

PARALLELISM

Laboratory 5

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

Index

1. Introduction	page 1
2. Analysing strategies	
2.1 Analysing sequential head diffusion	page 2
2.2 Analysing parallelism	page 3
3. Parallelization strategies	
3.1 parallelization Jacobi	
3.1.1 OpenMP parallelization Jacoby with serialization	page 5
3.1.2 Jacoby without serialization	page 5
3.2 Gauss-Seidel parallelization	page 6
4. Performance evaluation	
4.1 Performance evaluation of Jacoby	
4.1.1 Performance evaluation of Jacoby with serialization	page 8
4.1.2 Performance evaluation of Jacoby without serialization	page 9
4.2 Performance evaluation of Gauss-Seidel	page 11
5. Conclusions	page 13

1. Introduction

In this laboratory we have to parallelize a sequential code that simulates the diffusion of heat in a solid body using different solvers in the heat equation. That solvers are Jacobi and Gauss-Seidel and the characteristics are the next:

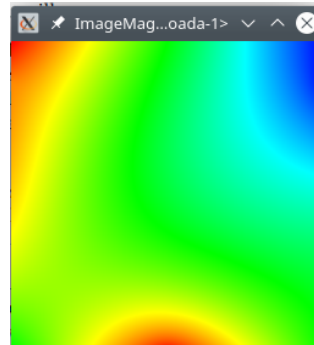
- Jacoby: in this strategy uses 2 matrix (one for the entry and another one for the exit). So the entry will be the matrix to compute and the exit the computed matrix.
- Gauss-Seidel: This strategy only needs one matrix reducing the memory, but it will need more depends.

We can see the images computed by the different strategies:

JACOBI



GAUSS-SEIDEL



As we can see, the images are not identical. We can see some differences between the light blue and green zone and the bottom red, yellow and green zone.

2. Analysing strategies

2.1 Analysing sequential head diffusion

In this section I will analyze the dependencies in the sequential code to see the dependencies that it has. I will do it with taredor that will show the TDG

```
// 2D-blocked solver: one iteration step
double solve (double *u, double *unew, unsigned sizex, unsigned sizey) {
    double tmp, diff, sum=0.0;

    int nblocksx=4;
    int nblocksy=4;

    //taredor disable object(&sum);
    for (int blocki=0; blocki<nblocksx; ++blocki) {
        int i_start = lowerb(blocki, nblocksx, sizex);
        int i_end = upperb(blocki, nblocksx, sizex);
        for (int blockj=0; blockj<nblocksy; ++blockj) {
            int j_start = lowerb(blockj, nblocksy, sizey);
            int j_end = upperb(blockj, nblocksy, sizey);
            taredor_start_task("subblock_i");
            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;
                }
            }
            taredor_end_task("subblock_i");
        }
    }
    //taredor_enable_object(&sum);

    return sum;
}
```

To make the TDG I add a taredor_start_task("subblock_i"); and a taredor_end_task("subblock_i"); as we can see in the image on the left:

The tasks are there because the 2 first for are to define the subblocks and the other two are to iterate every block.

If we have a look on the initial code, we will see the nexts TDG:

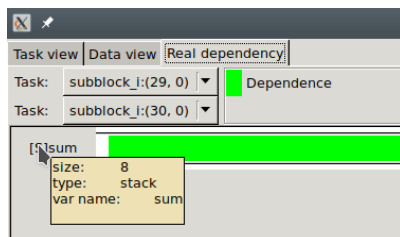
TDG Jacobi



TDG Gauss-Seidel



As we can see, both TDG are sequentially by the dependencies and Gauss-Seidel have more dependencies because it will need more.



If we do right click on an arc between the tasks generated between blocks dependency and go to Dataview -> edge, we can see the image on the left. As we can see on the image the dependency is caused by the variable `sum` in Jacobi and Gauss-Seidel.

Gauss-Seidel has one more dependency between the upper and left block, so it will execute by diagonals as we will see in the TDG Gauss-Seidel. It is because if we have a look on the execution in every point, it do

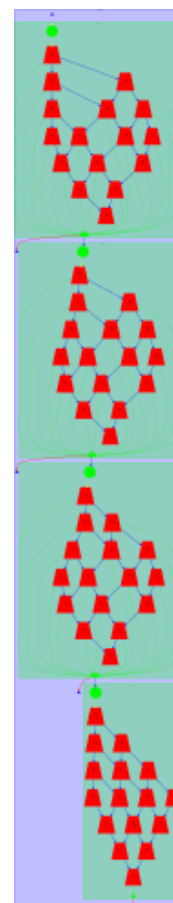
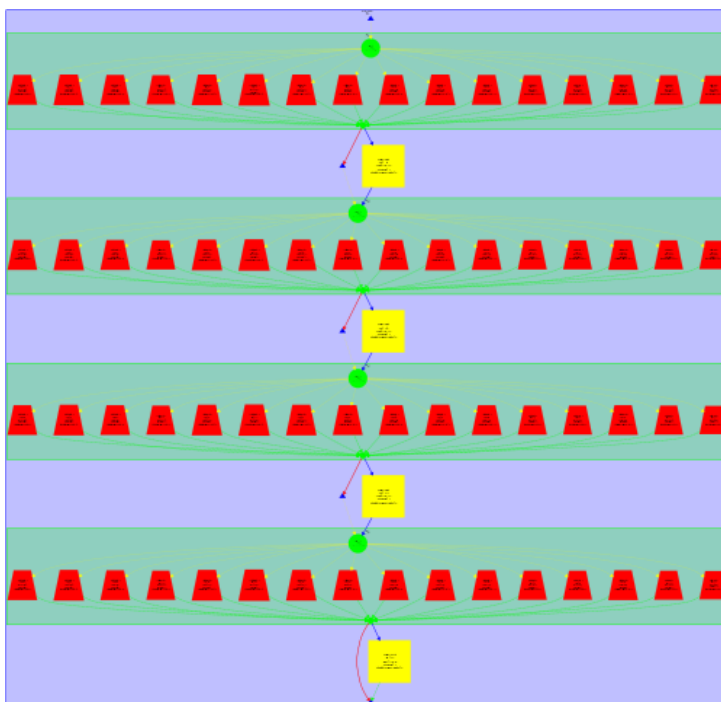
```
tmp = 0.25 * ( u[ i*sizey    + (j-1) ] + // left
              u[ i*sizey    + (j+1) ] + // right
              u[ (i-1)*sizey + j      ] + // top
              u[ (i+1)*sizey + j      ] ); // bottom
```

So the real dependencies are between the top and left block because sequentially if we have a block, it executes the left and top block before that block.

This doesn't happen in Jacobi because it has two matrix, one only read, so it won't be modified and the changes are in the other matrix.

2.2 Analysing parallelism

The nexts pictures are the TDG of Jacobi and Gauss-Seidel respectively:

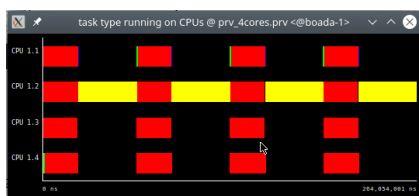


As we can see in the TDG shown before, Jacobi only has the sum dependency, so it will be more parallelizable, but it will use more memory and extra time to copy the matrix.

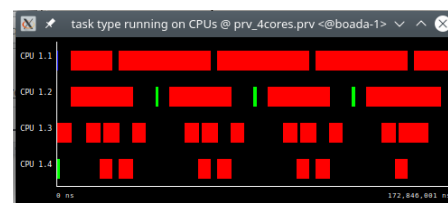
By the other hand, as we can see in the TDG shown before, Gauss-Seidel is less parallelizable (by diagonals) by the dependencies caused by the code because it uses only one matrix, but it won't have extra time to copy a matrix.

I can protect the sum variable that causes dependence using reduction in OpenMP. It will be better than using critical or an atomic because it internally will use less criticals or atomic than if we do with it.

Jacobi Simulation



Gauss-Seidel Simulation



As we can see in the simulation, with 4 threads, Jacobi is better than Gauss-Seidel because it has less execution time. It could be because it has more parallelizability, as we saw in TDG and it is better balanced (in the execution, we can see that threads 1, 3 and 4 have the same charge and thread 2 has more tasks because it has the copy mat too). If we have a look at Gauss-Seidel, it is unbalanced as we can see in the simulation. Thread 4 has a bit part of the tasks and if we reduce the thread, the charge of tasks increments.

3. Parallelization strategies

3.1 parallelization Jacobi

3.1.1 OpenMP parallelization Jacoby with serialization

To parallelize Jacoby, I modified the solver-omp.c by adding `reduction(+:sum)` and privatizing with `firstprivate` the variables `tmp` and `diff` like in the next line:

```
#pragma omp parallel reduction(+:sum) firstprivate(tmp, diff)
```

It is parallelized because if we execute `diff` with the image generated and the solution, it doesn't show any because produce the same image than the original as we can see at the screenshot of both image:

```
par3110@boada-1:~/lab5$ diff heat-jacobi.ppm solucio-jacobi.ppm
par3110@boada-1:~/lab5$
```

3.1.2 Jacoby without serialization

To improve the parallelization, I parallelized the tasks that compute the copy of the matrix by adding a `#pragma omp parallel` as we can see in the picture below.

```
#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];
}
}
```

I made a diff to know if it takes the same image than the original and, as we can see in the picture below, it does not print iny on the terminal and it implies that there are not differences and it implies that the parallelization is well implemented.

```
par3110@boada-1:~/lab5$ diff heat-jacobi.ppm solucio-jacobi.ppm
par3110@boada-1:~/lab5$
```

3.2 Gauss-Seidel parallelization

Gauss-Seidel has other different dependencies than Jacoby as I explained before, and to protect them I used the reduction for the variable sum and the firstprivate for the variable temp and diff as I said before. To respect the dependencies from the upper and left blocks I declared a matrix of blocks when if a block has a 0 it means that it hasn't been computed and if it is 1 it means that the block was computed. With that, I used a synchronization object to read if a block is computed and another to write a 1 in the matrix that I created that this block has been computed. The other dependency with sum and the variables diff and tmp I did the same as in Jacobi (I didn't modify it) and to don't modify the results when it is computed with Jacoby, i use an if(u == unew) { extra things for Gauss-Seidel} that condition in the if returns true if the computation of the heat diffusion wants to be computed with Gauss-Seidel and false if it wants to be computed with Jacobi.

I made a screenshot to the code and is the next:

```
int nBlocks[nblocksi][nblocksj]; |
if (u == unew) {
    //inicialitzo la matriu dels blocs a 0
    for (int i = 0; i < nblocksi; i++) {
        for (int j = 0; j < nblocksj; j++) {
            nBlocks[i][j] = 0;
        }
    }
}
```

With the last picture I created a matrix with the contents of the blocks (1 if it is computed and 0 if it has not been computed). At the beginning, the matrix will be all 0 because any block has been computed.

In the picture above, I add a synchronization with the do while constructs in every iteration of blockj to make the threads wait if his upper block has not been computed.

```
#pragma omp parallel reduction(+:sum) firstprivate(tmp, diff)// 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);
        if (u == unew && blocki > 0) {
            int sync;
            do { //si no es la primera fila
                #pragma omp atomic read
                sync = nBlocks[blocki - 1][blockj];
            } while (sync == 0); // mentre que el bloc de sobre no s'hagi computat, espera
        }
    }
}
```



```

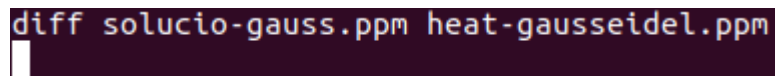
for (int i=max(1, i_start); i<=min(size_x-2, i_end); i++) {
    for (int j=max(1, j_start); j<=min(size_y-2, j_end); j++) {

        tmp = 0.25 * ( u[ i*size_y + (j-1) ] + // left
                      u[ i*size_y + (j+1) ] + // right
                      u[ (i-1)*size_y + j    ] + // top
                      u[ (i+1)*size_y + j    ] ); // bottom
        diff = tmp - u[i*size_y + j];
        sum += diff * diff;
        unew[i*size_y + j] = tmp;

    }
}
if (u == unew) {
    #pragma omp atomic write
    nBlocks[block_i][block_j] = 1;
}

```

Finally, I add an atomic write to change in the matrix of the computed blocks the 0 by 1 when the block is computed and make possible another block to be computed. If I do a diff, it doesn't show any difference as we can see in the right screenshot:



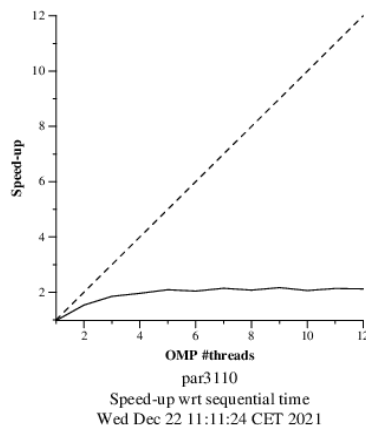
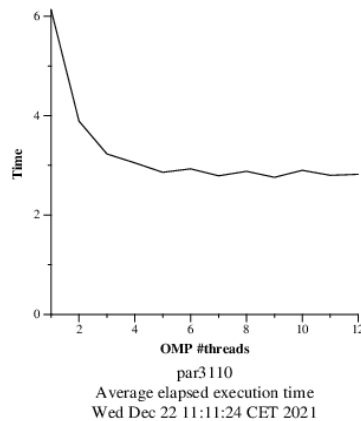
```
diff solucio-gauss.ppm heat-gausseidel.ppm
```

4. Performance evaluation

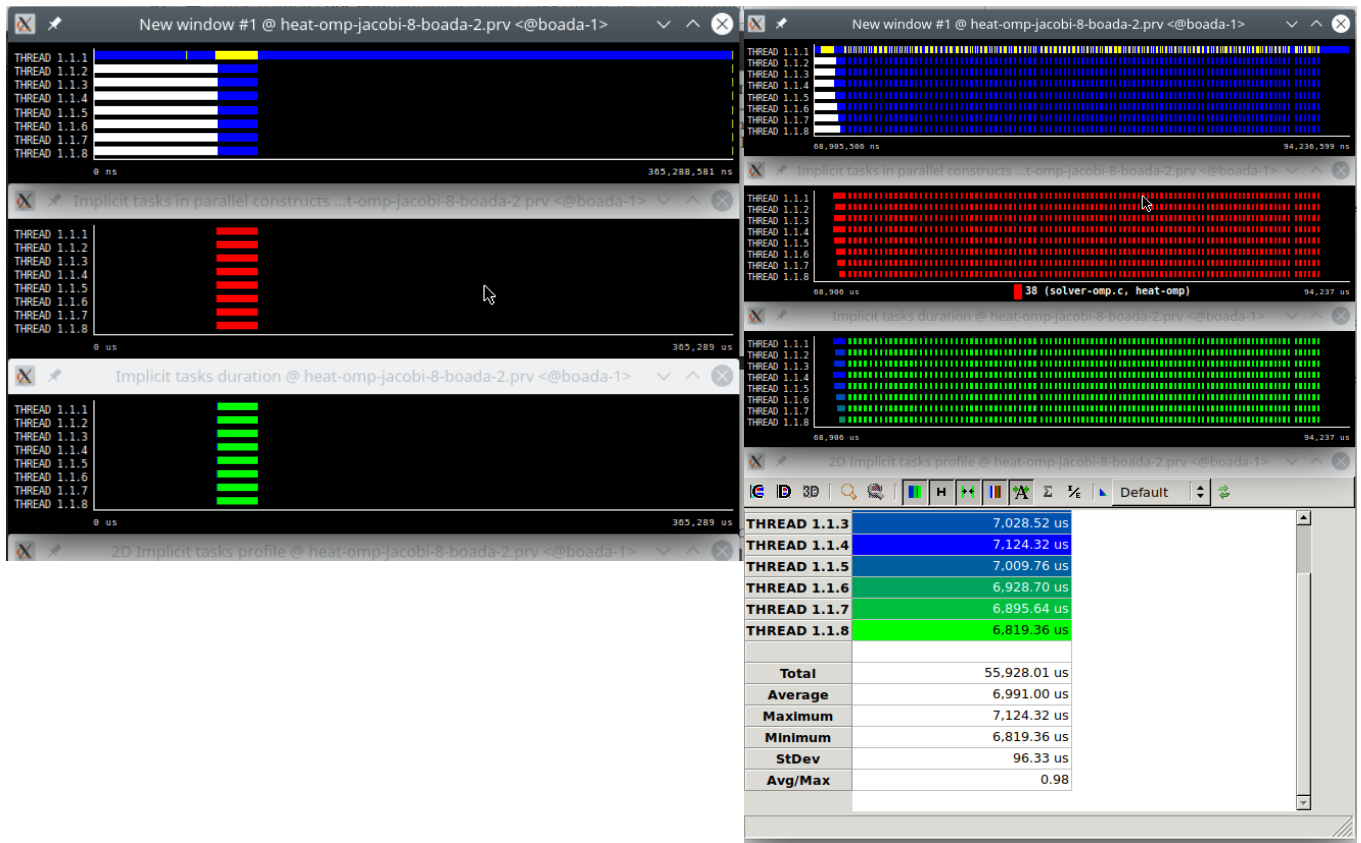
4.1 Performance evaluation of Jacoby

4.1.1 Performance evaluation of Jacoby with serialization

If we execute the command “sbatch submit-strong-omp.sh 0” and open the .ps file generated, we can see the next scalability plot:



As we can see in the scalability plot, the execution with 1 to 4 threads, the performance is better, but with 5 and more threads, is more or less the same. I think it could be because with 1 to 4 threads, it is balanced and with 4 threads achieve the best parallelization because at this moment, the tasks that are executing the computation of the color of the image takes less time than the tasks that are doing the copy of the matrix. The copy of the matrix is only done by one thread and the computation of the colour of the image is done by all the threads. For those reasons, the execution is unbalanced and it won't be a notable performance because we are doing the computation of the colours more parallelizable, but the copy of the matrix takes more time and it is not parallelized and in general, it will always be unbalanced and by the amdahl's law it never will have a significant performance with 5 or more threads.

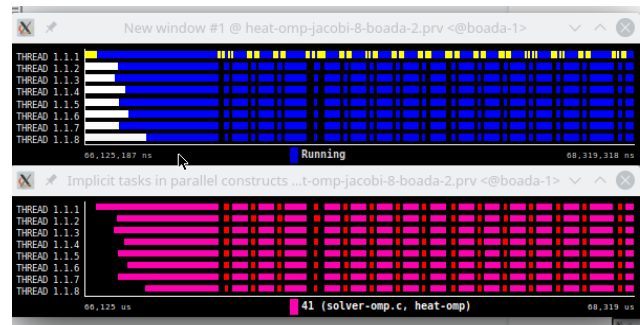
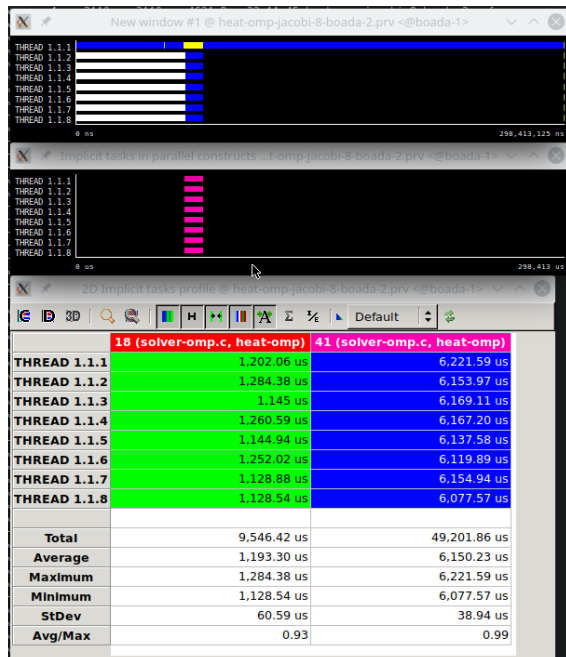


To make sure that is because it is unbalanced, I execute the program with the extrae and I obtain the image shown above. The first one is the execution and the 2 above are the time explicit tasks. On the right, we can see the same three pictures but with a zoom in the parallel part. The image above of the 3 on the right is the task profile option. As we can see in the taks profile, it is unbalanced because the values are with a difference of $7,124\text{ms} - 6,819\text{ms} = 0,305\text{ms}$ of difference.

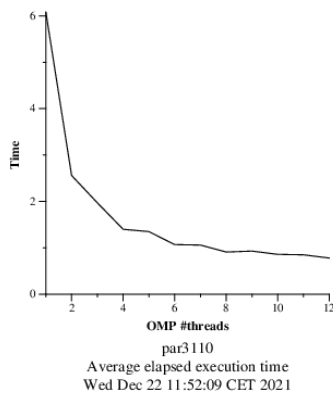
4.1.2 Performance evaluation of Jacoby without serialization

To see the timeline, I executed the submit-extrae.sh and I opened it with wxparaver. With the hints I obtained the pictures above.

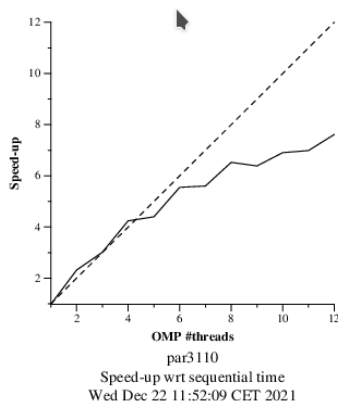
The first one is with all the execution and the 2 above are the time explicit tasks and the taks profile. On the right, we can see the same two pictures but with a zoom in the parallel part. The zoomed time explicit tasks (right above picture) show the tasks that are executing in the parallel execution and we can see that the pink colour are the tasks computation of the colour of the image and the red colour are the execution of the copy of the matrix.



Now, we have the copy matrix and the compute of the colours parallelized and we won't have more serialization parts except the other parts of the program, so we should have better performance because as we can see in the task profile, the tasks are balanced.



To see if we have better performance I execute the submit-strong.sh and it generates the plot on the left. As we can see, the scalability plot has a good performance when we increment the threads. It is because, as I said before, both parts (compute the copy of the matrix and the colour) are balanced and parallelized as we can see in the task profile and the execution timeline shown before.



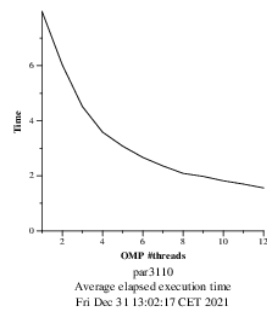
4.2 Performance evaluation of Gauss-Seidel

If I execute the `submit-extrae.sh`, I will obtain the first timeline and with the hints the next timeline and table:

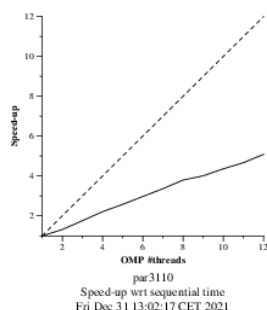


As we can see, the time spent in the parallel region is higher than in Jacobi and that can be due to synchronization.

With the `submit-strong-omp.sh` I obtained the next plot:

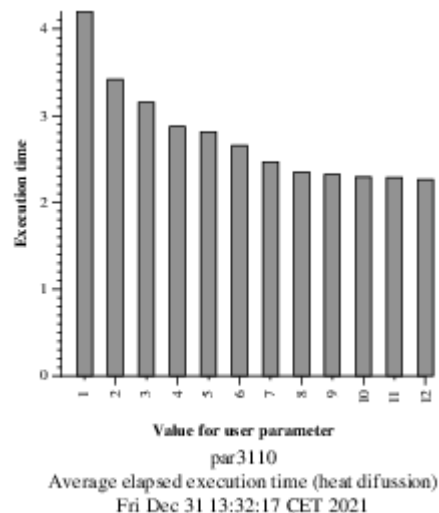
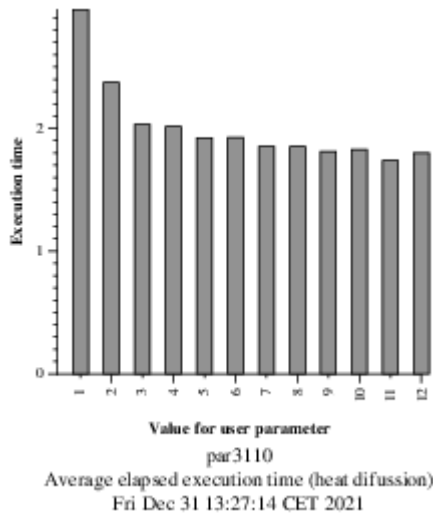


As we can see in the scalability plot, if we increase the number of threads, the time execution decreases and the scalability increases. I think this is because if we increase the number of threads, the matrix is more segmented by column (has more blocks) and it can be better parallelized.



If I change the number of columns by the userparam variable, as shown in the right image, and I executed the submit-userparam-omp,sh to know the execution time with other number of blockj and 8 and 4 threads respectively. The results are in the nexts plot:

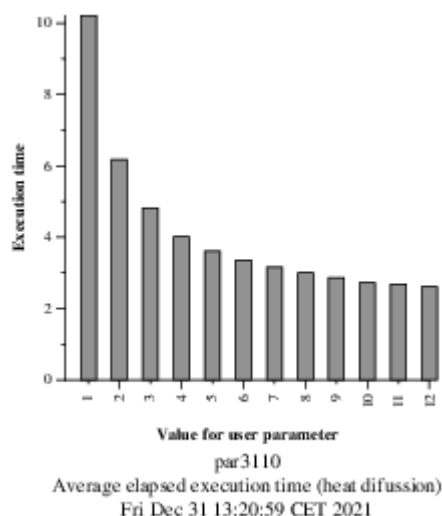
```
if(u == unew) nbblocksj = userparam * nbblocks;
```



As we can see in the plots, If we increment the userparam, and consequently, the number of blockj, his execution time decreases. I think it is normally because it has more blocks that can be done at the same time.

If we compare both plots, we can see that if we have less threads, the time execution will be higher with less threads than with more. It is normal because if we have more threads, the size of the blocks will be less and they will compute the block before.

If I execute that with blockj = userparam, I obtain the next plot with 8 threads:



As we can see, the execution time is higher than the last plot with 8 threads. It could be because the number of blockj are less and it is less parallelizable because we have to wait more time to execute a block and by this reason, the other threads will start later and it implies that the last thread will finish later by th dependencies.

5. Conclusions

The Jacobi strategy is a good way to parallelize the heat diffusion equation parallelizing the computation and the copy matrix because it has good scalability. Doing that strategy we will use more memory than with Gauss-Seidel. But with Gauss-Seidel we will need more time execution by the dependencies as we saw in the scalability plots. So choosing one strategy will depend if we want less time execution or less memory.