Parallelism

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

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

Jia Long Ji Qiu: par2212
Jiabo Wang: par2220
Spring 2021-22
May 2022

Index

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.

Analysis of task granularities and dependencies

Analysis of Jacobi solver

First we will analyse the simulation with the Jacobi solver. After taking a look at the initial implementation with an coarse-grain task definition, we have obtained the next TDG for the jacobi solver (Figure 1):

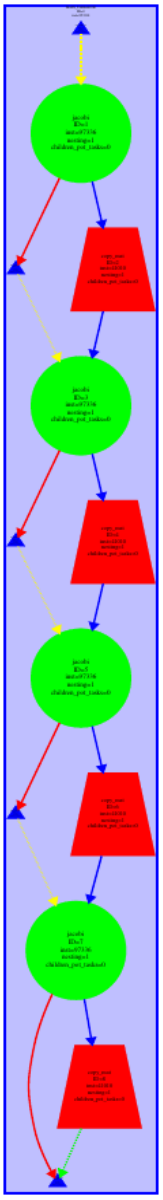

Figure 1: TDG of the program with Jacobi solver and coarse-grain granularity

The green tasks correspond to the solve function and the red ones to the copy_mat function. And by looking at the TDG, we can see that there are dependencies between the tasks and there is no parallelism that can be exploited at this granularity level, so we have to explore finer granularities.

In order to achieve finer granularities, we have modified the solve function of the program by adding the creation of tasks per block. The granularity we achieved is 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");
      }
    }
    //tareador_enable_object(&sum);

    return sum;
}
```

Figure 2: function solve of the solver-tareador.c

Once modified the code, we have obtained the TDG again (Figure 3). In this new TDG we can observe that more tasks are created, but there is a variable that is causing the serialisation of the tasks. By exploring the code, we have noticed that it is the variable sum that is causing the problem.

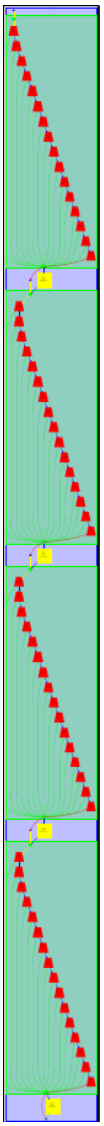

Figure 3: TDG of the program with Jacobi solver and finer-grain granularity

So 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. We have uncommented them and recompiled and executed them again. The TDG we have obtained is shown in the Figure below. We can see that more parallelism has been achieved and the way we can protect the variable in our OpenMP implementation is by putting a reduction in the #pragma omp parallel construct.

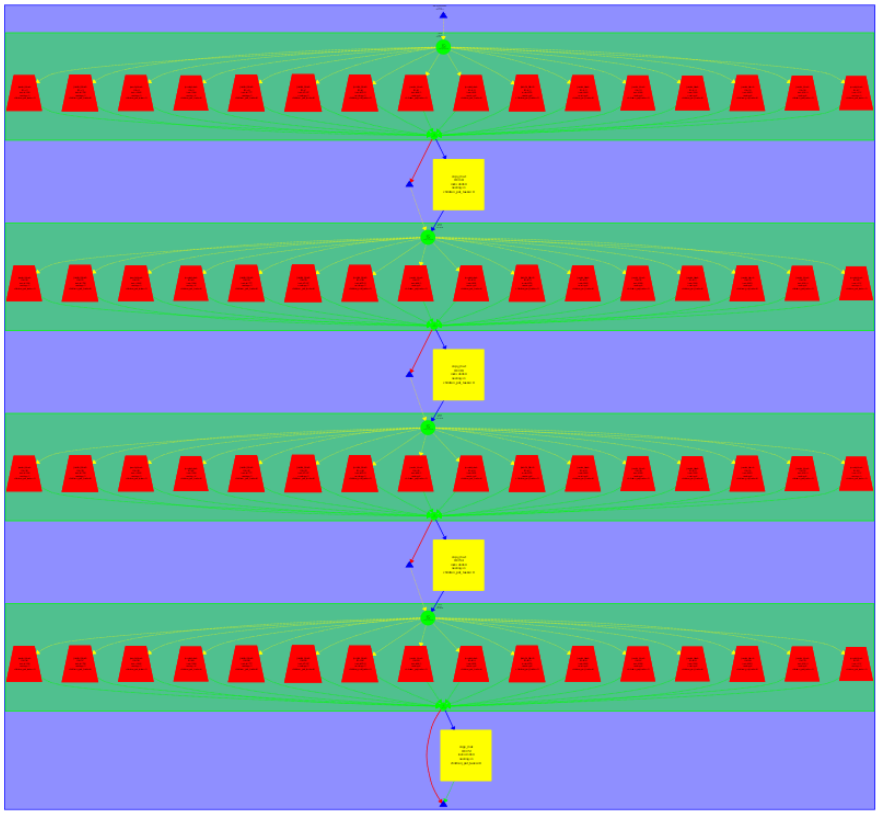

Figure 4: TDG of the program with Jacobi solver and finer-grain granularity

Finally we have simulated the execution using 4 processors (Figure 5).

Figure 5: Simulation of the program with 4 processors

The red regions correspond to the parallel region solve, and we can see that it is perfectly parallelised. However, there is also a yellow region that corresponds to the copy_mat function, and we think that it is also parallelizable.

In order to parallelize the copy_mat function, we have modified it by adding the tareador_start_task and tareador_end_task to the function as shown in the figure below: (Figure)

```
// 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_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++)
            v[i*sizey+j] = u[i*sizey+j];
        tareador_end_task("copy_mat_block");
       }
    }
}
```

Figure 6: function copy_mat of the solver-tareador.c

Once modified the code, we have obtained the TDG (Figure 7) again, and in this case we can see that there are more tasks created and the parallelism we achieved is bigger.

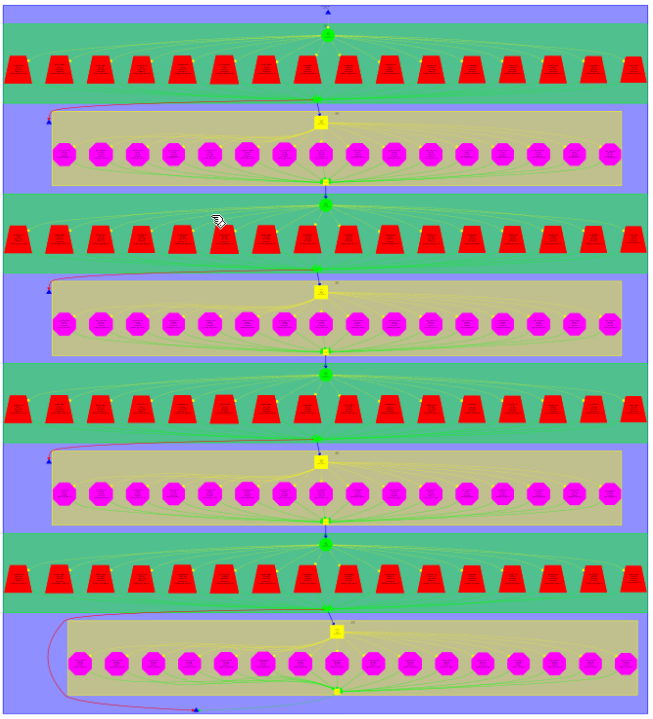

Figure 7: TDG of the program with Jacobi solver and finer-grain granularity for both functions

Finally, we simulated the execution again with 4 processors (Figure 8), and now we can see that all regions of the program are well-parallelised.

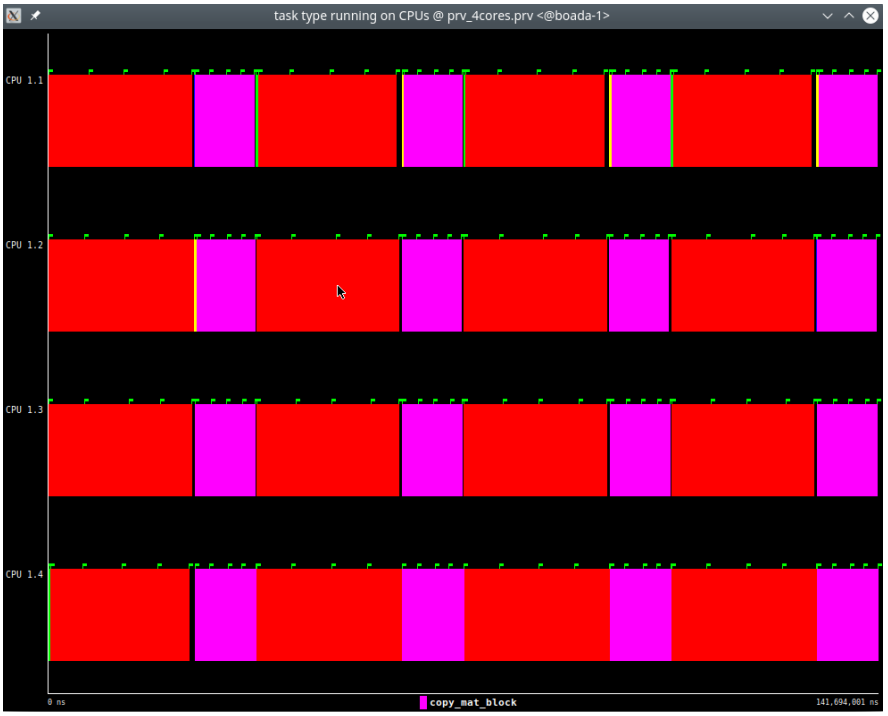

Figure 8: Simulation of the program with 4 processors

Analysis of Gauss-Seidel solver

After analysing the Jacobi solver, we will proceed now to do the same simulation for the Gauss-Seidel solver.

From the task dependency graph obtained with the Tareador tasks defined by default, as shown in figure 9, we can observe again that there is clearly no parallelism that can be exploited at this granularity level. All tasks are being executed sequentially since all tasks use the same data. Unlike the Jacobi solver, as the Gauss-Seidel solver saves the results computed in the same matrix, there are no matrix copying tasks being executed.

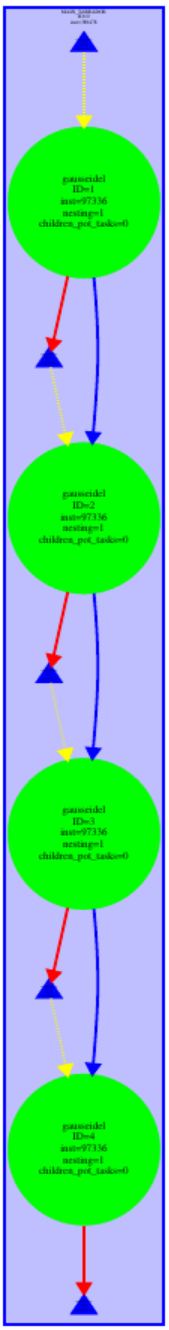

Figure 9: TDG of the program with Gauss-Seidel solver and coarse-grain granularity

Trying now with the finer granularity level, again consisting in one task per block, the task dependency graph obtained was the following:

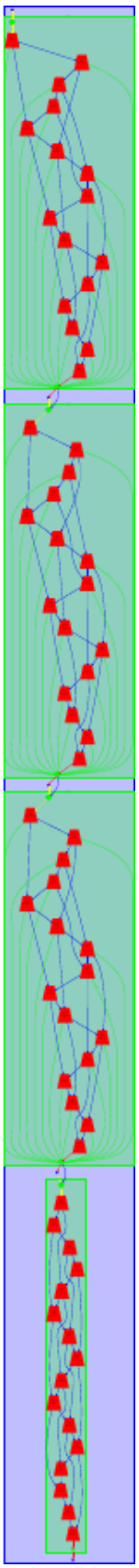

Figure 10: TDG of the program with Gauss-Seidel solver and finer-grain granularity

Although the dependencies between tasks are different from the ones generated in the Jacobi solver, tasks are still being executed sequentially due to the sum variable. To confirm this, we used once again 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 parallelisation is being achieved, but dependencies between some blocks are still being generated, caused by the access and storage of data from the same matrix. These dependencies are visually shown in figure 12.

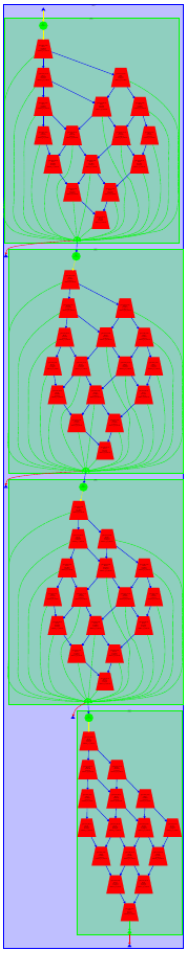

Figure 11: TDG of the program with Gauss-Seidel solver and finer-grain granularity

Figure 12: Simulation of the program with 4 processors

OpenMP parallelization and execution analysis: Jacobi

In this section we are going to parallelise the sequential code for the heat equation code considering the use of the Jacobi solver and using the implicit tasks generated in #pragma omp parallel.

The parallelisation with the Jacobi solver has been 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;
}
```

Figure 13: function solve of the solver-omp.c

After completing the code, we compiled it using the Makefile and submitted its execution to the queue using the submit-omp.sh script. (sbatch submit-omp.sh heat-omp 0 8)

The next step after validating that the image generated by the modified code does not have any difference with the original sequential version (using diff), we have obtained the next scalability plots (Figure 14) using the submit-strong-omp.sh script.

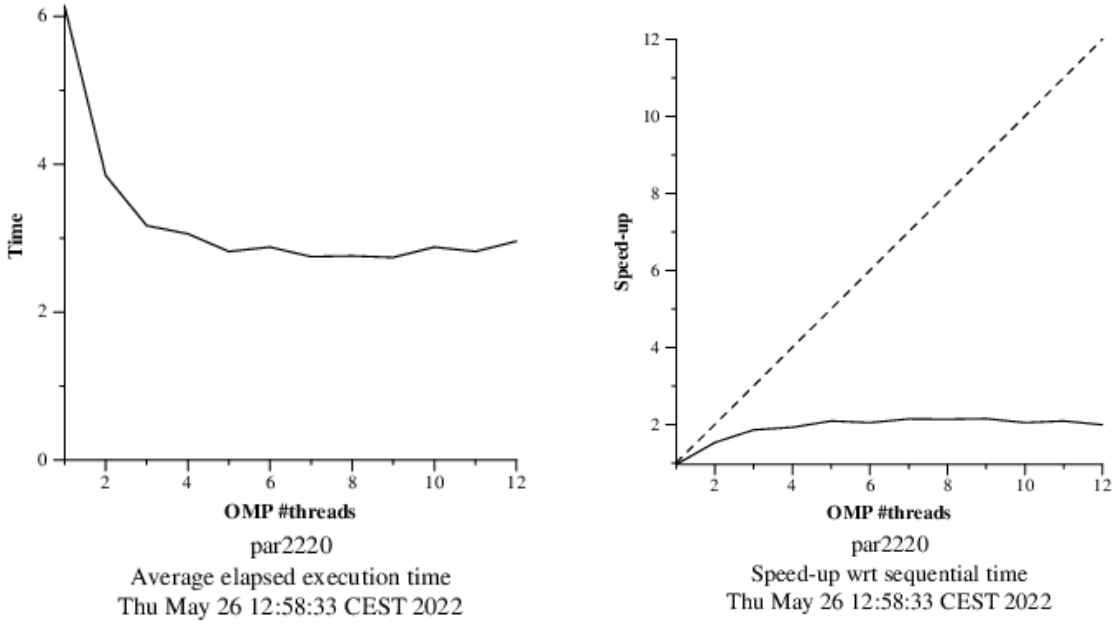

Figure 14: Execution Time plot (left) and Speed-up plot (right)

As we can observe in the scalability plots, the parallelisation we acquired is not appropriate; by increasing the number of threads, the speed-up does not increase. We can see that the speed-up remains almost constant at 2.

In order to understand what is happening, we got the traces of the execution timeline of the program (Figure 15).

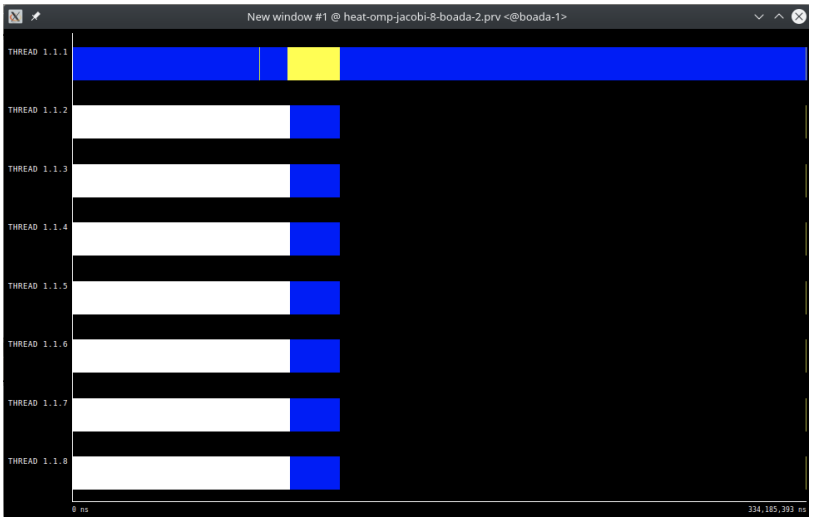

Figure 15: Trace of the new window

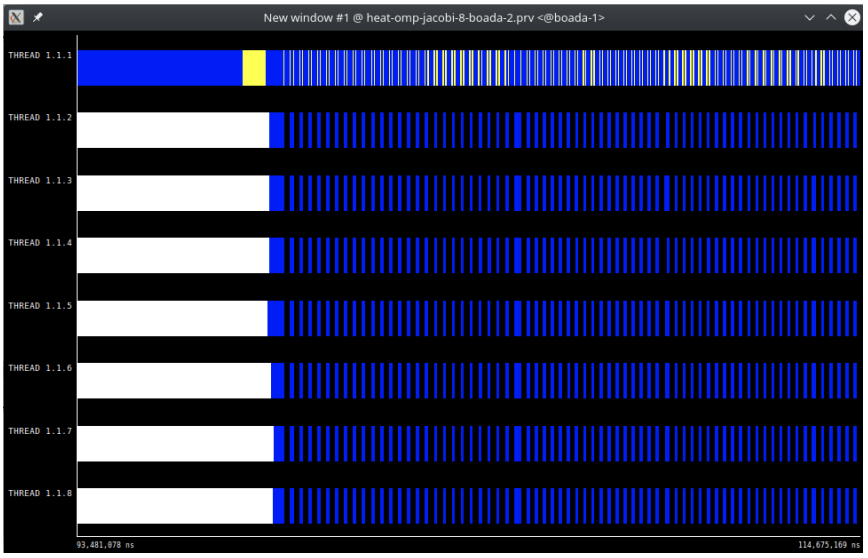

Figure 16: Zoomed trace of the new window

By analysing the two traces, we can see that first of all, the parallel region is just a small part of the whole program (Figure 15), and also in the parallel region, not all the parts are parallelised (Figure 16), the blue parts indicate that the thread is running, so it has regions that it is not parallelised.

So in order to improve the efficiency of our parallel code, we have parallelised also the copy_mat function, as shown in the next 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];
      }
    }
}
```

Figure 17: function copy_mat of the solver-omp.c

Again, after compiling the new version and submitting its execution to the queue using the submit-omp.sh script, we validated the results by comparing it with the sequential code and seeing no differences.

By comparing the characteristics of this version (Figure 18) with the initial version (Figure 19), we can see that the execution time is reduced.

```
Iterations        : 25000
Resolution        : 254
Residual          : 0.000050
Solver            : 0 (Jacobi)
Num. Heat sources : 2
   1: (0.00, 0.00) 1.00 2.50
   2: (0.50, 1.00) 1.00 2.50
Time: 3.063
Flops and Flops per second: (11.182 GFlop => 3650.57 MFlop/s)
Convergence to residual=0.000050: 15756 iterations
```

Figure 18: characteristics of the initial 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: 0.756
Flops and Flops per second: (11.182 GFlop => 14790.61 MFlop/s)
Convergence to residual=0.000050: 15756 iterations
```

Figure 19: characteristics of the improved version

Once validated, we have instrumented its parallel execution with Extrae in order to see the new parallel behaviour.

In this case, as the function copy_mat is parallelised, there has been a reduction of load imbalance. As shown in the next Figure 20, we can see that most of the time in the parallel region, all the threads are doing relevant computation.

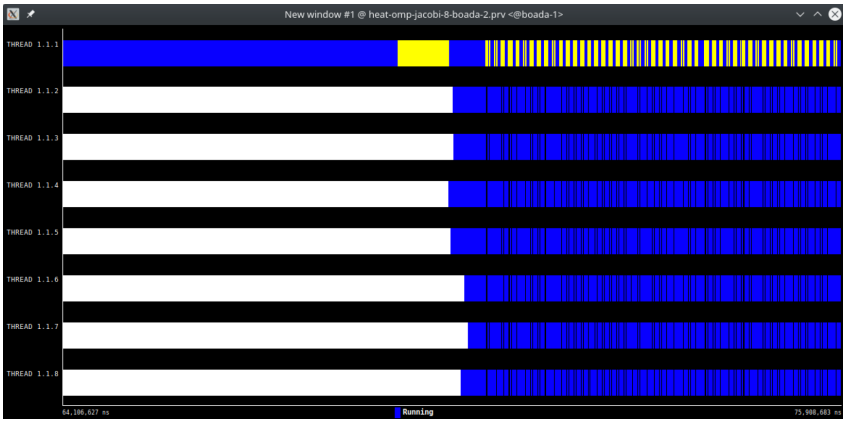

Figure 20: Trace of the new window

Finally, using the submit-strong-omp.sh script to queue its execution we got the scalability plots (Figure 21):

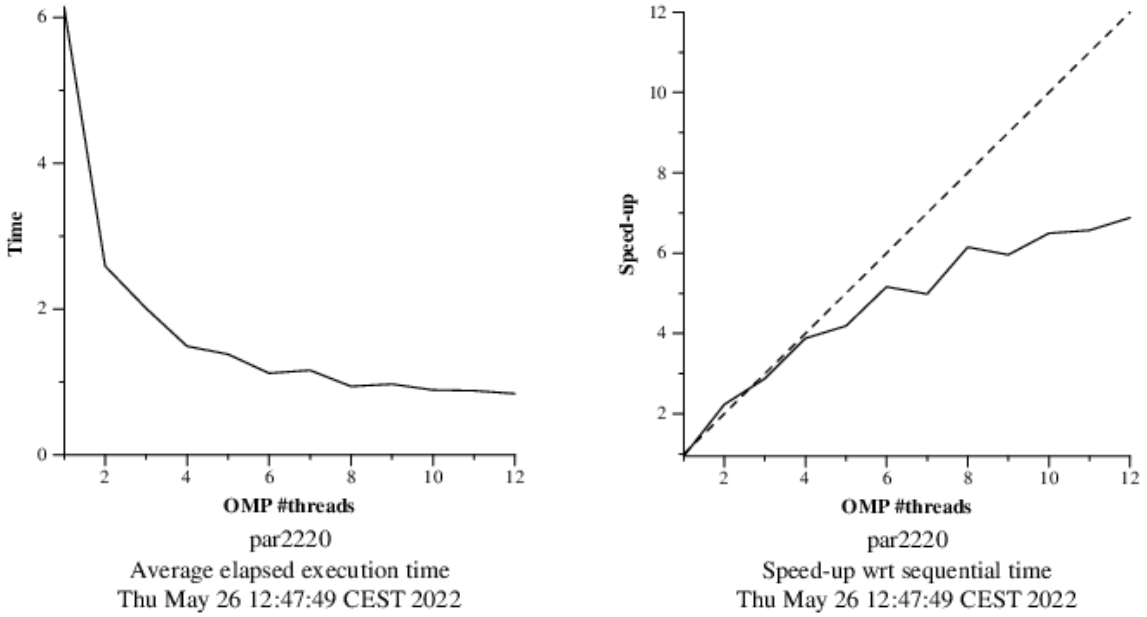

Figure 21: Execution Time plot (left) and Speed-up plot (right)

This time, we can see that the execution time decreases by increasing the number of threads and also the speed-up increases. This is due to the fact that we parallelised more regions than the initial version, so we obtained much better results and a better performance.

OpenMP parallelisation and execution analysis: GaussSeidel

For the parallel implementation of the Gauss-Seidel solver, we need to remember first the dependencies that appear when this solver is used, discovered before using Tareador.
First, we have to consider the sum, diff and tmp data variables; as those variables would be shared between all threads so data racing might be produced, they will need to be protected. Thus, firstprive constructs will be used for diff and tmp variables, while a sum reduction will be implemented for the sum variable, since it is a shared value needed after the parallel region.
Next, for the dependencies produced by the access and storage of data within the same matrix, task ordering constraints will be used to simulate task depend clauses for implicit tasks. In order to respect the dependencies between blocks (where a certain block with indexes [i, j] depends of the blocks [i-1, j] and [i, j-1]), in addition to the geometric data decomposition specified, these constraints should be implemented so that a diagonal of block tasks cannot be executed before the previous diagonal has finished, and each thread executes a single row of blocks.
The horizontal dependencies are already solved by the assignment of the implicit tasks, since each thread executes a single row of blocks, this dependency is ensured to be protected. As for the vertical dependencies, we have to consider that the execution of a certain block task cannot begin before the block of above has finished. To accomplish this, a vector of size omp_get_max_threads() should be enough to keep track of which blocks can be executed by a certain thread: whenever a thread finishes the execution of a certain block, it should increment by 1 the **blockj** position of this vector declared, meaning that the next block of the same column can start its execution (for each thread waiting, they will be able to continue the execution when the value in a certain **blockj** position is equal to their **blocki** level assigned).
The final implementation is shown below in figure 22. Note that if-conditional statements are used since both solvers share the same code and this implementation is only needed for the Gauss-Seidel. The output image is shown in figure 23, which has been verified to be exactly the same as the image generated by the sequential version by using the diff command.

```c
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=n_threads;

    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) {
          int flag;
          do {
            #pragma omp atomic read
            flag = next[blockj];
          }
          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;
}
```

Figure 22: function solve of the solver-omp.c

Figure 23: output image

After looking at the traces generated with Paraver shown in figure 24, we can say that the parallel behaviour did not match our expectations. By contrast with the timeline generated for the Gauss-Seidel with Tareador task creations, we can observe that this time there were not any black regions representing that a thread was waiting to start its execution, which leaded us to think that all implicit tasks were being generated and executed in parallel without respecting the dependencies defined. However, the reason for this to happen is that since the previously mentioned implicit task depend clauses are implemented by using while loops, Paraver will treat them as useful work (so blue regions are being defined), even though we consider these loops as task waits (useless work). Thus, although the traces are not really intuitive and may make the program seem to be incorrect, it is actually correct.
If we compare this timeline with the one obtained using the Jacobi solver (after parallelising the matrix copy), both are similar in terms of time waiting between tasks but, of course, Jacobi generates much more tasks.

Figure 24: Trace of the new window

By analysing the scalability plots obtained from the Gauss-Seidel solver shown in figure 25, even though the speed-up has a linear growth, it is not as good as the scalability obtained from the Jacobi solver as well as far from being ideal.

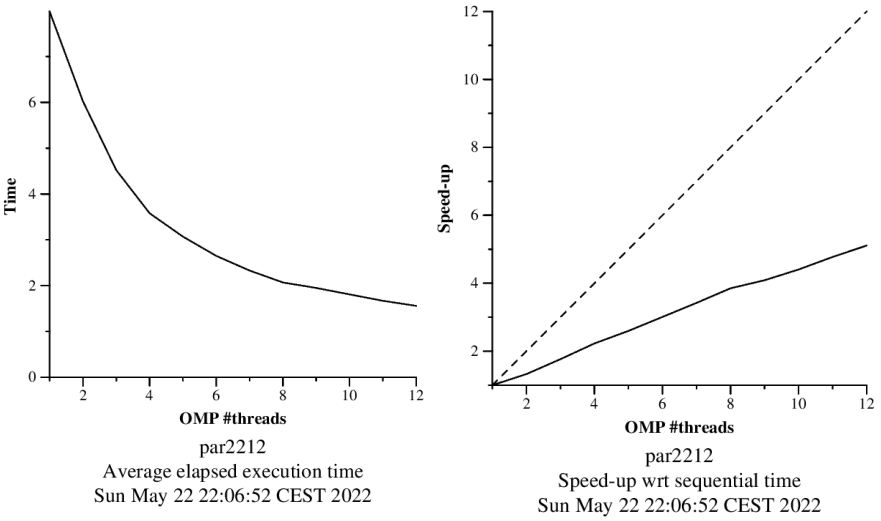

Figure 25: Execution Time plot (left) and Speed-up plot (right)

Finally, we are going to study the impact of changing the number of blocks in the j dimension. Figures 26 and 27 consist of graphs representing how the execution changes with the different number of threads and blocks.

As we can observe, the execution time can be reduced by incrementing both the number of threads and blocks. Intuitively, we can say that better parallelism can be achieved by incrementing the number of blocks in both the i and j dimension, which is the same as saying by reducing the granularity of tasks. The reason for this is that since with a finer granularity all tasks take less time to be finished, every thread can start its execution earlier and the time spent for synchronisation is being reduced. Even though this affirmation seems to be in contradiction with what we have been seeing along this course, where the increase of tasks usually leaded to an incrementation of the execution time caused by the overhead generated for synchronisation or cache misses, this time, the data decomposition strategy implemented allowed us to reduce this overhead by taking advantage of the principle of locality.

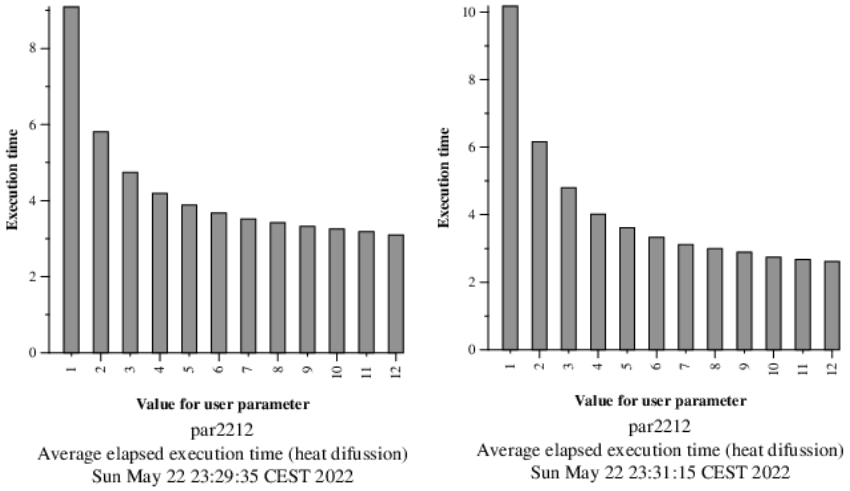

Figure 26: Execution Time plot with 4 threads (left) and with 8 threads (right)

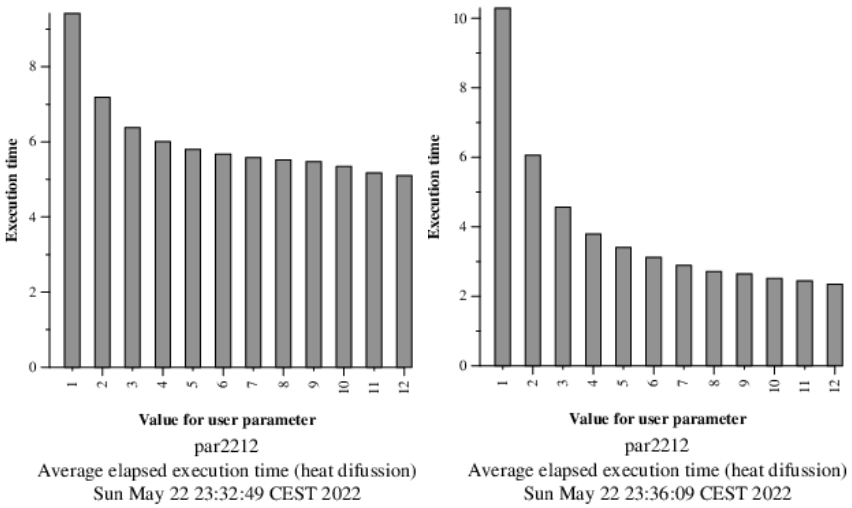

Figure 27: Execution Time plot with 2 threads (left) and with 12 threads (right)

Conclusions

Through this last laboratory assignment we have studied the impact of the data decomposition strategies by parallelising the Jacobi and Gauss-Seidel solvers for the heat equation.

For the Jacobi solver, first we have used the Tareador tool to analyse the potential parallelisation of the program using the Jacobi solver. By obtaining the TDG, we saw the dependencies that were created by different variables and how the tasks granularities can impact on the potential parallelisation. At first instance, we just introduced the parallelisation in the solve function of the solver, but we realised that the results were not good enough by looking at the different plots, because there were other regions that can also be parallelised. So, we also parallelised the copy matrix function. And this time, we saw that the results improved a lot. In the second session, we changed the code so it can be parallelised in the OpenMP program and we analysed the execution. As in the first session, we first implemented the parallelisation on the solver function of the solver, and after seeing the poor performance, we also introduced the parallelisation on the copy matrix function and as we expected, the performance was much better.

For the Gauss-Seidel solver, we have analysed how dependencies were being generated caused by saving the results in the same matrix used for computation, which allowed us to correctly parallelise the solver by taking into account the different variables that were causing these dependencies. In this case, the results after implementing the parallelisation were not as good as for the Jacobi solver, in terms of scalability. However, by analysing how the number of threads and blocks in the j dimension would impact the execution time, we have realised that an increase of the number of blocks in both i and j dimensions could help to improve the performance of the program. This last observation was a proof of how important the data decomposition strategies can be, allowing less overhead to be generated by taking advantage of the principle of locality.