PAR Laboratory Assignment
Lab 5: Geometric (data) decomposition: heat diffusion equation

Alexandre Fló Cuesta

QT 20/21

Index

1. Introduction

In this laboratory assignment we will work on the parallelisation of a sequential code (heat.c) that simulates the diffusion of heat in a solid body using two different solvers for the heat equation (Jacobi and Gauss-Seidel). Firstly, we are going to analyse with Tareador, then parallelise the Jacobi solver and we will finish parallelizing the Gauss-Seidel solver.

2. Analysis with Tareador

In this section, we will see the task graphs generated with Tareador of the different solvers in order to see the dependencies.

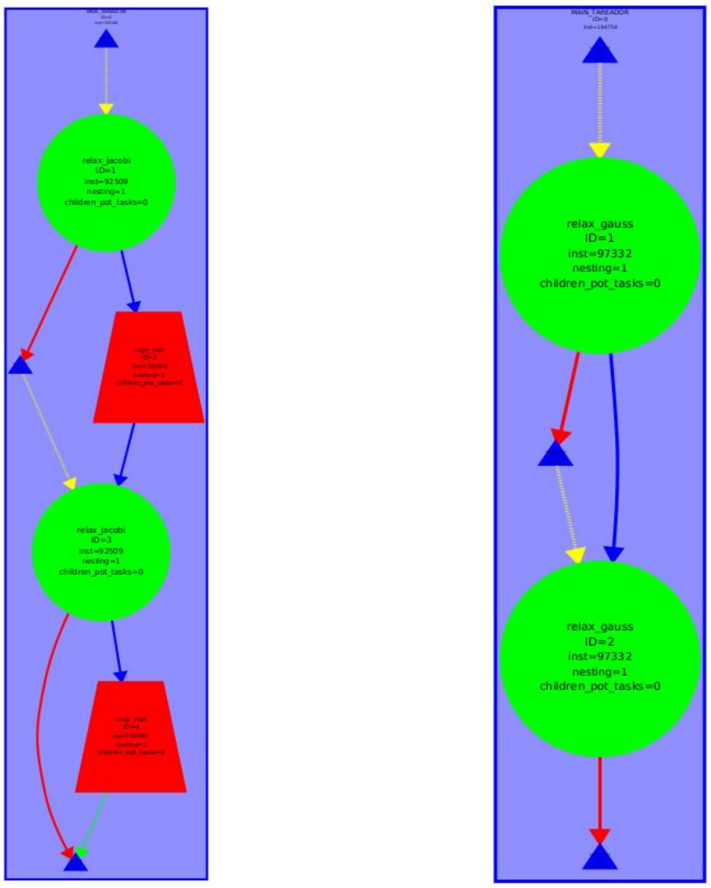


Figure1: Task-dependencies graphs. Jacobi and Gauss-Seidel

The granularity level we wanted to explore for the tasks was one task per block. Because of the dependency problems caused by the variable “sum” we need to ignore that variable using the function “tareador_disable_object(&object)”.

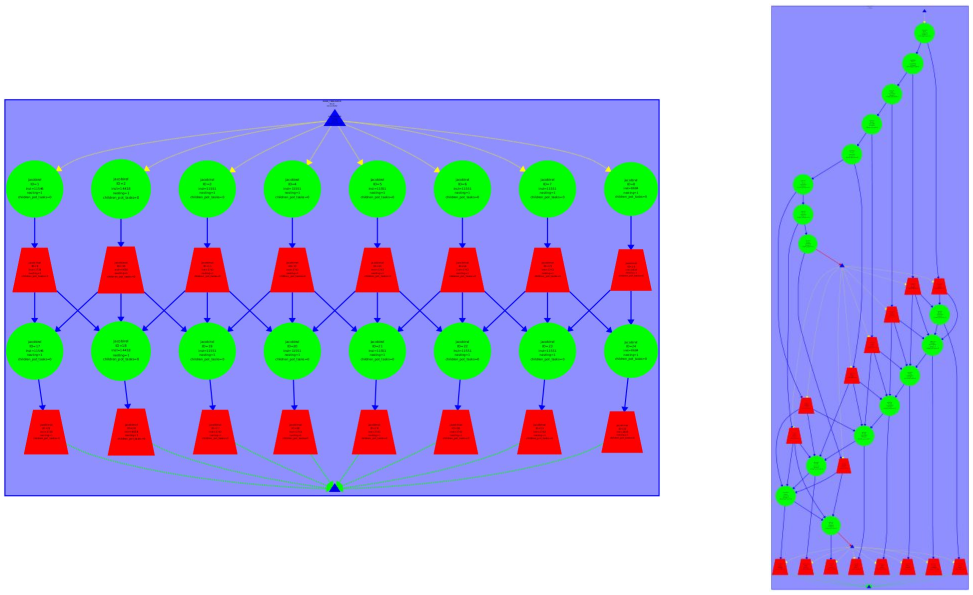


Figure2: Jacobi task dependencies graph with and without the “sum” commented in the code.

Comparing the two graphs (figure2) we can see that indeed the sum variable gave problems. We also can see an improvement in the parallelism and in the load balancing. The Jacobi solver includes an auxiliary function called copy_mat , used for copying a matrix, that can also be parallelized.

Same happens with Gauss-Seidel solver, but it seems to be less parallelizable than the other one.

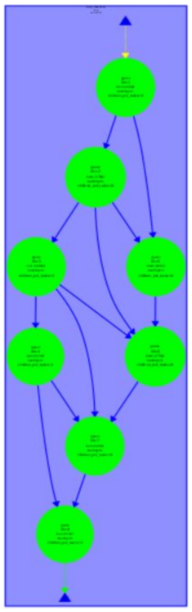


Figure3: Gauss-Seidel dependencies graph

3. Parallelisation of the Jacobi solver

In this section we will parallelise the sequential code for Jacobi using the implicit tasks in `#pragma omp parallel`, following a *geometric block data decomposition by rows*, as shown in for 4 threads running on 4 processors (figure4).

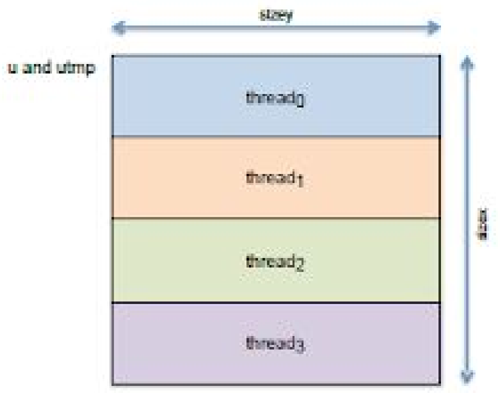
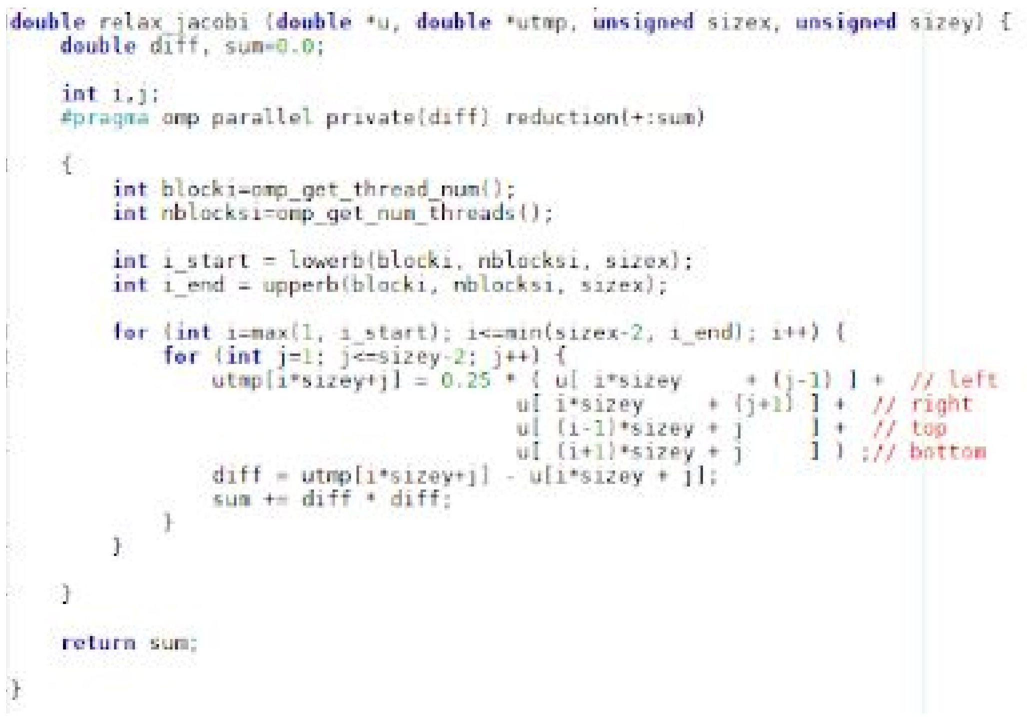


Figure4: Geometric (data) decomposition for matrix u (and utmp) by rows, for 4 threads.

We started parallelizing the code of the function `relax_jacobi`. We have implemented a `#pragma omp parallel` as said before. Specifically we made the variable "diff" private and a reduction for the variable "sum".



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

    int i,j;
    #pragma omp parallel private(diff) reduction(+:sum)

    {
        int blocki=omp_get_thread_num();
        int nblocksi=omp_get_num_threads();

        int i_start = lowerb(blocki, nblocksi, sizex);
        int i_end = upperb(blocki, nblocksi, 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;

}
```

We checked that the parallelization was done correctly using the command `diff` to make sure that both heat maps are identical (figure5).

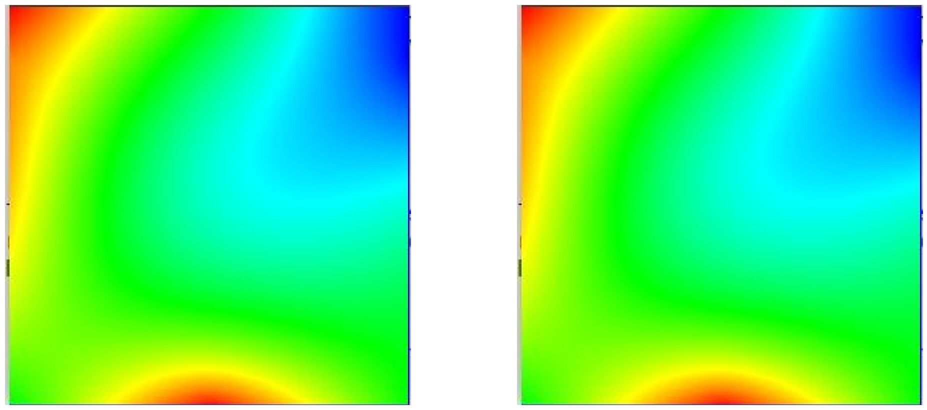


Figure5: Heat maps for sequential (left) and parallelized (right) versions of the code.

In order to further understand the performance of the Jacobi solver, we used the *Extrae* tool and Scalability analysis to see how good was our parallelization.

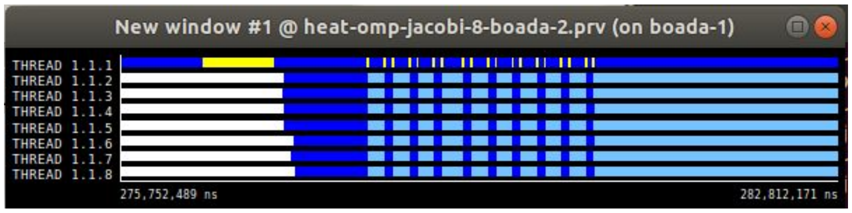


Figure6: Paraver trace without `copy_mat` function parallelized

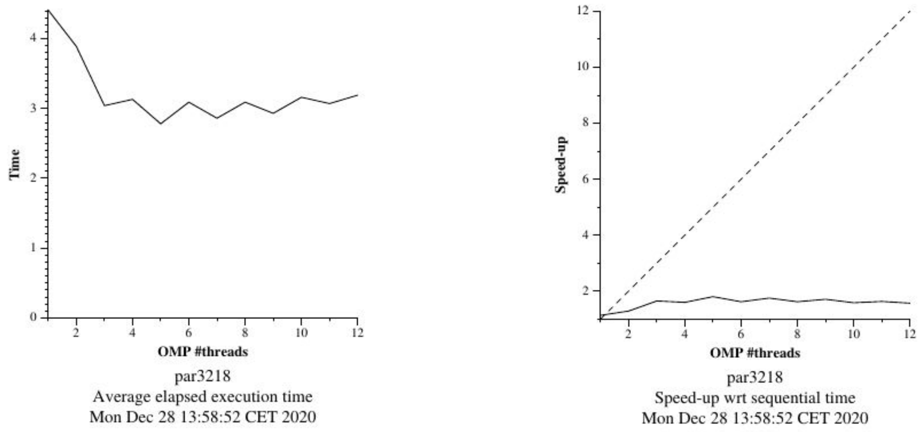
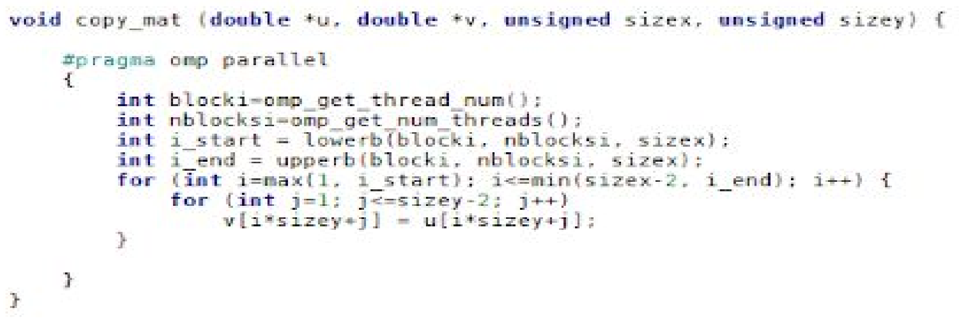


Figure7: Strong Scalability for Jacobi Solver (no `copy_mat`)

We can observe that it was not as good as expected (figure6 and 7). As we increment the number of threads we expect a reduction of the execution time, but it is somehow stable from thread 4 and the speed-up is constant. That is due to the fact that the Jacobi solver uses the `copy_mat` function which was not parallelized.



```
void copy_mat (double *u, double *v, unsigned sizex, unsigned sizey) {

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

    }
}
```

Once the parallelization is complete, there is a noticeable improvement of the load imbalance, and it seems that the threads are parallelizing all at once during more time and improving the efficiency of the program. (figure8 and 9).

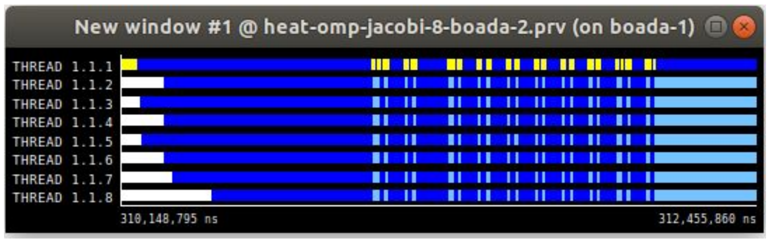
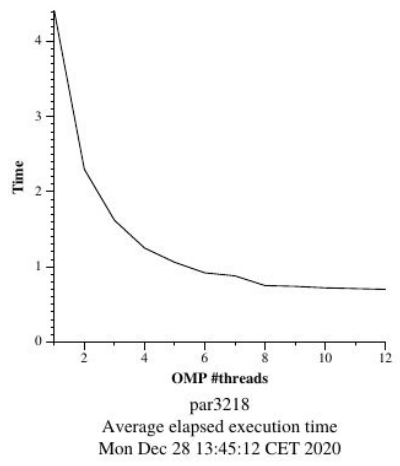


Figure8: Paraver trace with `copy_mat`function parallelized



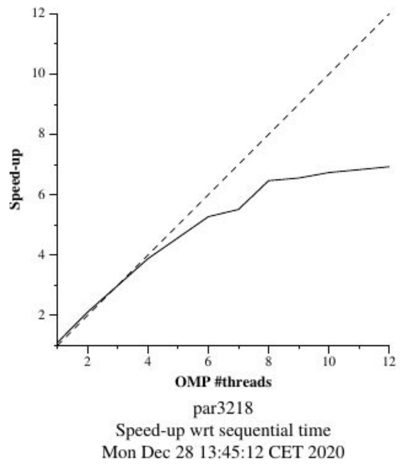


Figure9: Strong Scalability plots for Jacobi Solver (with `copy_mat`)

4. Parallelisation of the Gauss-Seidel solver

Finally in this session we will parallelise the Gauss-Seidel solver, following the same geometric block by row data decomposition that is shown in Figure4. In order to respect the dependences among tasks you discovered using Tareador, you should introduce the necessary synchronization and data sharing constraints among implicit tasks.

In Jacobi we assigned a block of rows to each thread, for Gauss-Seidel we will do the same but we have to do additional synchronization. The process will be at the block level.

Each thread before starting to process a block, it has to make sure that the same block from the previous thread has already finished, and that's why we need to do this synchronization explicitly. A synchronization vector is required to implement this and it will have as many entries as threads and must indicate the block it has just processed. This will be helpful to avoid false sharing and data races.

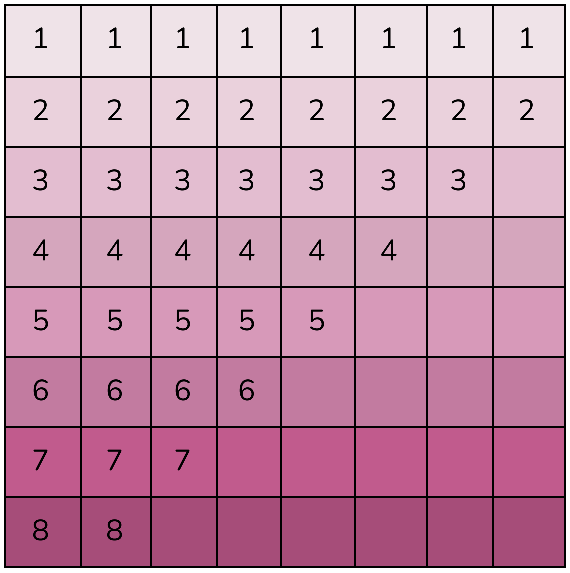
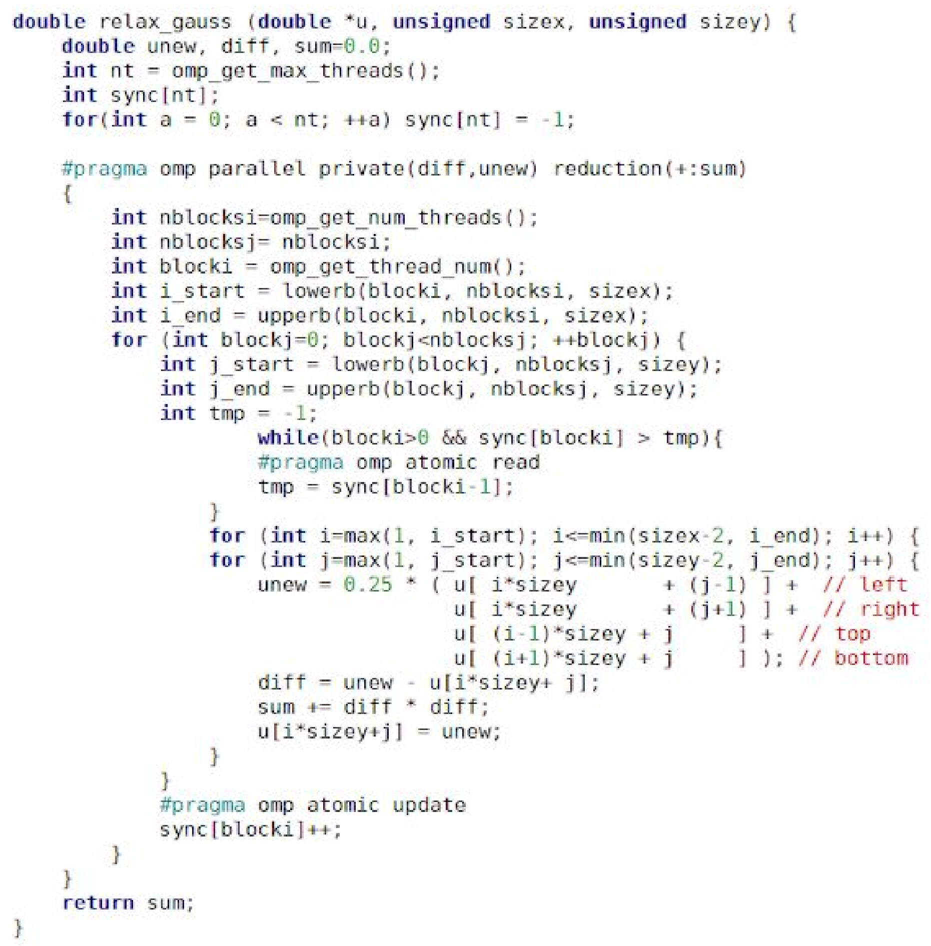


Figure10: Possible thread distribution per block

Access to this sync vector must be done with `#pragma atomic update` or `#pragma atomic read` to ensure that the last value written in the vector entry is read.



```
double relax_gauss (double *u, unsigned sizex, unsigned sizey) {
    double unew, diff, sum=0.0;
    int nt = omp_get_max_threads();
    int sync[nt];
    for(int a = 0; a < nt; ++a) sync[nt] = -1;

    #pragma omp parallel private(diff,unew) reduction(+:sum)
    {
        int nblocksi=omp_get_num_threads();
        int nblocksj= nblocksi;
        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);
            int tmp = -1;
                    while(blocki>0 && sync[blocki] > tmp){
                    #pragma omp atomic read
                    tmp = sync[blocki-1];
                }
                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 atomic update
            sync[blocki]++;
        }
    }
    return sum;
}
```

The parallelization seems to be working as expected because as we can observe in the plots (figure11 and 12) the speed-up seems reasonable and the execution time is constantly decreasing while increasing the number of threads. We can also notice at The Paraver trace that the parallel part of the program is well distributed among all the threads.

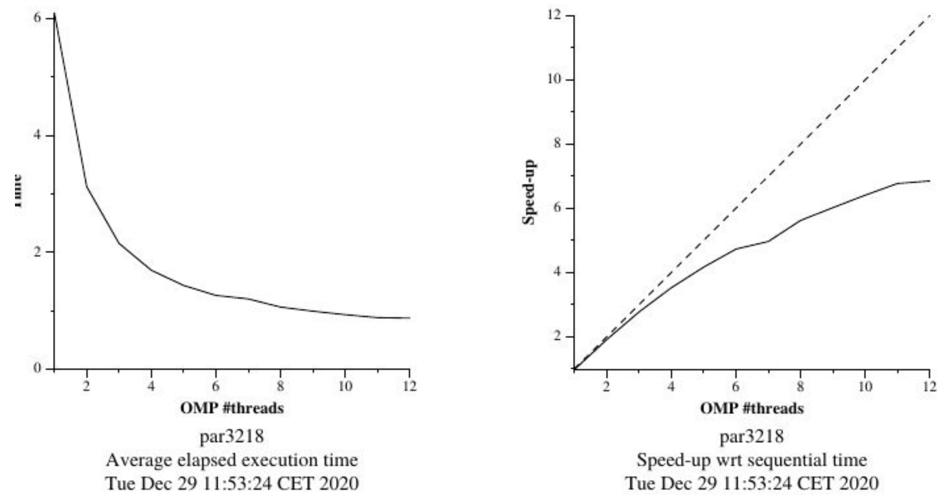


Figure11: Strong Scalability plots Gauss-Seidel solver

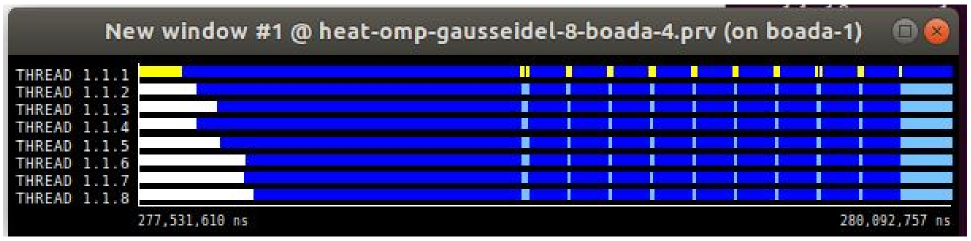


Figure12: Paraver trace Gauss-Seidel solver

In order to exploit more parallelism in the execution of the solver, we changed the number of blocks in the j dimension just to verify if changing the number of blocks in the j dimension changes the ratio between computation and synchronisation. With submit-userparam-omp.sh script we explored for a different range of values (8 threads and values 1 to 12). It seems that the execution time has its minimum value when the userparameter is 4.

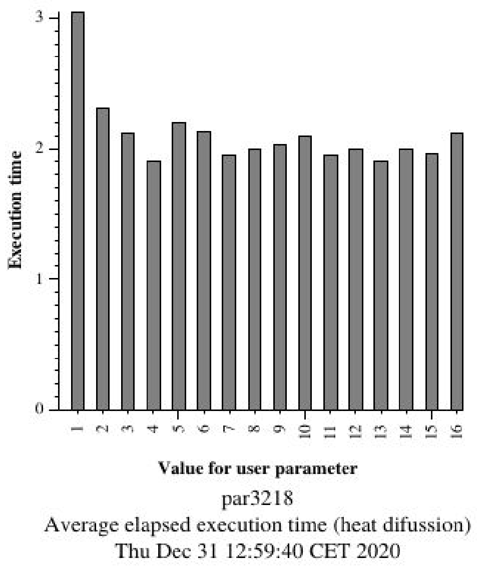


Figure13: UserParameter/Time plot 8 threads

5. Conclusion

During this session we have analysed the different possibilities of parallelization for two different solvers for the heat distribution. We can conclude that the Jacobi-solver had a better performance because the speed-up seems to be slightly better.
Moreover, it is very important to be aware of the dependencies between tasks so we can avoid false sharing, memory inconsistency and load imbalance issues. Preventing those problems will help us in the process of parallelization of a program.