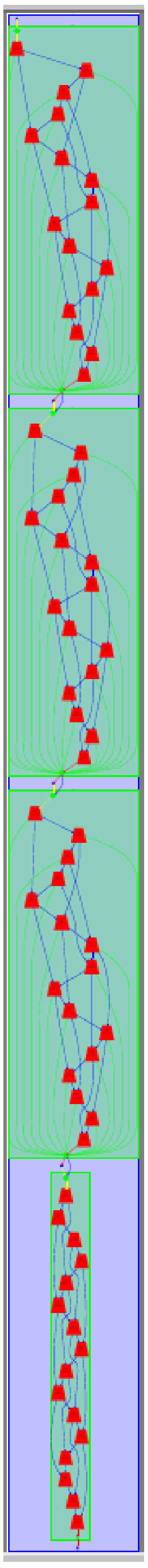


Fig. 10. TDG of the program with Guass-Seidel solver and finer-grain granularity

Although the dependencies between tasks are different from those created in Jacobi solver, the tasks are always executed sequentially, free of the sum variable. Confirm this, we reused the tareador_disable_object and tareador_enable_object calls to make Tareador ignore this variable.
The new graph is shown in Figure 11. This time better parallelization is achieved, but the dependency between some blocks is still generated due to access and store data of the same matrix. These dependencies are visually illustrated in Figure 12.

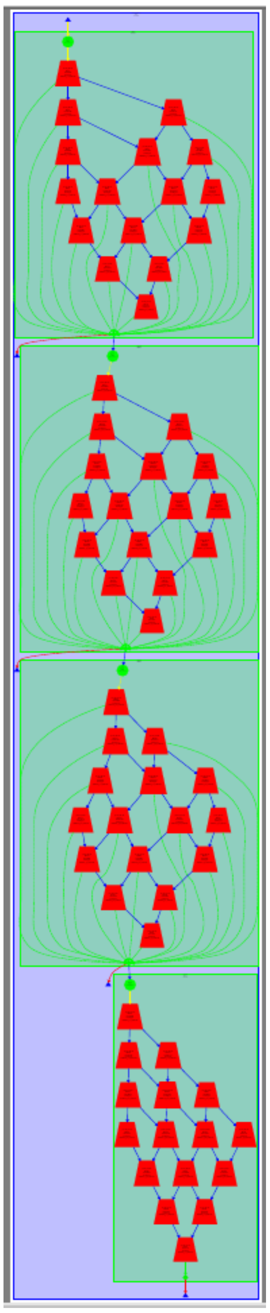


Fig. 11. TDG of the program with Guass-Seidel solver and finer-grain granularity

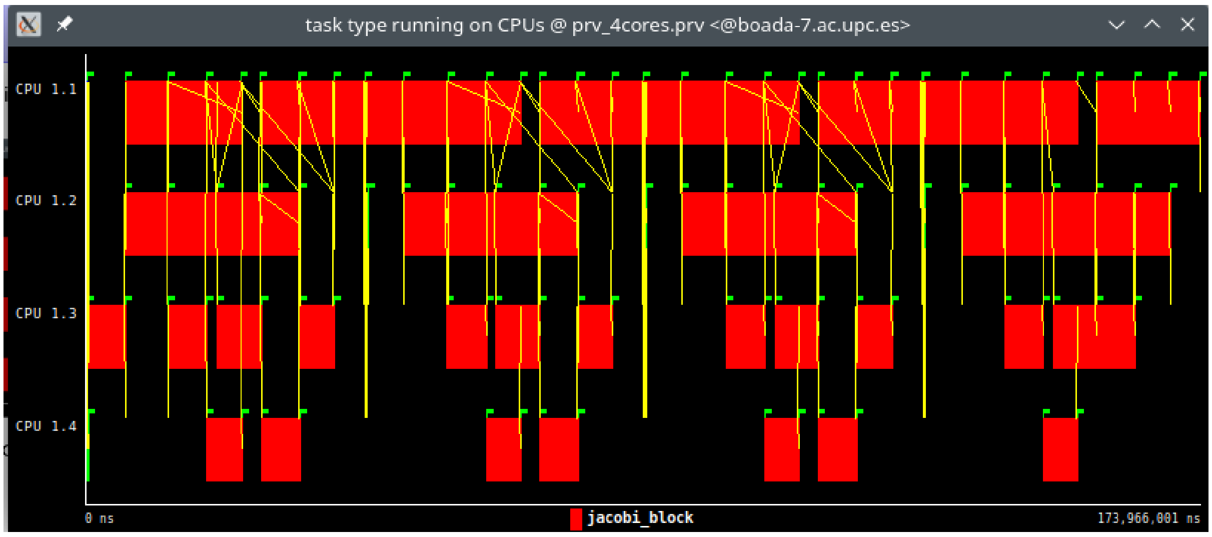

task type running on CPUs @ prv_4cores.prv <@boada-7.ac.upc.es>
CPU 1.1
CPU 1.2
CPU 1.3
CPU 1.4
0 ns
jacobi_block
173,966,001 ns

Fig. 12. Simulation of the program with 4 threads

2.Parallelisation of the heat equation solvers

In this second part, we will apply what we analyzed in the previous section using the Tareador tool. Parallelize the code using OpenMP clauses and adapt it to each of the two solvers:
Jacobi and Gauss-Seidel.

2.1 Jacobi solver

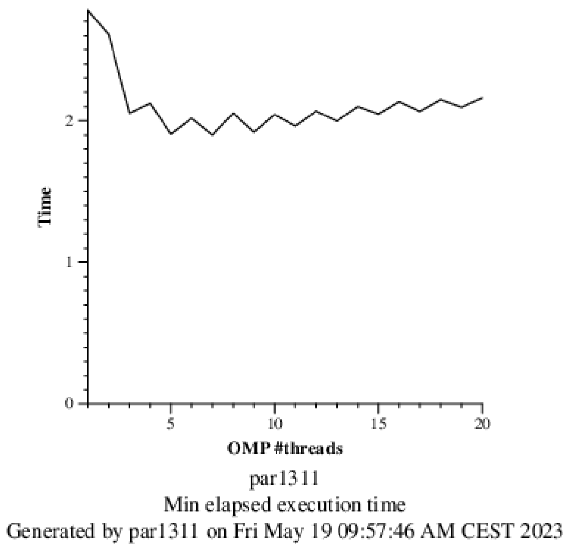
In this section, we parallelize the sequential code of the heat equation code, considering the use of the Jacobian solver and the implicit tasks generated in parallel with #pragma omp. Parallelization with the Jacobi solver was implemented as follows.

```
// 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;
}
```

Fig. 13. function solver of the solver-omp.c

Once the code was complete, I used a makefile to compile it and submitted its execution to a queue using the submit-omp.sh script. (sbatchsubmit-omp.sh heat-omp 0 8)

In the next step, after verifying (using diff) that the image produced by the modified code does not differ from the original sequential version, the submit-strong-omp.sh script is used to I got a chart (Fig. 14).



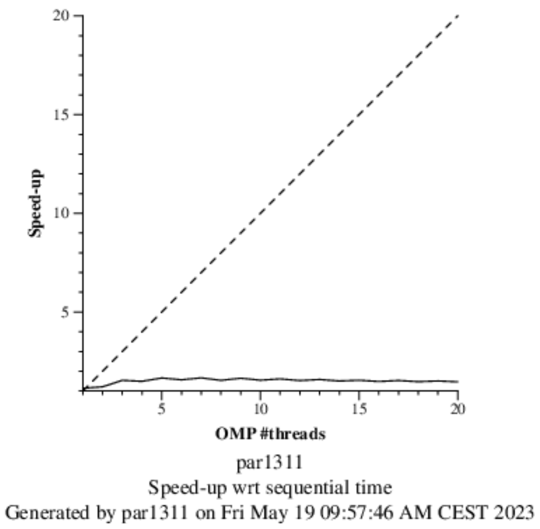


Fig. 14. Execution Time plot (left) and Speed-up plot (right)

As can be seen from the scalability graph, the parallelism obtained is not good. Increasing the number of threads does not improve speed. You can see that the acceleration remains fairly constant at 1.5.

To understand what's going on, I took a trace of the program's execution timeline (Figure 15).

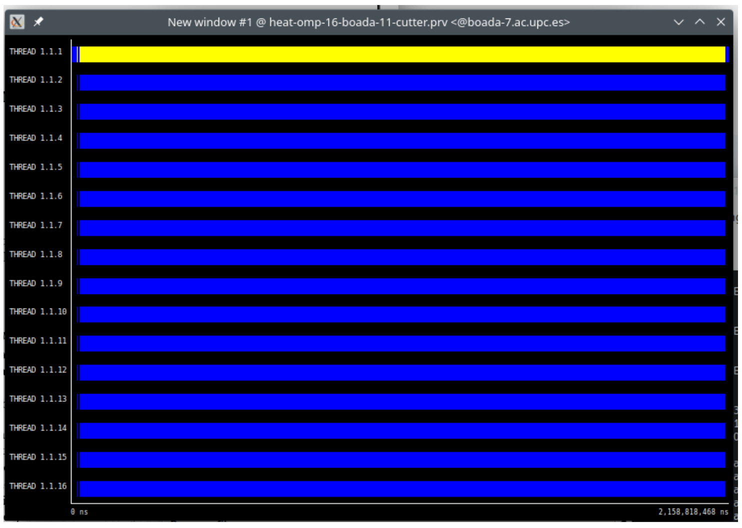


Fig. 15. Trace of the new window

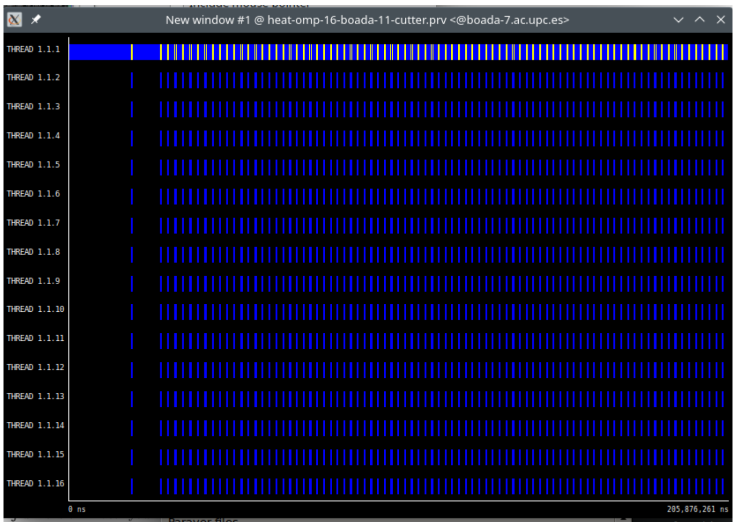
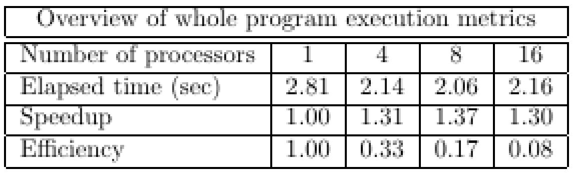


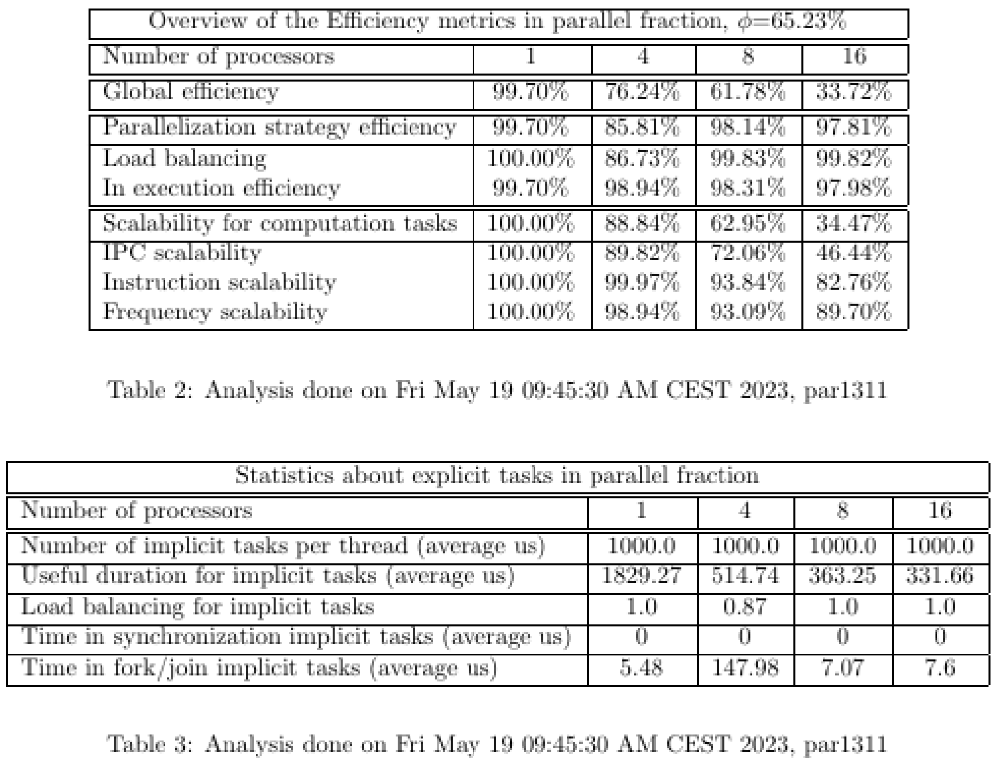
Fig. 16. Zoomed trace of the new window

Analysis of the two traces reveals that the parallel regions are only a small part of the overall program (Figure 15), and that the parallel regions do not contain all the pieces.
If it is parallelized (Figure 16), the blue part indicates that the thread is running, or has regions that is not parallelized.



Overview of whole program execution metrics				
Number of processors	1	4	8	16
Elapsed time (sec)	2.81	2.14	2.06	2.16
Speedup	1.00	1.31	1.37	1.30
Efficiency	1.00	0.33	0.17	0.08

Table 1: Analysis done on Fri May 19 09:45:30 AM CEST 2023, par1311



Overview of the Efficiency metrics in parallel fraction, ϕ=65.23%				
Number of processors	1	4	8	16
Global efficiency	99.70%	76.24%	61.78%	33.72%
Parallelization strategy efficiency	99.70%	85.81%	98.14%	97.81%
Load balancing	100.00%	86.73%	99.83%	99.82%
In execution efficiency	99.70%	98.94%	98.31%	97.98%
Scalability for computation tasks	100.00%	88.84%	62.95%	34.47%
IPC scalability	100.00%	89.82%	72.06%	46.44%
Instruction scalability	100.00%	99.97%	93.84%	82.76%
Frequency scalability	100.00%	98.94%	93.09%	89.70%

Table 2: Analysis done on Fri May 19 09:45:30 AM CEST 2023, par1311

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)	1829.27	514.74	363.25	331.66
Load balancing for implicit tasks	1.0	0.87	1.0	1.0
Time in synchronization implicit tasks (average us)	0	0	0	0
Time in fork/join implicit tasks (average us)	5.48	147.98	7.07	7.6

Table 3: Analysis done on Fri May 19 09:45:30 AM CEST 2023, par1311

Fig. 17. Modelfactor tables of Jacobi solver

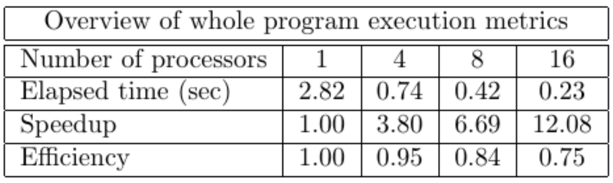
As can be seen from the tables, the speed increases with the number of threads, the opposite happens with the overall efficiency dropping sharply as more threads are added.
We observe in the second and third tables that it is only parallelized the 64.63% of the code and that the scalability for the computation tasks highly decreases.

To improve the efficiency of parallel code, use the copy_mat function as shown in the following figure:

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

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

    #pragma omp parallel
    {
            int blocki = omp_get_thread_num();
      int i_start = lowerb(blocki, nblocksi, sizex);
      int i_end = upperb(blocki, nblocksi, sizex);
      for (int blockj=0; blockj<nblocksj; ++blockj) {
        int j_start = lowerb(blockj, nblocksj, sizey);
        int j_end = upperb(blockj, nblocksj, sizey);
        for (int i=max(1, i_start); i<=min(sizex-2, i_end); i++)
          for (int j=max(1, j_start); j<=min(sizey-2, j_end); j++)
            v[i*sizey+j] = u[i*sizey+j];
      }
    }
}
```

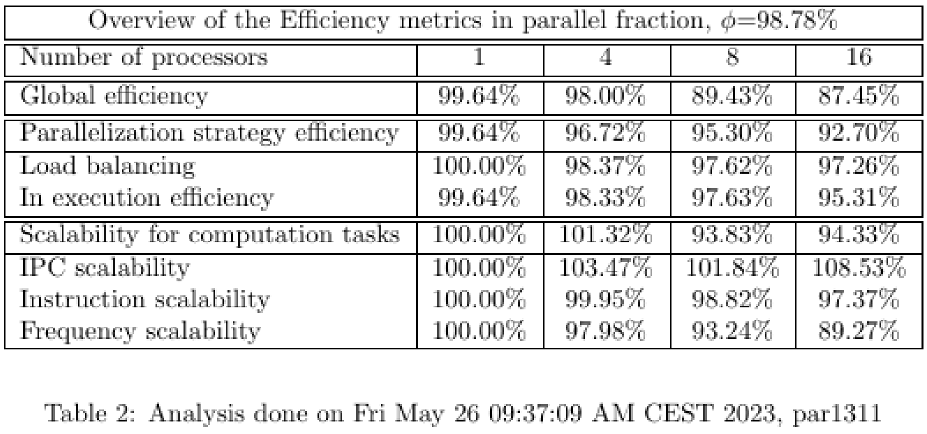
Fig. 18. function copy_mat of the solver-omp.c

Now we executed the modelfactor analysis to compare and analyze the new version of the code.



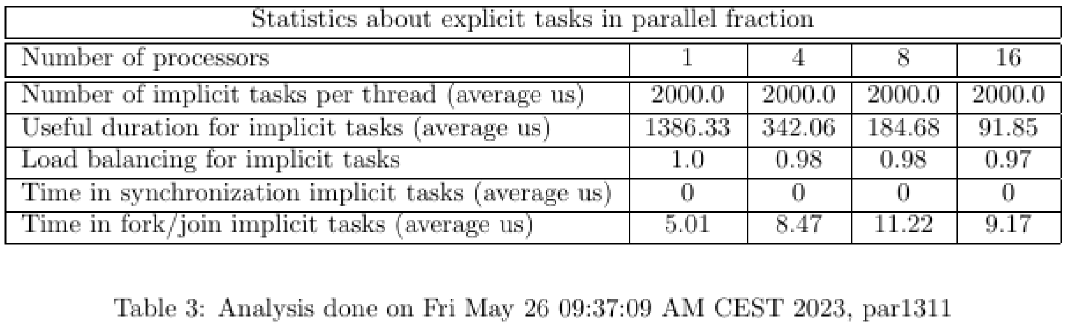
Overview of whole program execution metrics				
Number of processors	1	4	8	16
Elapsed time (sec)	2.82	0.74	0.42	0.23
Speedup	1.00	3.80	6.69	12.08
Efficiency	1.00	0.95	0.84	0.75

Table 1: Analysis done on Fri May 26 09:37:09 AM CEST 2023, par1311



Overview of the Efficiency metrics in parallel fraction, ϕ=98.78%				
Number of processors	1	4	8	16
Global efficiency	99.64%	98.00%	89.43%	87.45%
Parallelization strategy efficiency	99.64%	96.72%	95.30%	92.70%
Load balancing	100.00%	98.37%	97.62%	97.26%
In execution efficiency	99.64%	98.33%	97.63%	95.31%
Scalability for computation tasks	100.00%	101.32%	93.83%	94.33%
IPC scalability	100.00%	103.47%	101.84%	108.53%
Instruction scalability	100.00%	99.95%	98.82%	97.37%
Frequency scalability	100.00%	97.98%	93.24%	89.27%

Table 2: Analysis done on Fri May 26 09:37:09 AM CEST 2023, par1311



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)	1386.33	342.06	184.68	91.85
Load balancing for implicit tasks	1.0	0.98	0.98	0.97
Time in synchronization implicit tasks (average us)	0	0	0	0
Time in fork/join implicit tasks (average us)	5.01	8.47	11.22	9.17

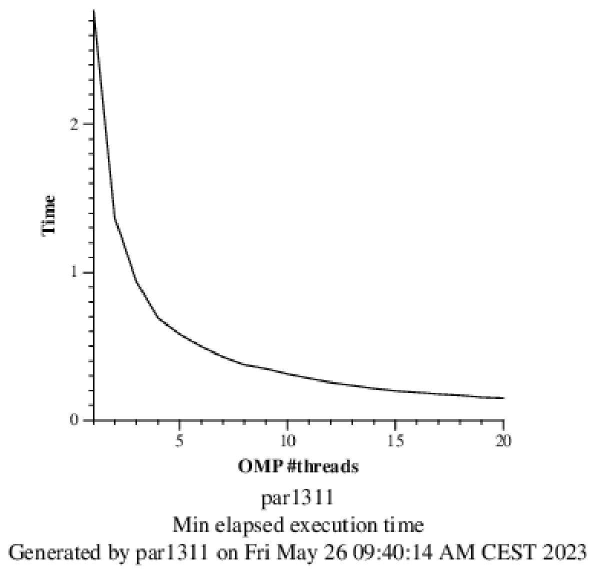
Table 3: Analysis done on Fri May 26 09:37:09 AM CEST 2023, par1311

Fig. 19. Modelfactor tables of the copy_mat version

As it can be seen in the first table, the execution time is significantly reduced in this version as more threads are added, resulting in better overall performance compared to the previous version. Likewise, the speed-up metric shows more substantial improvements in this version as more threads are incorporated. Although there is a slight reduction in efficiency, it is still superior to the version where the "copy_mat" operation is not parallelized.

Upon examining the other tables, it becomes apparent that the parallelized section of the code has experienced significant improvements. This increase in parallelization explains why this version exhibits better performance in parallel execution compared to the previous version.

When examining the scalability values and corresponding plots, it becomes evident that the new version exhibits significantly better scalability compared to the previous version. The new version approaches a nearly perfect scalability line, as demonstrated in the plot below.
This indicates that as the number of resources (threads or processors) increases, the performance of the new version scales almost linearly, resulting in more efficient utilization of available resources.



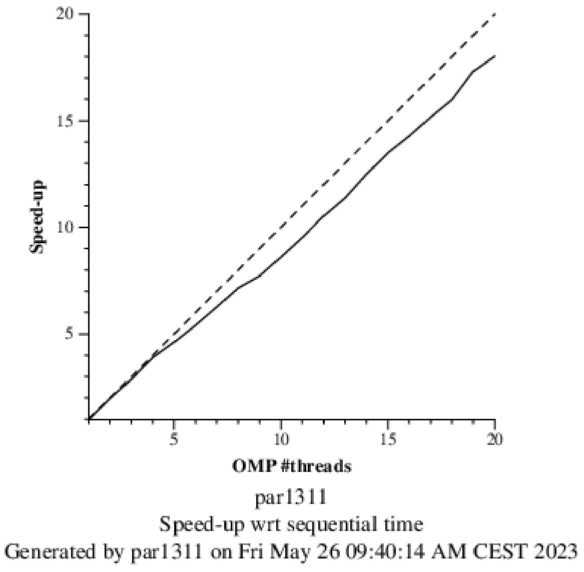


Fig. 20. Execution Time plot (left) and Speed-up plot (right)

Finally we executed the paraver analysis obtaining the time trace of the Fig.19.

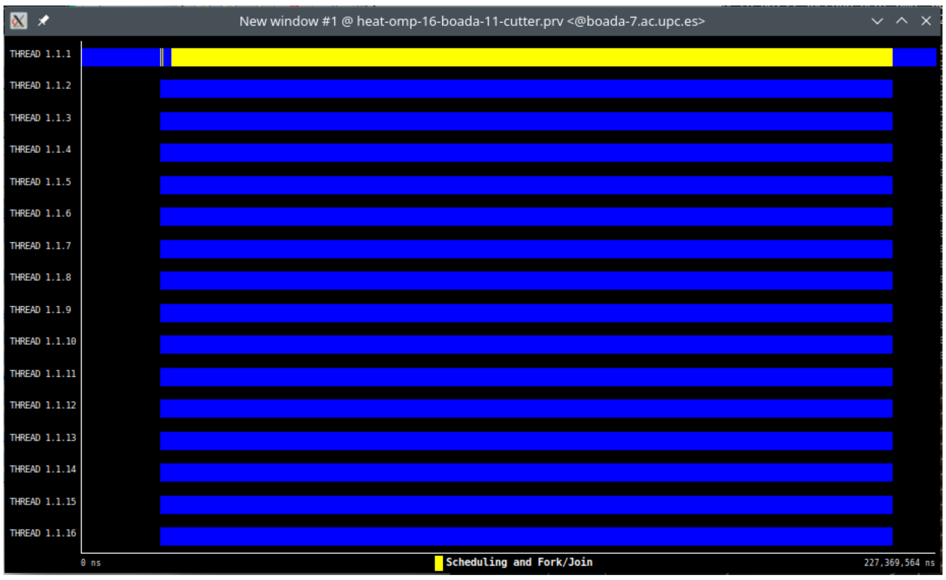


Fig. 21. Trace of the new window

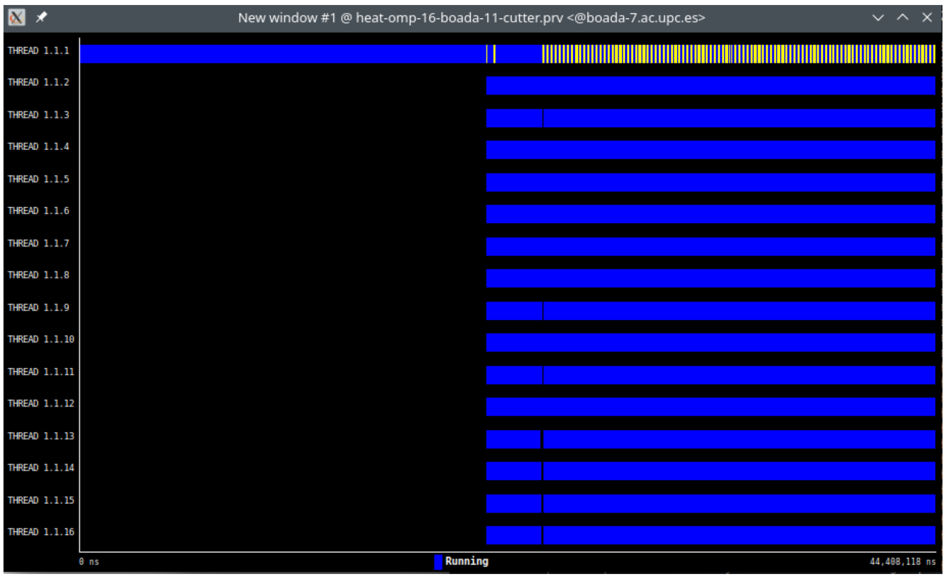


Fig. 22. Zoomed trace of the new window

By comparing the traces of the two versions, it is noticeable that the trace of the new version exhibits fewer instances of "black holes" among the threads. This indicates that the waiting time for one of the threads is reduced as a result of parallelizing the "copy_mat" function. Consequently, the new version demonstrates improved performance and enhanced parallelization compared to the previous version.

Additionally, even though the execution time of the "solve" operation remains the same, the overall execution time is improved in the new version. This improvement is attributed to the parallelization of the matrix copying process, which leads to a more efficient total execution time.

2.2 Gauss-Seidel solver

Now, we will attempt to parallelize this code while considering the dependencies of the Gauss-Seidel approach. Our parallelization strategy will involve a block-by-row data decomposition, utilizing implicit tasks within parallel regions.

Additionally, we have taken into consideration the annex of the lab to address any memory consistency issues. As a result, we have modified the code accordingly, and the updated version can be seen below.

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

        int n_threads = omp_get_max_threads();

        int nblocksi=n_threads;
        int nblocksj=4;

    int next[n_threads];
    for (int i = 0; i < n_threads; ++i) next[i] = 0;

        #pragma omp parallel firstprivate(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);

        if (u == unew && blockj > 0) {
        int flag;
        do {
                #pragma omp atomic read
                flag = next[blockj -1];
                 }
                while (flag < blocki);
        }

        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
                next[blockj]++;
         }
        }
        }

        return sum;
}
```

Fig. 23. function solve of the solver-omp.c

We can observe that the main issue with this parallelization approach is its scalability. As we increase the number of threads, the global efficiency decreases.

The subsequent plots (Fig. 24) visually depict this conclusion, showing that the speedup achieved is far from optimal, as it deviates from the ideal speedup line.
This visual representation emphasizes that the observed speedup falls short of the expected and desired levels, further reinforcing the conclusion that the parallelization approach has limitations in terms of achieving optimal performance gains.

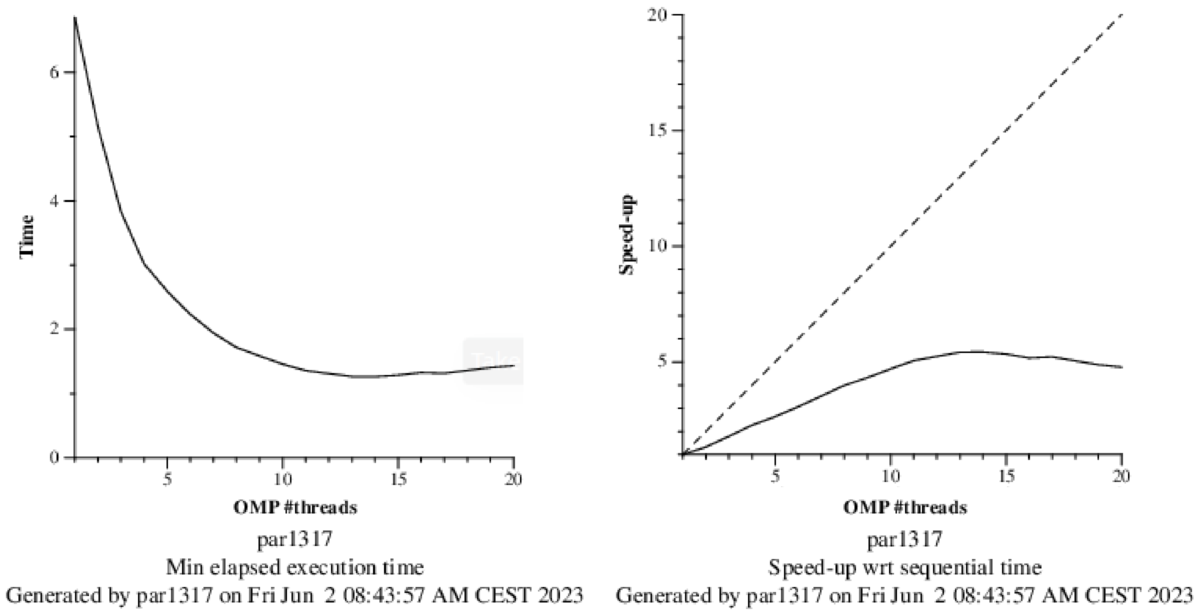


Fig. 24. Execution Time plot (left) and Speed-up plot (right)

We modified the code to make use of the value “user_param” in nblocksj, so we can choose the number of blocks in the j dimension.

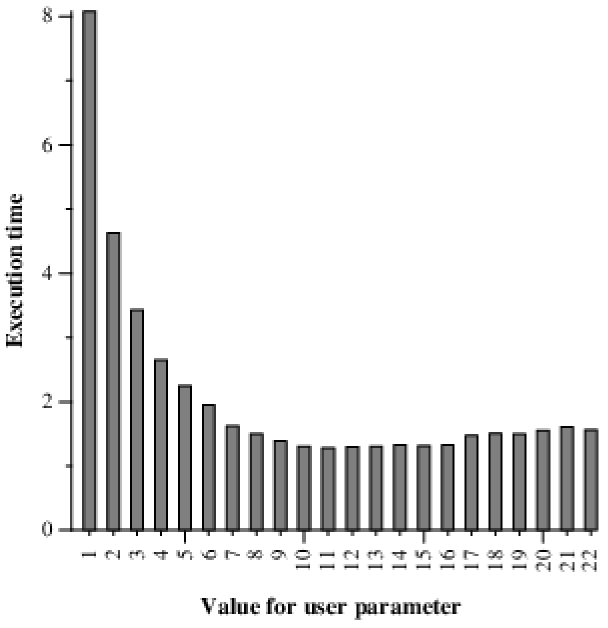


Fig. 25. Execution time plot with 16 threads

Here we can see the plot of the different execution times with the different values of userparam in nblocksj and 16 threads. Executing ./submit-userparam-omp.sh we can see in Fig.25. that the execution time decreases as the value used as parameter is higher. This is because the parameter value increases the number of blocksj, creating more loops that can be parallelized. We can conclude that, with the information of this plot, the optimal value for 16 threads is 11, approximately.

Now, we take a look at the scalability with this modification in the code, and we are going to compare it with the previous scalability plot with nblocksj = 4

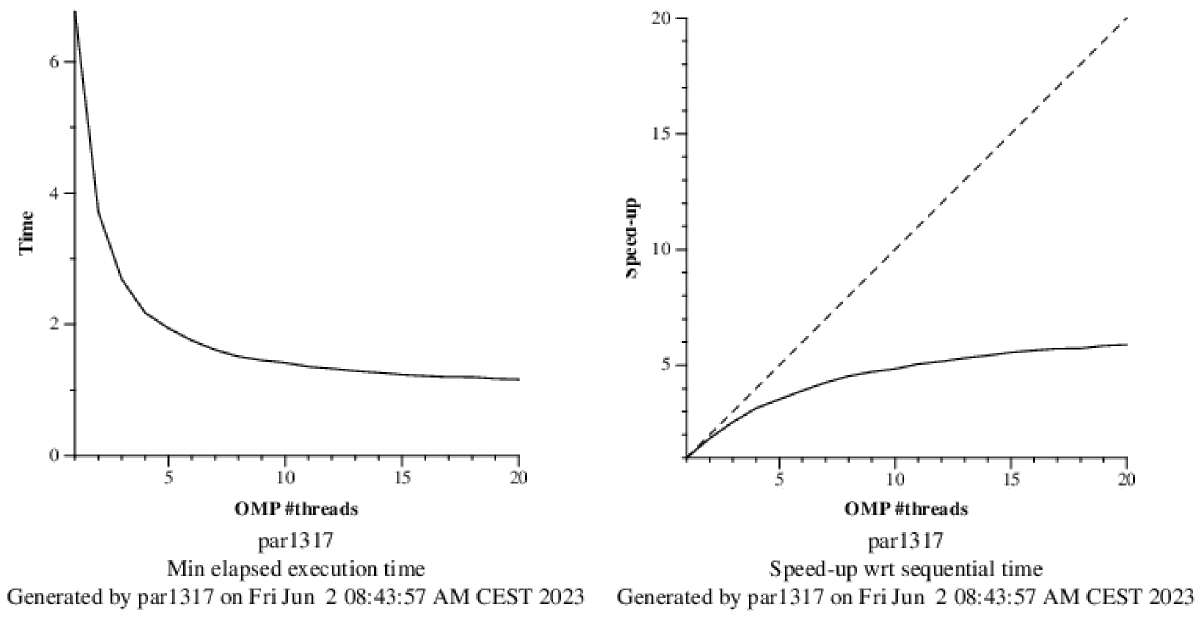


Fig. 26. Zoomed trace of the new window

In the plot of Fig. 26, we can appreciate that the speedup improves as more threads are used, compared to the previous plot without using the UserParams value.
In general, the code does not approach ideal scalability, but at least we have managed to improve it compared to the previous code.

3. Conclusions

During our recent laboratory assignment, we investigated the impact of data decomposition strategies by parallelizing the Jacobi and Gauss-Seidel solvers for the heat equation. We initially focused on the Jacobi solver and utilized the Tareador tool to analyse the potential for parallelizing the program. By examining the Task Dependency Graph (TDG), we identified dependencies among different variables and understood how task granularity can affect parallelization possibilities.

Initially, we parallelized the solve function of the Jacobi solver. However, upon observing the results through various plots, we realized that there were additional regions within the code that could also be parallelized. Consequently, we parallelized the copy_mat function as well. The results showed significant improvement, indicating the effectiveness of our approach.

In the second phase of our study, we modified the code to enable parallelization in the OpenOMP program. Similar to the first phase, we initially implemented parallelization in the solver function, but the performance was unsatisfactory. Therefore, we extended parallelism to the copy_mat function as well. As anticipated, this led to much better performance.

Regarding the Gauss-Seidel solver, we made an analysis of the dependencies that arose from storing the results in the same matrix used for computation. This analysis enabled us to parallelize the solver effectively by considering the variables responsible for these dependencies. We modified the code to change the number of nblocksj, to see how scalability was with this value. However, unlike the Jacobi solver, the scalability of the Gauss-Seidel solver after implementing parallelization was not as impressive.
Subsequently, we made further modifications to the code by replacing the value of nblocksj with userparam. This allowed us to investigate which value of nblocksj had better scalability in the program. Through this analysis, we observed a notable improvement in scalability as we increased the number of threads. And we saw that the best values for nblocksj were around 11 and 12.