PAR Laboratory Assignment

Lab 5

Geometric (data) decomposition: solving the heat equation

Semester 2 - 2018

Group par2203

13/06/2018

Bernat Gené Škrabec

Ferran Ramírez Navajón

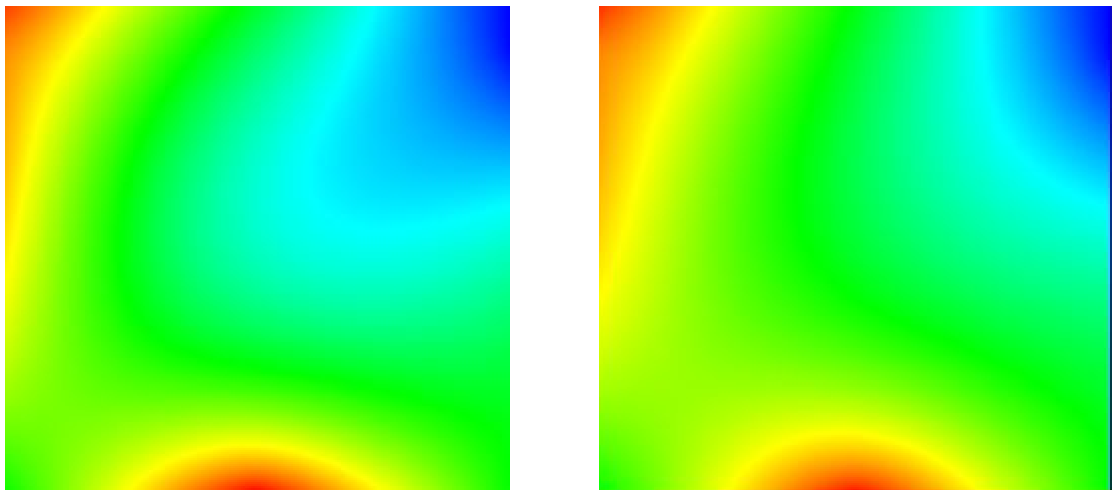
Index

Introduction

In this session we will explore new and known parallelization strategies to deepen our understanding and knowledge, and to better equip our parallelization toolbox.

The code on which we are going to work on is from a program that simulates heat diffusion in a solid body, using two different solvers for the heat equation (Jacobi and Gauss-Seidel).

Each of these solvers has different numerical properties and thus generate somewhat different results. These differences, however, are not relevant for the purposes of this assignment, we use them because of their different parallel behaviours.



In Fig0 we can see the images resulting from the execution of the program using either solver. The image shows the resulting heat distribution when two heat sources are placed on the borders if a 2D solid (one in the upper left corner and the other in the middle of the lower border). It is quite obvious what the color scheme represents (Red = hot, blue = cool).

Analysis with Tareador

As we did in other sessions, we will start by analyzing our code and trying to figure out the best possible parallelization strategies. To do so, a good tool is the Tareador API, which let's us define tasks and generates a graph simulating the dependencies between those tasks.

First of all, we ran the code "as is" without making any changes and analyzed it with Tareador. We tried it using both solvers, Jacobi and Gauss. Next we include the graphs we obtained.

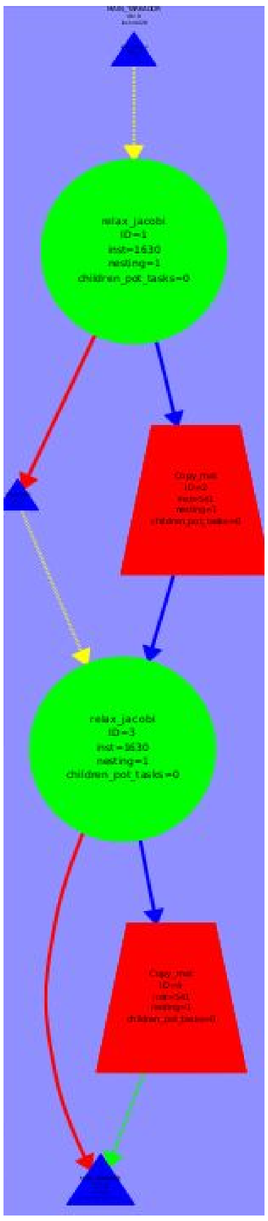


Fig1. Tareador graph of all the tasks generated with Jacobi solver using the original code

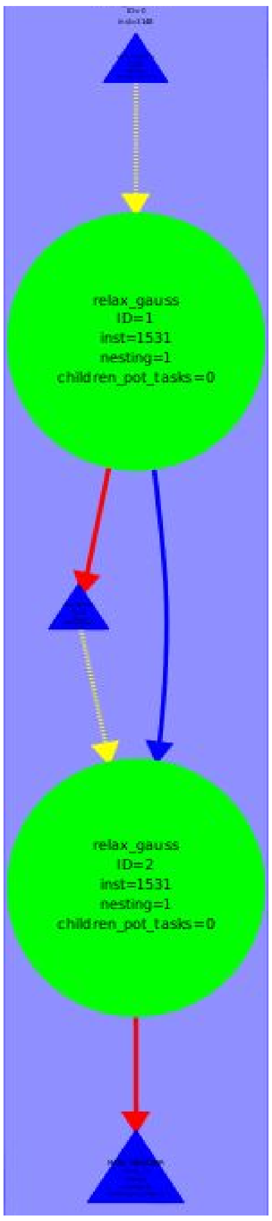


Fig2. Tareador graph of all the tasks generated with Gauss solver using the original code

Fig1 and Fig2, show how the default instrumentation of the code, which defined a task just for each call to the solvers, is completely serialized.

Next we will try a different strategy. We are going to instrument the code in “solver-tareador.c” so that a task is defined for each iteration of the innermost loop of the Jacobi function and later analyze the graph generated by Tareador. We will do the same for the Gauss solver.

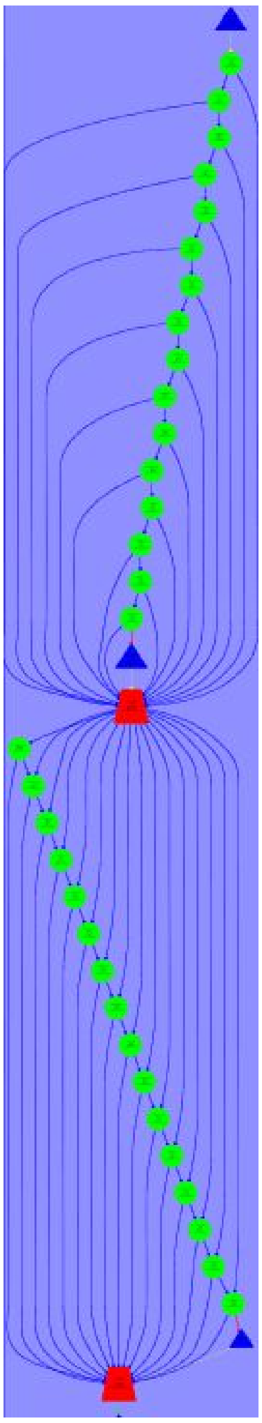


Fig3. Tareador graph of all the tasks generated with Jacobi solver using the first

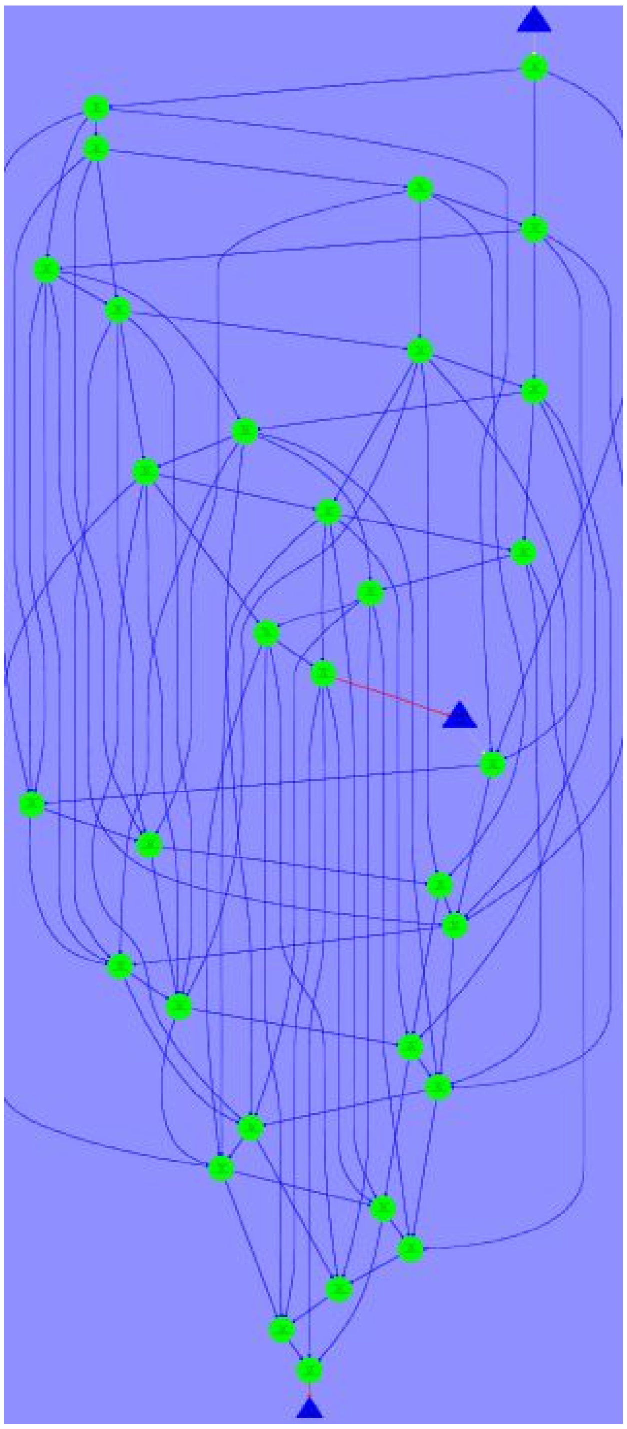


Fig4. Tareador graph of all the tasks generated with Gauss solver using the first strategy

strategy

As it can be observed in Fig3 and Fig4, in both cases, there is some variable that is causing dependencies between the defined tasks and thus the graphs generated, especially in the case of the Gauss solver, are a complete mess. Fortunately, the Tareador UI gives us enough tools to find the aforementioned variable. By clicking on the dependency edges, we can see what variables are causing them. This way we determined that in both cases, the variable responsible for the serialization was “sum”.

To generate the correct graph, we can disable the variable using the appropriate directive. The next figures show the graphs obtained for each solver when the serializing variable is no longer an issue.

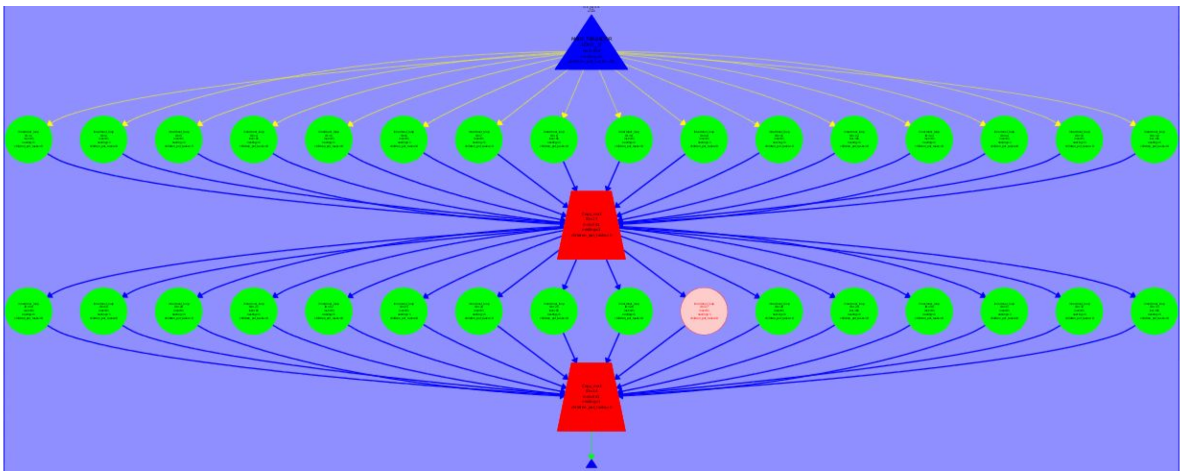


Fig4. Tareador graph using Jacobi solver with the correct instrumentation

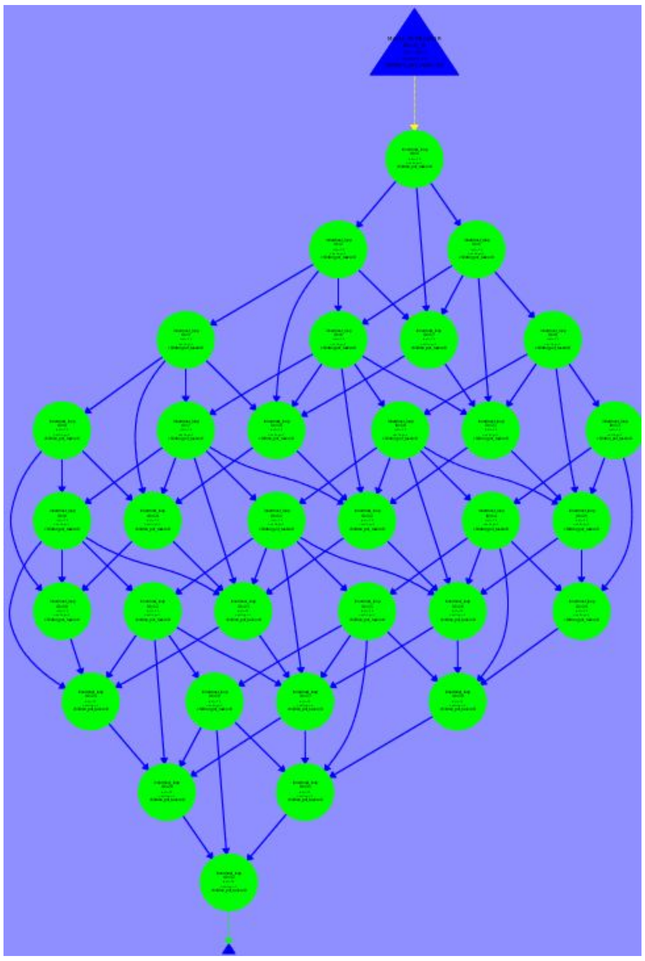


Fig5. Tareador graph using Gauss solver with the correct instrumentation

The previous figures, Fig4 and Fig5, show a much tider graph, now that the dependencies have been solved. Later, when we instrument the code using OpenMP for real parallelization we will have to use an appropriate directive to protect the “sum” variable in order to avoid serialization. A possible pragma which would solve this issue would be “critical” or “reduction”.

OpenMP parallelization and execution analysis: Jacobi

To better understand the inner workings of our code, first we will analyze how the data matrix is decomposed. In the header file, we have some macros defined that help to determine the appropriate size for each block in which the matrix is decomposed.

The method is quite straightforward; we divide the matrix in as many blocks as defined by the “how_many” variable, originally 4 (though it is meant to represent the number of processors). If the number of elements per column is not exactly divisible by the “how_many” variable, then the remainder rows are assumed by one of the blocks.

The following figure represents the decomposition with the variable “how_many” equal to 4, and a number of elements 3 modulo 4 (this is why the last block is slightly bigger).

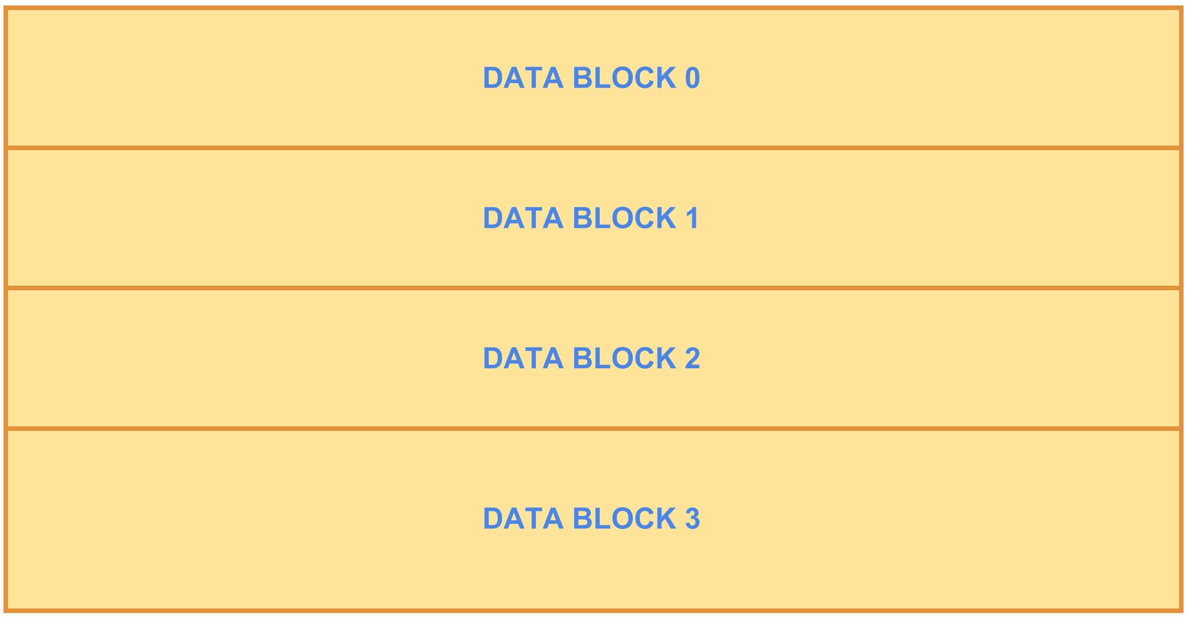
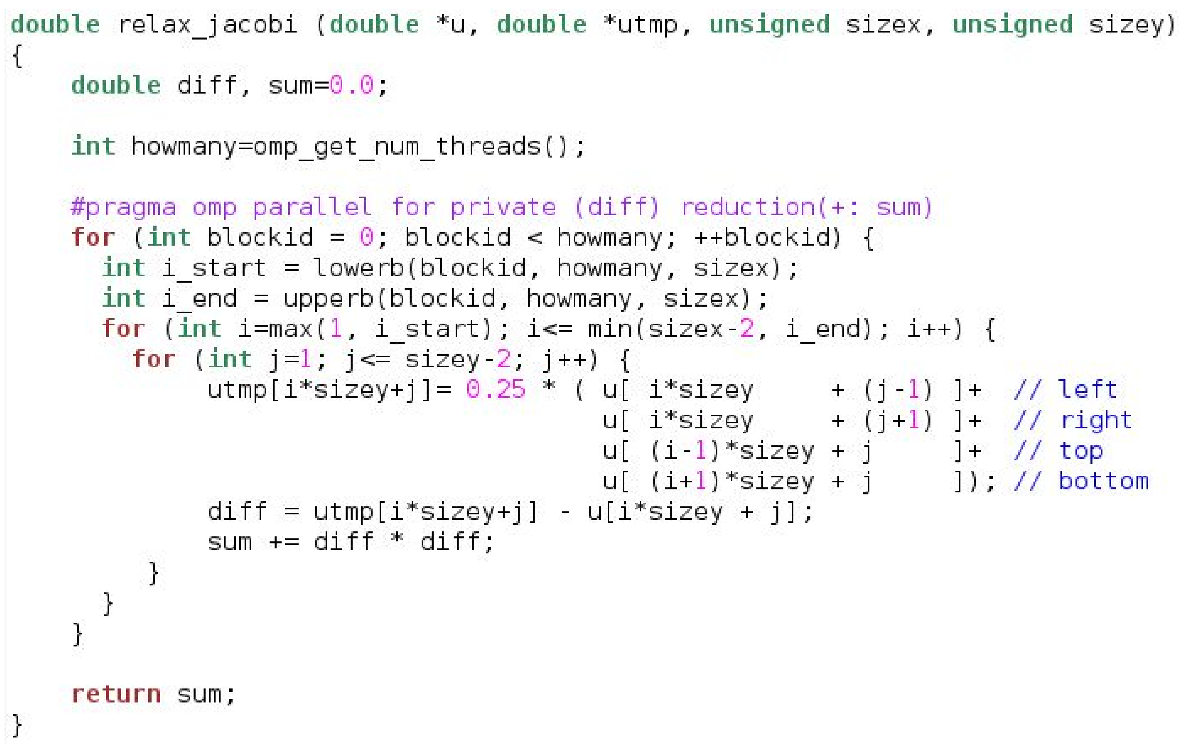


Fig6. Geometric decomposition

Next, we are going to modify the relax_jacobi function to parallelize it. We will include the changes made in the code. To make sure that our implementation is correct, we compared the image generated by our code and the original one using the “diff” command. It returned that the images are identical, but we have included them nonetheless because they look pretty (Fig9 and Fig10).



```c
double relax_jacobi (double *u, double *utmp, unsigned sizex, unsigned sizey)
{
    double diff, sum=0.0;

    int howmany=omp_get_num_threads();

    #pragma omp parallel for private (diff) reduction(+: sum)
    for (int blockid = 0; blockid < howmany; ++blockid) {
      int i_start = lowerb(blockid, howmany, sizex);
      int i_end = upperb(blockid, howmany, sizex);
      for (int i=max(1, i_start); i<= min(sizex-2, i_end); i++) {
        for (int j=1; j<= sizey-2; j++) {
             utmp[i*sizey+j]= 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 = utmp[i*sizey+j] - u[i*sizey + j];
             sum += diff * diff;
         }
      }
    }

    return sum;
}
```

Fig8. Code of the relax_jacobi function belonging to solver-omp.c

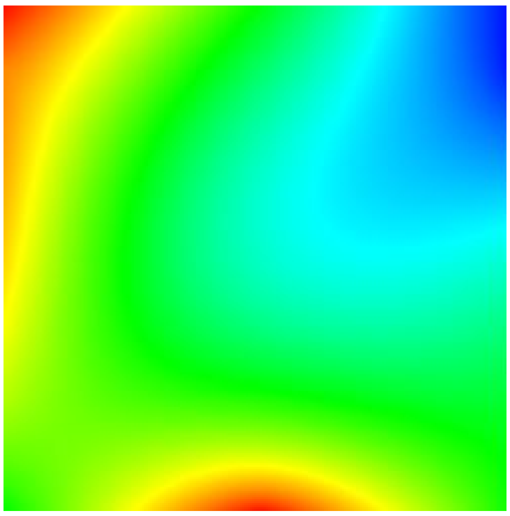


Fig9. Image generated with the original code using the jacobi solver

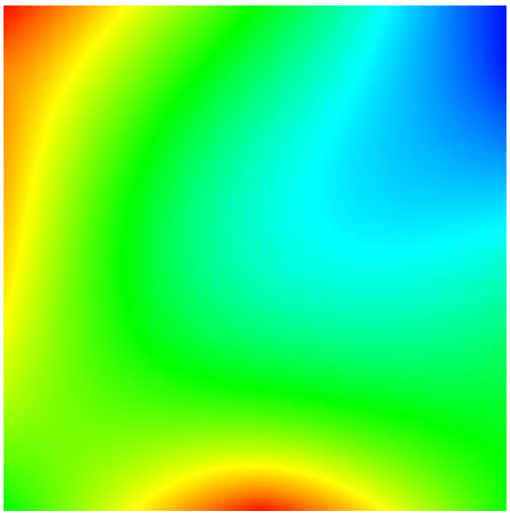


Fig10. Image generated with the new code using the jacobi solver

Now we will submit the “submit-omp-i.sh” script to generate Paraver traces and analyze the actual parallelization obtained. We will also submit the “submit-strong-sh” to determine the speed-up compared to the original sequential version. By zooming in on the trace, we see that the parallelization is not perfect.

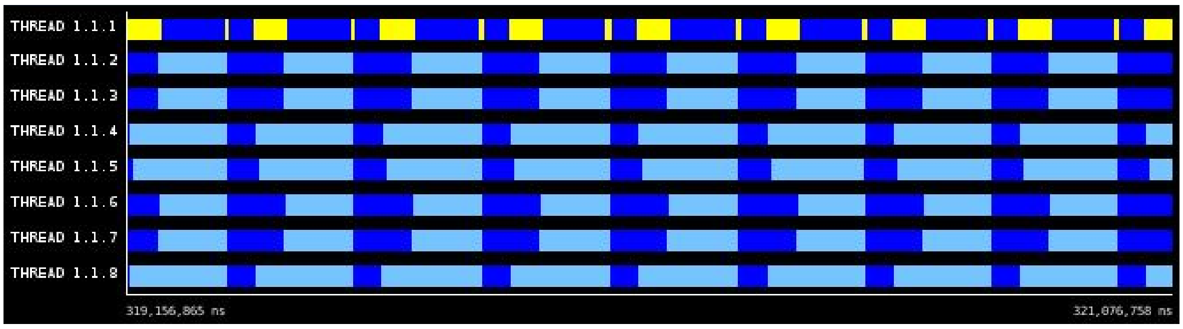
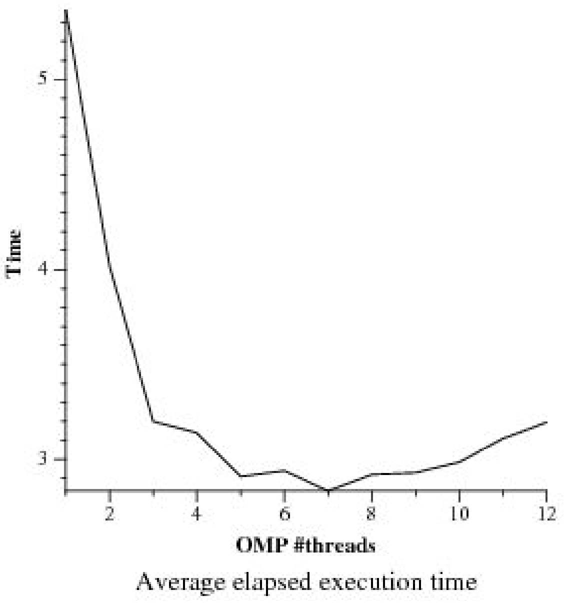


Fig11. Paraver trace of parallelisation in relax_jacobi function

Looking at the code we can see that the original has a serious flaw: the **howmany** variable is always set to 4, no matter the number of threads that are used. It is obvious that if we want the best load balance, we need to divide the matrix in as many equal parts as there are threads. This can easily be achieved by assigning to **howmany** the value given by the directive “omp_get_num_threads()”.

Fig11 shows the zoomed in region of the trace that demonstrates the flawed parallelism. After each call to the jacobi function, the matrix is copied using a sequential copy which is probably degrading our performance, as it is only ran by one thread. We will try to parallelize that function as well in an attempt to improve the performance.

We will include the execution time and speed-up plots of the first parallelization, which did not modify the copy function, and of the second one, which it dit.



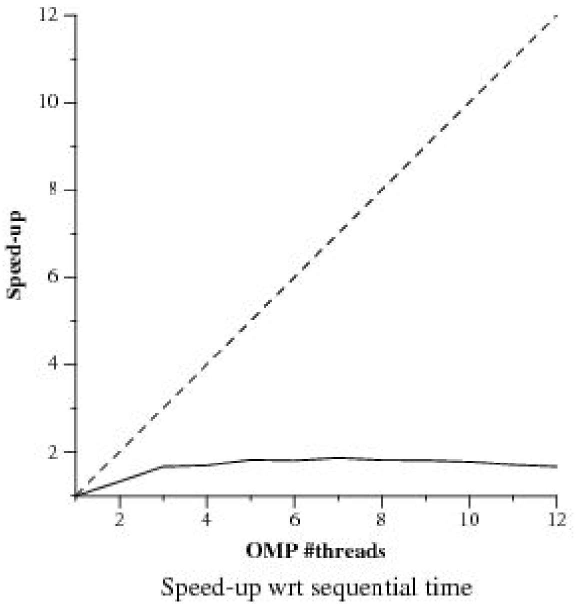


Fig12. Execution time plot of the solver-omp.c code using the jacobi solver (parallelized)

Fig13. Speed-up plot of the solver-omp.c code using the jacobi solver (parallelized)

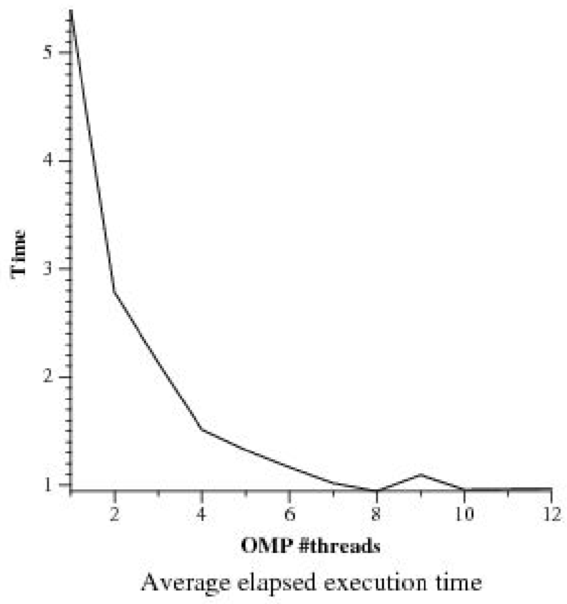


Fig14. Execution time plot of the solver-omp.c code using the jacobi solver (parallelized) with a modified copy_mat function

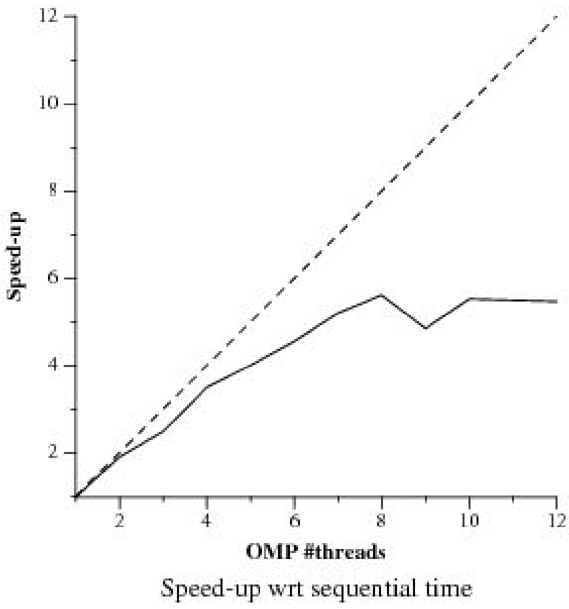
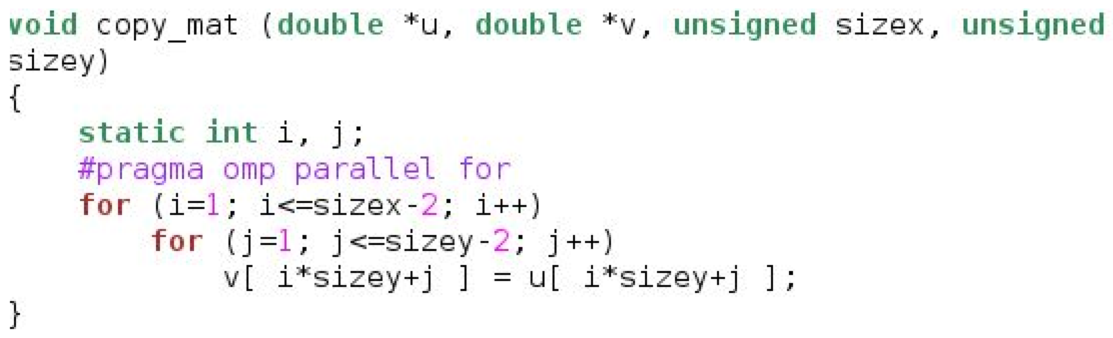


Fig15. Speed-up plot of the solver-omp.c code using the jacobi solver (parallelized) with a modified copy_mat function

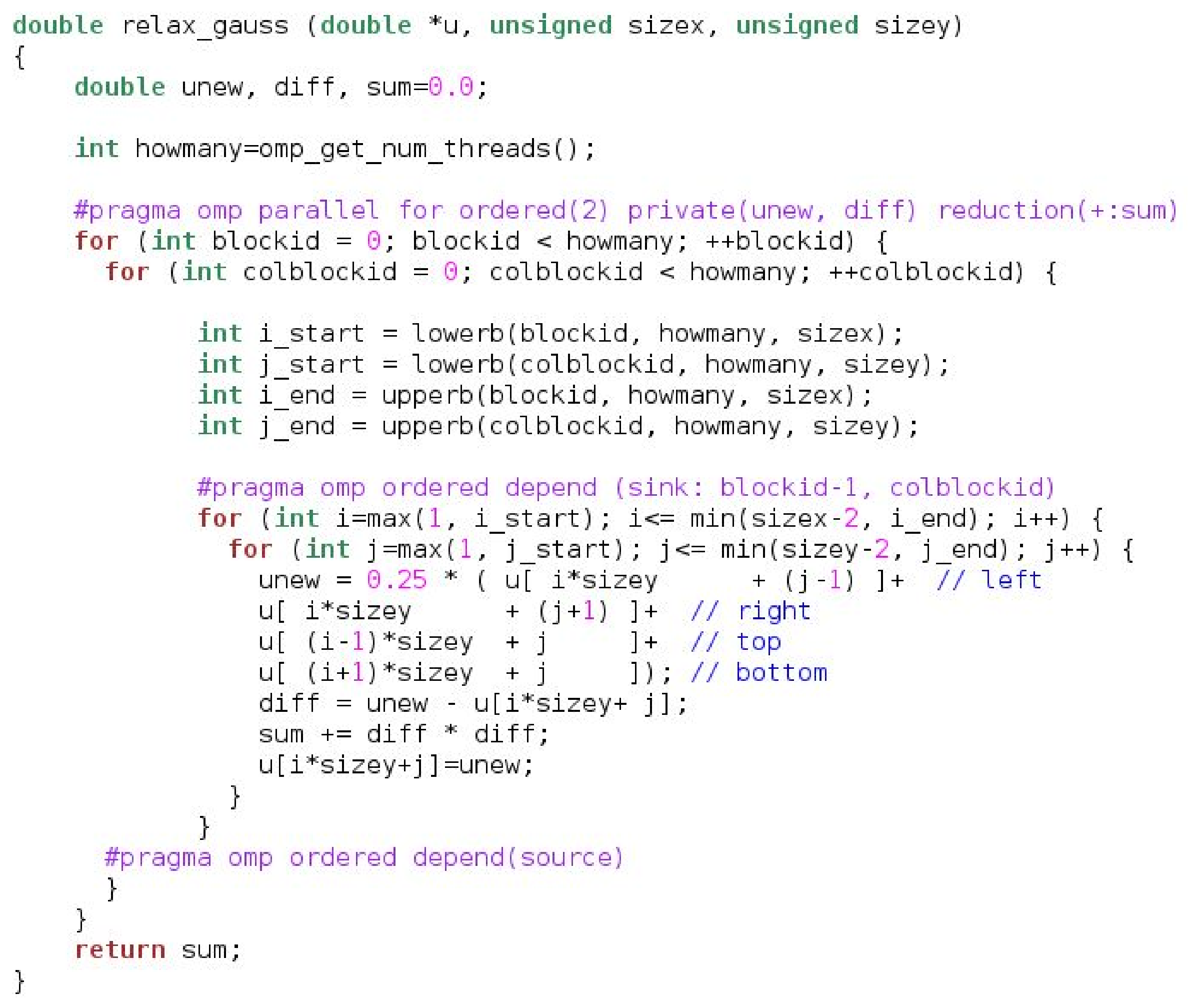
The previous figures (Fig12, 13, 14 and 15), show the plots we previously mentioned. We can observe that we were right in our assumption of the impact that the copy_mat function had in the performance. There is an obvious difference in speed-up between the version in which we didn't modify it and the version in which we did. The code can be found in the source folder included in the deliverable of this session, but we will include a capture of the relevant part of this last modification for clarity.



```
void copy_mat (double *u, double *v, unsigned sizex, unsigned
sizey)
{
    static int i, j;
    #pragma omp parallel for
    for (i=1; i<=sizex-2; i++)
        for (j=1; j<=sizey-2; j++)
            v[ i*sizey+j ] = u[ i*sizey+j ];
}
```

Fig16. Code of the copy_mat function belonging to solver-omp.c

OpenMP parallelization and execution analysis: Gauss-Seidel



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

    int howmany=omp_get_num_threads();

    #pragma omp parallel for ordered(2) private(unew, diff) reduction(+:sum)
    for (int blockid = 0; blockid < howmany; ++blockid) {
      for (int colblockid = 0; colblockid < howmany; ++colblockid) {

            int i_start = lowerb(blockid, howmany, sizex);
            int j_start = lowerb(colblockid, howmany, sizey);
            int i_end = upperb(blockid, howmany, sizex);
            int j_end = upperb(colblockid, howmany, sizey);

            #pragma omp ordered depend (sink: blockid-1, colblockid)
            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++) {
                unew = 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 = unew - u[i*sizey+ j];
                sum += diff * diff;
                u[i*sizey+j]=unew;
              }
            }
      #pragma omp ordered depend(source)
      }
    }
    return sum;
}
```

Fig17. Code of the relax_gauss function belonging to solver-omp.c

Fig17 is an excerpt of the code which shows the parallelization of the relax_gauss function.

As we saw in the previous section, the one where we used Tareador, the task dependencies in this solver are quite intricate. Therefore, the parallelization strategy is equally complicated. The main concept we have to keep in mind is how each subsequent task is created from the previous one and their dependencies. Looking at the task graph, Fig5, we can observe the following: A first task is created at the highest level. This one creates two subsequent tasks in the lower level that depend from it. Those two, at their time, create two subsequent tasks, but the dependencies begin to be more complicated. This exponential behaviour has a limit, after which, at each level, the number of tasks decreases.

The conclusion of this analysis is that the tasks can be organized by levels and that each level is dependent from the previous one. Also that, at every level, the number of tasks is doubled, with a limit after which it start to shrink again in the same manner.

This is the behaviour which our pragmas have to keep under control. The first directive uses **for** and its **ordered** clause. We also use **reduction** for the *sum* variable, which is useful when all threads accumulate values into a single variable, exactly the case with *sum*. Then we use **depend** to declare the dependencies that we previously mentioned, and define where the dependent region ends for the next iteration using **depend(source)**.

To make absolutely sure that our code is still correct, we submitted it using the appropriate script to generate the image. We compared the resulting image using “diff” with the original (obtained with the gauss method) and the result indicated they were identical. We decided to include both of them anyway in order to make this report more colorful.

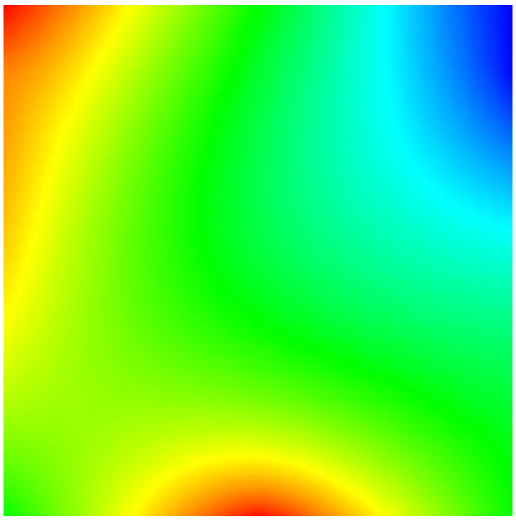


Fig18. Image generated with the original code using the gauss solver

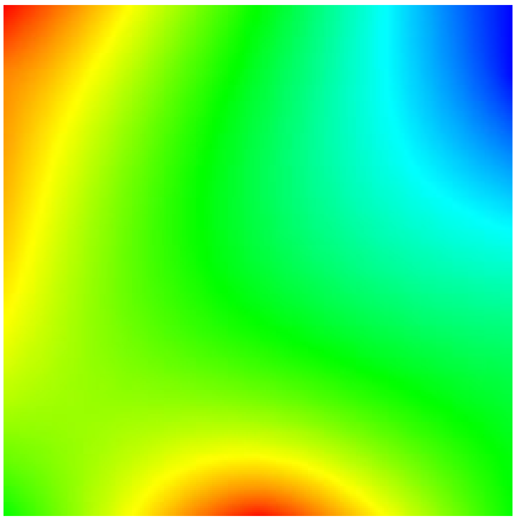


Fig18. Image generated with the new code using the gauss solver

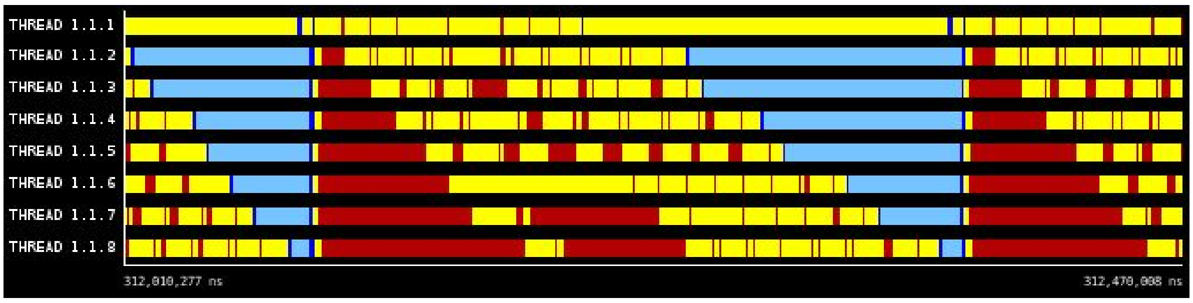


Fig19. Paraver trace of parallelisation in relax_gauss function

Fig19 is a zoomed in paraver trace that exemplifies the intricate behaviour of our program we previously commented. Next we can see the result of submitting the strong scalability script.

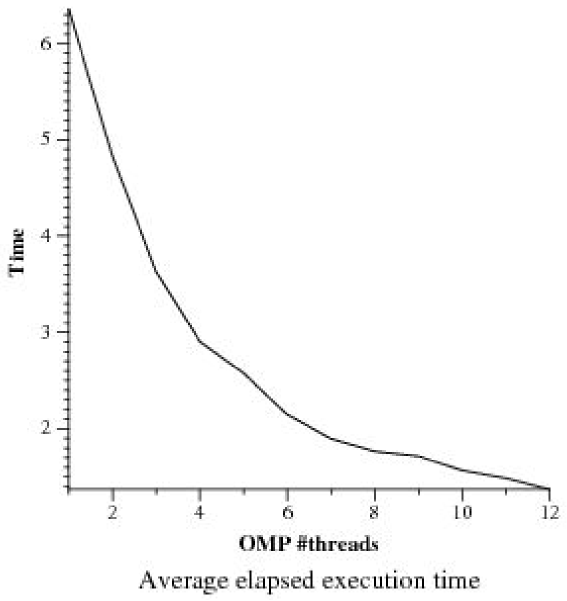


Fig23. Execution time plot of the solver-omp.c code using the gauss solver (parallelized) with an appropriate block size

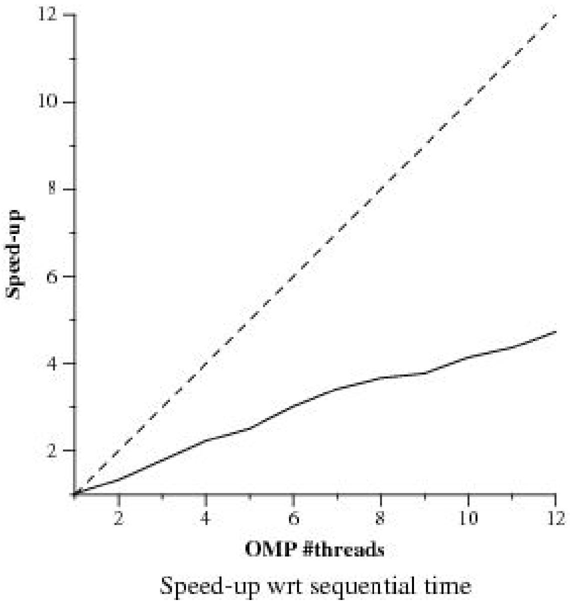


Fig24. Speed-up plot of the solver-omp.c code using the gauss solver (parallelized) with an appropriate block size

A way to control the trade-off between computation and synchronization is to change the number blocks. If we have fewer blocks, there will be less time spent in synchronization, but if we have more, each block will require less computation, obviously. Originally, similar to the jacobi solver, we have determined that the best number of blocks will be the number of threads we are using, but we will test if that is true.

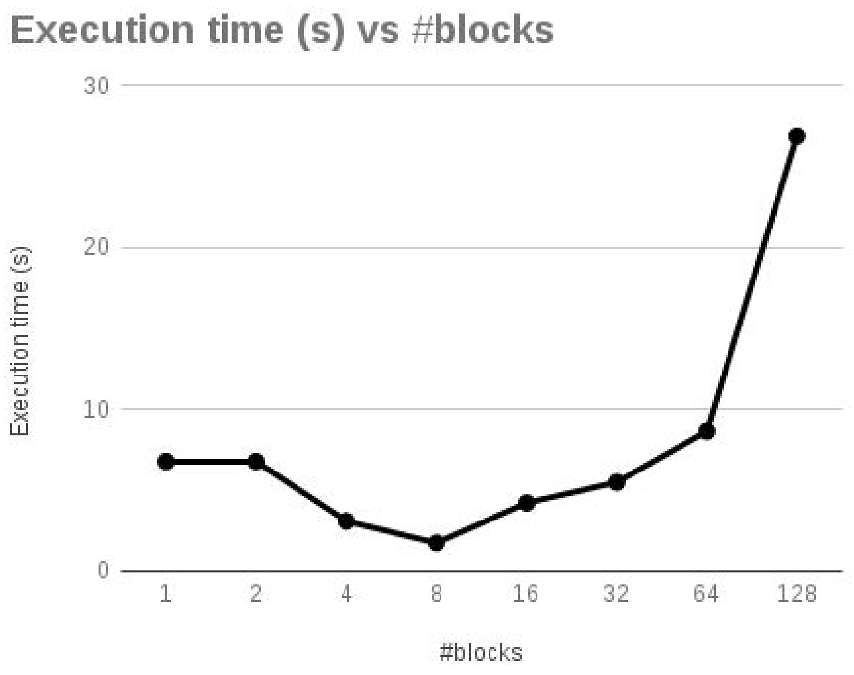


Fig25. Execution time in seconds versus block size chart

As we can see in the previous plot (Fig25) we were right about the block size. The best possible number of blocks is eight.

Optional

In this section we will try a different strategy to implement the parallelization of the gauss solver. The idea is to use the directive “task” and task dependencies instead of “for”, and compare the performance of the two implementations. The following section of the code shows our implementation.

```
double relax_gauss (double *u, unsigned sizex, unsigned sizey){
    double unew, diff, sum_all=0.0;
        int howmany=omp_get_num_threads();
        int howmany_col=omp_get_num_threads();
        char dependency[howmany][howmany_col];
        omp_lock_t lock;
        omp_init_lock(&lock);
        for (int blockid = 0; blockid < howmany; ++blockid) {
            int i_start = lowerb(blockid, howmany, sizex);
            int i_end = upperb(blockid, howmany, sizex);
            for (int z = 0; z < howmany_col; z++) {
                int j_start = lowerb(z, howmany_col, sizey);
                int j_end = upperb(z,howmany_col, sizey);
                #pragma omp task firstprivate (j_start,j_end, i_start,
i_end) depend(in: dependency[max(blockid-1,0)][z], dependency[blockid][max
0,z-1)]) depend (out: dependency[blockid][z]) private(diff,unew)
                {
                    double sum=0.0;
                    for (int i=max(1, i_start); i<= min(sizex-2, i_end); i+
+) {
                        for (int j = max(1, j_start); j<= min(j_end,
sizey-2); j++) {
                            unew= 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 = unew - u[i*sizey+ j];
                            sum += diff * diff;
                            u[i*sizey+j]=unew;
                        }
                    }
                    omp_set_lock(&lock);
                    sum_all=sum+2;
                    omp_unset_lock(&lock);
                }|
            }
        }
        omp_destroy_lock(&lock);
        return sum_all;
    }
}
```

Fig26. Gauss function code using tasks and task dependencies

The performance results of this versions are very similar when using a fixed number of threads. However, when executing with an increasing number of threads, we see that it does not scale very good, probably by the time lost in the creation of tasks and the synchronization between an increasing number of threads to execute them.

Conclusions

After finishing this session, and in fact all of them as this is the last one, we can say that we are quite satisfied with the work we have done. There are many concepts and tools we have acquired and thanks to the practical work done by using them in the laboratory, they have been heavily consolidated. There is an enormous difference between learning about parallelization only in theory, listening in class and doing exams, and actually using the concepts learned to parallelize real programs and observe first-hand the potential of this discipline.

In this session in particular, we compared two different approaches to solve the same task; computing the temperature of a 2D solid when multiple heat sources are in contact with it. Those to solvers were particularly interesting because of their differences. We saw that the Gauss solver, in general, had a better performance, probably to its smaller grain-size. However, as exemplified by the dependency graph generated by Tareador, this smaller grain-size also came with the drawback of very complicated and tight knit dependencies.

We also took a look at geometric data decomposition. We saw how dividing the data structures used in the code can have an important impact in performance and, if we choose its structure wisely, we can get great benefits. There are many ways to divide a matrix or vector so that each of the threads running in parallel can work on a section of it without tripping one with the other. However it will always depend on the idiosyncrasy of the problem at hand and therefore require great thought and carefulness.