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

GUILLEM GRÀCIA ANDREU, GERARD MADRID MIRÓ

TARDOR 2021-22

PAR3109

1. Introduction

In this lab assignment, we will go through different parallelization methods applied to two different approaches to a heat diffusion calculation which show unique parallel behaviours. The different approaches are Jacobi and Gauss-Seidel's.

The main difference between Jacobi and Gauss-Seidel is that the former uses two matrixes whereas the latter only uses one, which is both read to and written to.

2. Sequential heat diffusion program and analysis with Tareador

2.1 Jacobi sequential execution

```
par3109@boada-1:~/lab5$ ./heat test.dat -a 0 -o heat-jacobi.ppm
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: 4.571
Flops and Flops per second: (11.182 GFlop => 2446.39 MFlop/s)
Convergence to residual=0.000050: 15756 iterations
```

Figure 1: Sequential execution Jacobi

In the figure above (Figure 1), we can see the output of the sequential execution of the Jacobi method which we will use for comparison through the whole assignment.

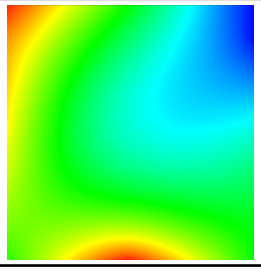

Figure 2: Visualization of sequential execution Jacobi

In Figure 2, we can clearly see the image generated by the Jacobi calculation. This image will be kept in order to check the correctness of later executions.

2.2 Gauss-Seidel sequential execution

```
par3109@boada-1:~/lab5$ ./heat test.dat -a 1 -o heat-gauss.ppm
Iterations        : 25000
Resolution        : 254
Residual          : 0.000050
Solver            : 1 (Gauss-Seidel)
Num. Heat sources : 2
   1: (0.00, 0.00) 1.00 2.50
   2: (0.50, 1.00) 1.00 2.50
Time: 6.015
Flops and Flops per second: (8.806 GFlop => 1464.06 MFlop/s)
Convergence to residual=0.000050: 12409 iterations
```

Figure 3: Sequential execution Gauss-Seidel

In the figure above (Figure 3) we can see the output of the sequential execution of the Gauss-Seidel method.

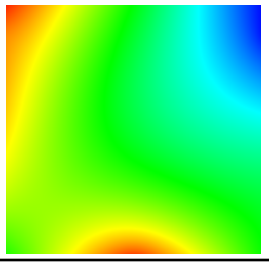

Figure 4: Visualization of sequential execution Gauss-Seidel

In Figure 4, we can clearly see the image generated by the Gauss-Seidel calculation. This image will be kept in order to check the correctness of later executions.

2.3 Analysis with Tareador

2.3.1 Jacobi in Tareador

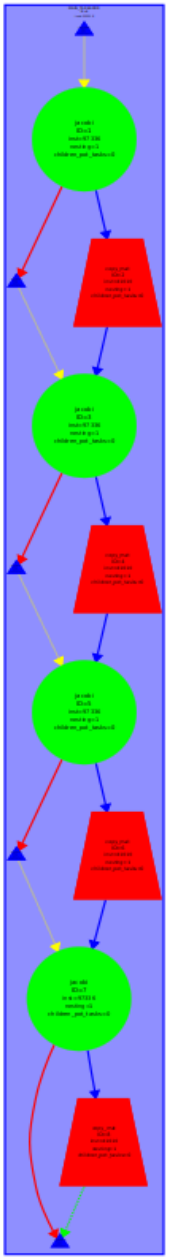

Figure 5: Initial Jacobi tareador tasks

In Figure 5 we can see the first version of the Jacobi method tasks with tareador. The image shows how the program is executed in a totally sequential way. This can be seen by checking the dependencies in the image.

2.3.2 Gauss-Seidel in Tareador

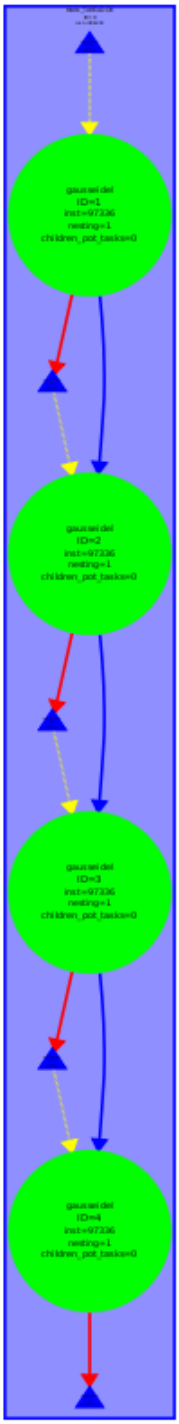

Figure 6: Initial Gauss-Seidel tareador tasks

In Figure 6 we can see the first version of the Gauss-Seidel method tasks with tareador. The image shows how the program is executed in a totally sequential way as well as the image shown in Figure 5. The sequentiality can also be seen by checking the dependencies in the image.

2.3.3 Task per block with Jacobi

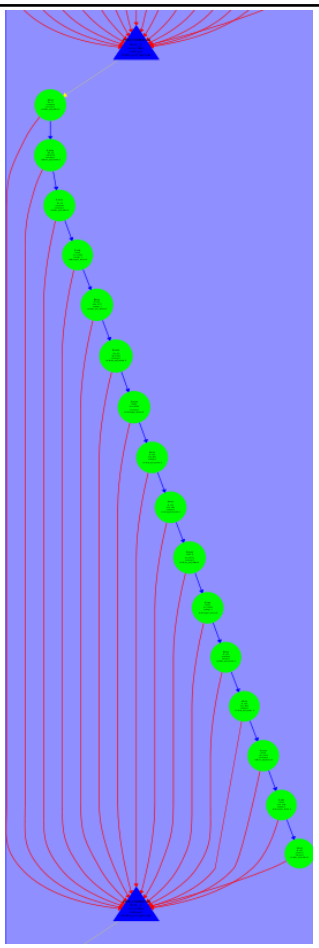

Figure 7: Task per block Jacobi tareador Tasks

This time, our goal was to parallelize the Jacobi version with a Task per each block. However, even though the image (Figure 7) is different, the parallelization is still non-existent, so we have to keep improving it in order to stray away from sequentiality.

2.3.3 Task per block with Gauss-Seidel

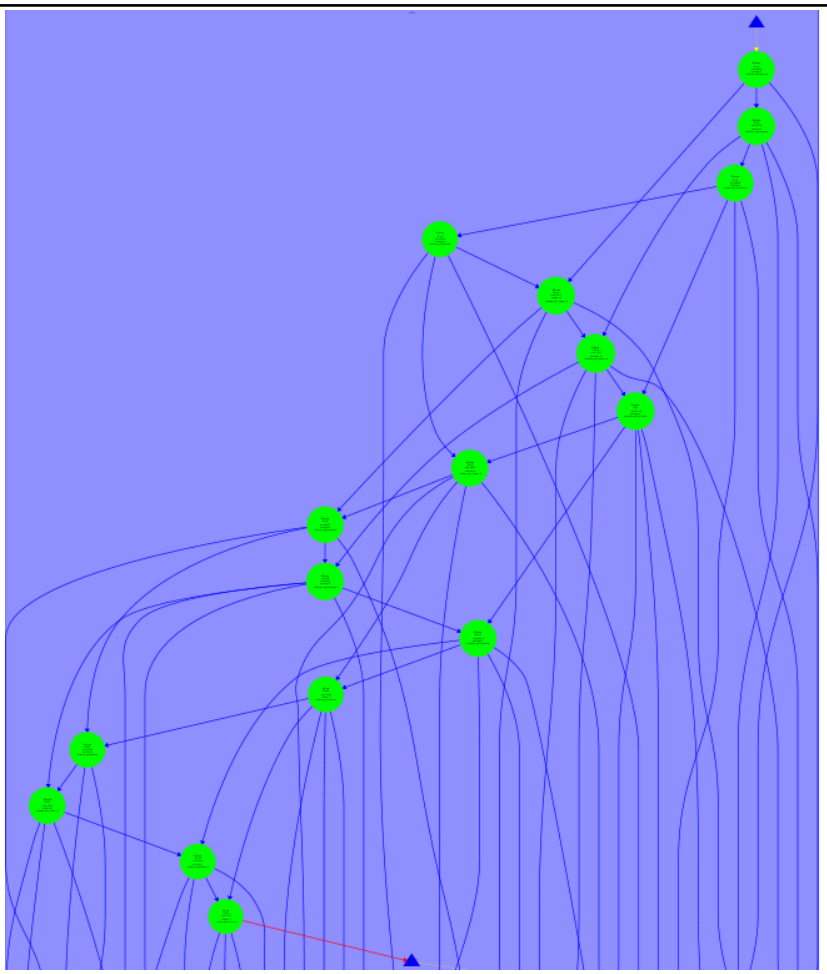

Figure 8: Task per block Gauss-Seidel tareador Tasks

This time, our goal was to parallelize the Gauss-Seidel version with a Task per each block. However, even though the image (Figure 8) is different and quite amazing, the parallelization is still non-existent, so we have to keep improving it in order to stray away from sequentiality.

What we can see in the image above is that the matrix is divided into 16 nodes. The dependencies among these nodes are quite clear: The first four nodes only have one dependency, since they only depend on the node on the left and have no nodes above them. The following 12 nodes depend on the one above and the one on the left when looked at on the matrix.

2.3.4 Disabled objects with Jacobi

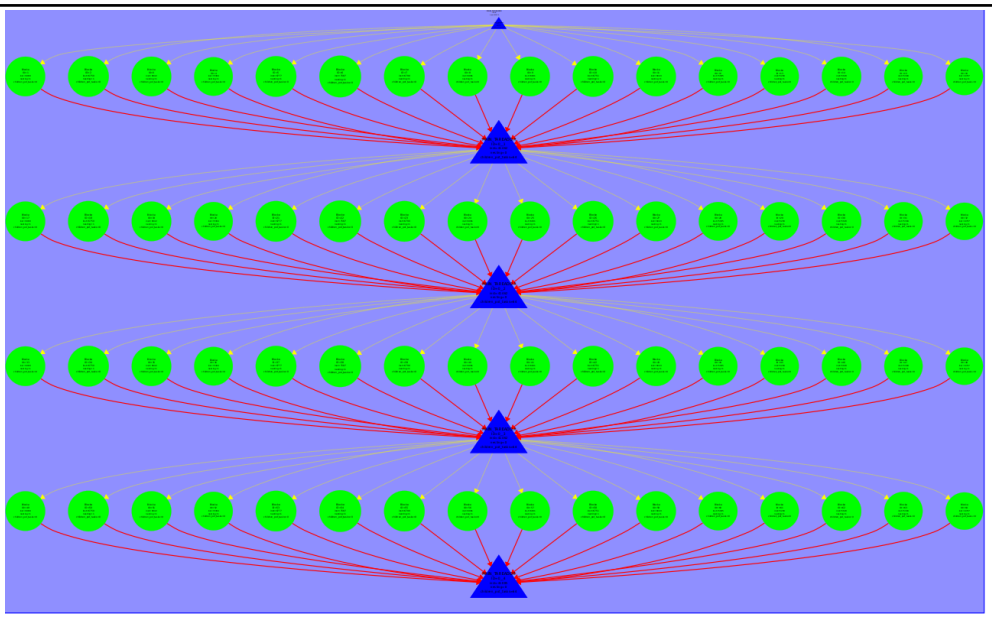

Figure 9: Disabled objects Jacobi tareador tasks

Finally we tried to use the command `tareador_disable_object` and `tareador_enable_object` in the Jacobi version to see the effect of protecting the dependencies of each task and now we can see (in Figure 9) how the parallelism is really good. We think that the best option of protection is to use the reduction method with the sum variable and private for the other variables.

2.3.5 Disabled objects with Gauss-Seidel

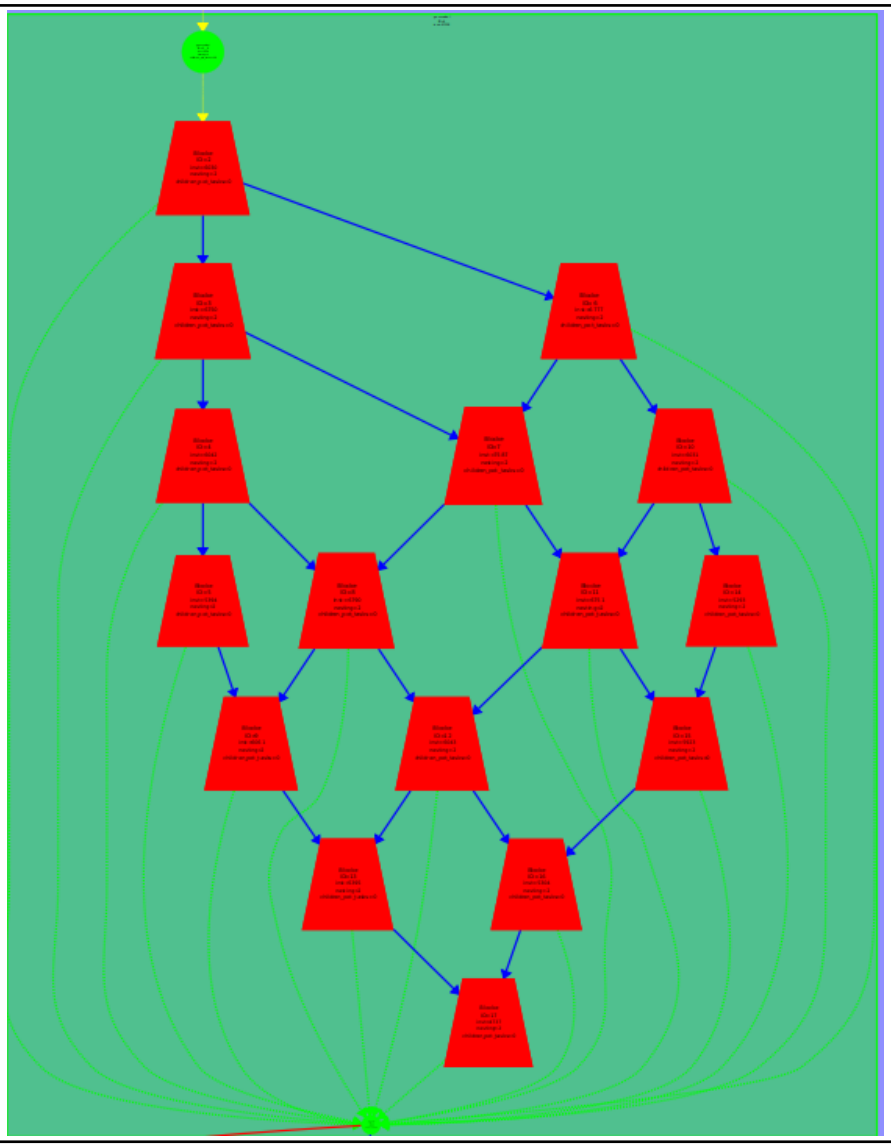

Figure 10: Disabled objects Gauss-Seidel tareador tasks

We also used the commands to see the effect of protection of the dependencies we used before in the Jacobi version and we can clearly see parallelization here as well. It's not as obvious as the Jacobi version, but we can see (in Figure 10) the pattern that we could identify before: Each matrix cell depends on the cell above and the cell on its left.
So we can see a parallelization that follows the diagonals of the Matrix with the following parallelization pattern: 1-2-3-4-3-2-1 (seen from top to bottom).

2.3.6 Jacobi Simulation

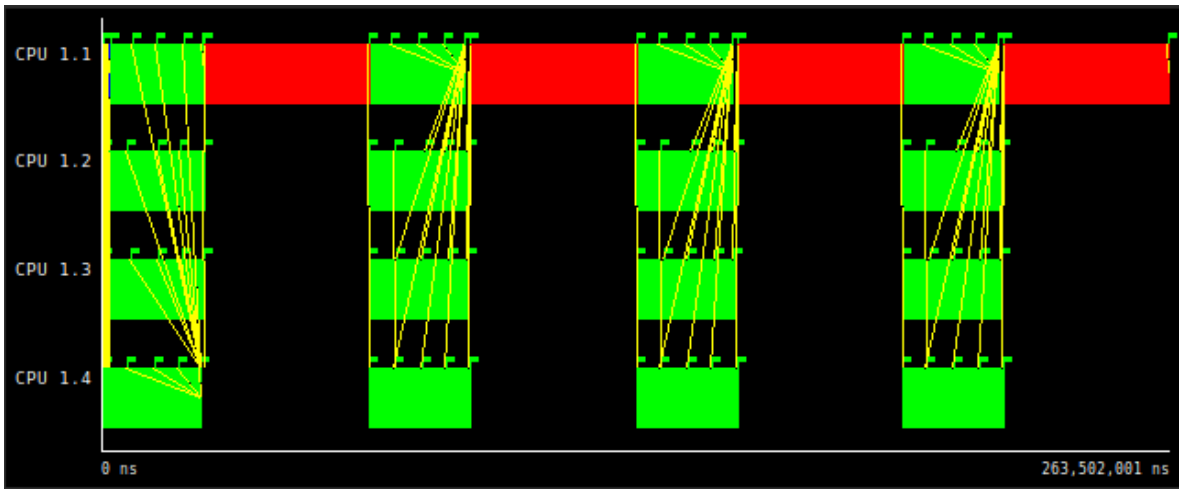

Figure 11: 4 processor Jacobi simulation

Figure 11 shows us an execution simulation of the Jacobi version and we can clearly see that all the tasks of a block can be executed in parallel as we predicted before with the task dependence graph.

2.3.7 Gauss-Seidel Simulation

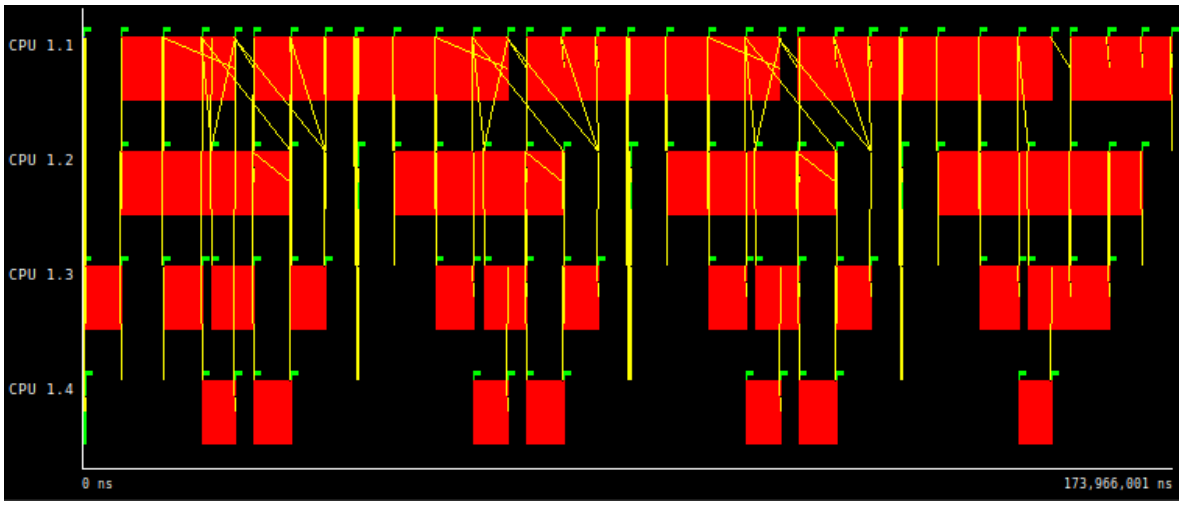

Figure 12: 4 processor Gauss-Seidel simulation

On the other hand, when taking a look at the Gauss-Seidel simulation shown in Figure 12, we can see how the same structure of 1-2-3-4-3-2-1 is applied. Although it is harder to see, now the structure is seen from left to right vertically: Processor 1.3 executes the first, processors 1.1 and 1.2 execute the second layer, then 1.1,1.2 and 1.3 execute the third layer, and so on.

2.3.8 More parallelization improvements

When asked if anything else could be done to improve the execution time of these calculations and improve overall performance, we thought of checking the remaining part of the code that was still sequential: Matrix copying.

When adding a parallel pragma to the code and removing the first for loop, we managed to get the following image (Figure 13):

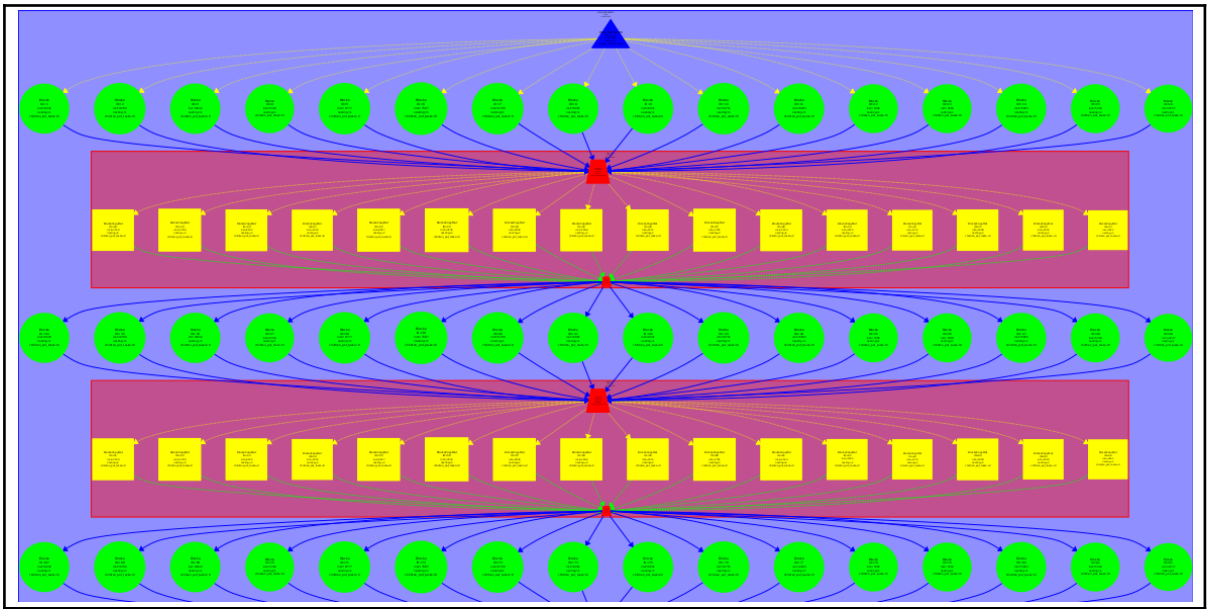

Figure 13: Jacobi with copy mat parallelization tareador tasks

In this image (Figure 13), we can see how the Jacobi TDG improved and the matrix copying was now done in parallel instead of sequentially. There is no need to check the difference in Gauss-Seidel since it doesn't use this part of the code and would remain the same.

3. Parallelisation of the heat equation solvers

3.1 Jacobi version

After understanding the solver-omp.c code, we completed the code for it to work with Jacobi.

```
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 reduction(+:sum) private(diff,tmp)
   {
     int blocki = omp_get_thread_num();
     int i_start = lowerb(blocki, nblocksi, sizex);
     int i_end = upperb(blocki, nblocksi, sizex);
     ...
   }
   return sum;
}
```

Figure 14: Jacobi implementation code fragment

In order to do the Jacobi implementation, we changed the first loop to:

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

This is done to protect the variables. "`sum`" has to be a reduction of type (+) because we want to add the result of all the threads at the end of the execution of each one but "`diff`" and "`tmp`" are only temporary variables so it is more logical to simply privatize them.

The "`nblocksi`" will then be the max thread number to divide the matrix in blocks in order to execute each block in parallel.

Finally, "`blocki`" will get the value of the thread id.

```
par3109@boada-1:~/lab5$ diff heat-jacobi.ppm heat-jacobi-seq.ppm
par3109@boada-1:~/lab5$
```

Figure 15: Jacobi parallelization comprobation

After compiling and checking the image generated manually, we found no differences. However, we needed a more thorough check in order to be completely sure of the correctness of the code. That's why we used the diff command with the newly obtained image and the one we saved on the first step of this assignment. As we can see in Figure 15, the diff didn't return anything, meaning that both images were equal.

```
par3109@boada-1:~/lab5$ cat heat-omp-jacobi-8-boada-2.txt
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: 3.001
Flops and Flops per second: (11.182 GFlop => 3725.52 MFlop/s)
Convergence to residual=0.000050: 15756 iterations
```

Figure 16: Jacobi parallelization time duration

In order to see how much the new code improved the parallelization, we submitted it to boada and checked the execution time obtained. In Figure 16 we can see that the time is roughly 3 seconds, which sets us on a better path than the previous 4.7 seconds obtained when executed sequentially.

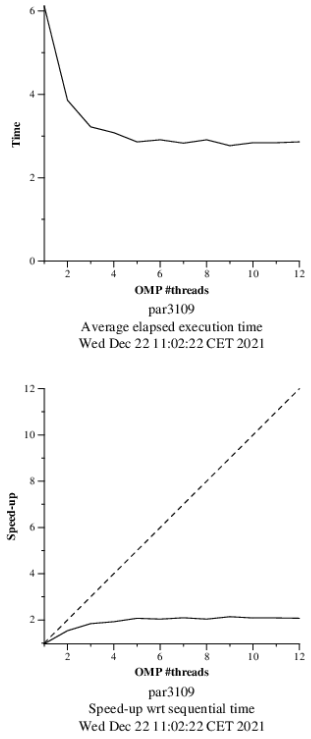

Figure 17: Jacobi parallelization strong scalability

As we can see in Figure 17, the speed-up is not looking good at all, since it doesn't go up nor follows the x=y reference line. Having seen this, we can be sure that changes need to be made.

3.2 Improved Jacobi version

Next, after having understood the jacobi version, we can see that improvement can be applied in order to get even better results. To do that, we followed the same strategy that we applied before: Parallelizing the matrix copying section; since it was the only thing that was not fully optimized.

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

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

   #pragma omp parallel // 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);
     ...
   }
}
```

Figure 18: Improved Jacobi implementation code fragment

We figured out that the function "`copy_mat`" could be parallelized too and to do that we basically copy the same schema that we did in the solve code (can be seen in Figure 14) but adapted it to the copy_mat function as we can see in Figure 18.

```
par3109@boada-1:~/lab5$ diff heat-jacobi.ppm heat-jacobi-seq.ppm
par3109@boada-1:~/lab5$
```

Figure 19: Jacobi improvement dif

After compiling and checking the image generated manually, we found no differences. Once again, we used the diff command to get a more thorough check in order to be completely sure of the correctness of the code. We can see in Figure 19 that everything was OK.

```
par3109@boada-1:~/lab5$ cat heat-omp-jacobi-8-boada-2.txt
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: 0.729
Flops and Flops per second: (11.182 GFlop => 15332.88 MFlop/s)
Convergence to residual=0.000050: 15756 iterations
```

Figure 20: Jacobi improvement time

In order to see how much this change improved the parallelization, we submitted it to boada and checked the execution time obtained. In Figure 20 we can see that now the time obtained is less than a second, which is a high improvement coming from the 3 seconds obtained before applying these changes.

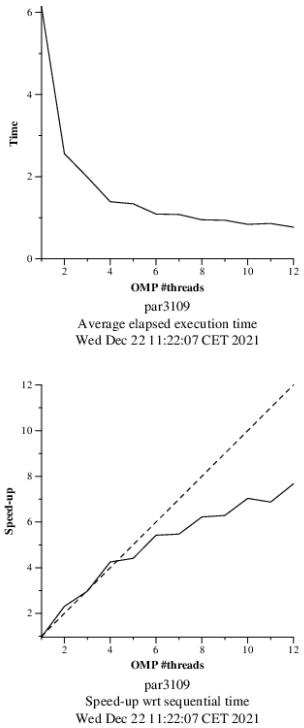

Figure 21: Jacobi improvement strong scalability

Finally, as we can see in Figure 21, the speed-up is now looking way better than before. It is now closer to the x=y reference line and is not as logarithmic as we saw in the previous version.

3.3 Gauss-Seidel version

This time, we had to parallelize the solver for the Gauss-Seidel. We used the Jacobi code as a starting point to create the Gauss-Seidel code.
We had to make a custom synchronisation object in order to synchronise correctly all the tasks using the pattern of the parallelization of matrix diagonals.

```
double solve (double *u, double *unew, unsigned sizex, unsigned
sizey) {
   int nblocksi=omp_get_max_threads();
   int nblocksj=userparam;
   int next[nblocksi];
   for(int i = 0; i < nblocksi; ++i) {
     next[i] = 0;
   }
   #pragma omp parallel reduction(+:sum) private(diff,tmp)
   {
     int temp = 0;
     ...
     for (int blockj=0; blockj<nblocksj; ++blockj) {
       ...
       if(u == unew && blocki != 0) {
         do {
           #pragma omp atomic read
           temp = next[blocki-1];
         } while(temp <= blockj);
       }
       for (int i=max(1, i_start); i<=min(sizex-2, i_end); i++) {
         ...
       }
       if (u == unew) {
         #pragma omp atomic write
         next[blocki] = next[blocki]+1;
       }
     }
   }
   return sum;
}
```

Figure 22: Gauss-Seidel implementation code fragment

In order to do that, as we can see in Figure 22, we added a vector with the max threads value elements which enables the program to set each row to a single

thread. This allows the threads to work in a more efficient way by waiting the minimum time without needing to have a matrix as our main data structure since every time that we move on to a new "Jblock", the thread that follows will be notified to start.

We added our custom synchronisation object before the "i" loop.
Also, we assigned "`nblocksj`" to "`userparam`" in order to be able yo execute correctly the script `submit-userparam.sh`

```
par3109@boada-1:~/lab5$ diff heat-gausseidel.ppm heat-gauss-seq.ppm
par3109@boada-1:~/lab5$
```

Figure 23: Gauss-Seidel diff with the original output image

As we can see in Figure 23, the diff command returns nothing, and we have established before that means that both the newly obtained and the saved images are equal, so everything is OK.

```
par3109@boada-1:~/lab5$ cat heat-omp-gausseidel-8-boada-2.txt
Iterations        : 25000
Resolution        : 254
Residual          : 0.000050
Solver            : 1 (Gauss-Seidel)
Num. Heat sources : 2
   1: (0.00, 0.00) 1.00 2.50
   2: (0.50, 1.00) 1.00 2.50
Time: 1.588
Flops and Flops per second: (8.804 GFlop => 5542.28 MFlop/s)
Convergence to residual=0.000050: 12405 iterations
```

Figure 24: Gauss-Seidel parallelization time

In order to see how much these changes improved the performance, we submitted it to boada and checked the execution time obtained. In Figure 24 we can see that now the time obtained, which is always around 1.5 seconds. Comparing this time with the first one obtained (sequential code), which was 6 seconds, gives us quite the speed-up. This speed-up can be checked with the strong scalability plot.

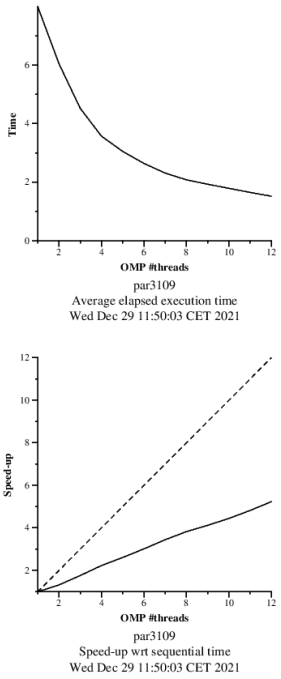

Figure 25: Gauss-Seidel parallelization strong scalability

Now, we can see (in Figure 25) the speed-up in a more understandable way. Even though the speed-up doesn't seem as much, it has a reasonable improvement compared to a fully sequential approach.

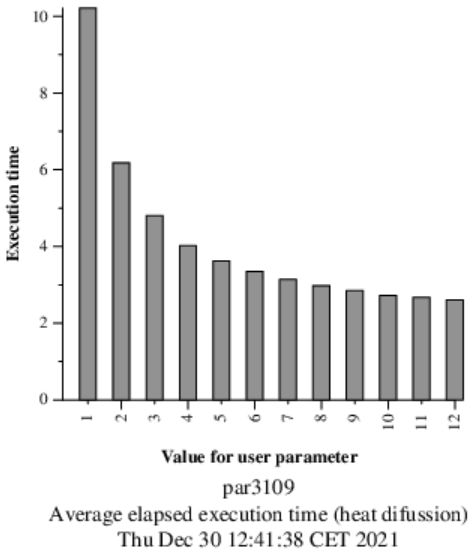

Figure 26: Gauss-Seidel with different nblocksj values

As we can see in Figure 26 above, increasing the nblocksj value reduces the granularity of the block and the execution time too and it seems to be asymptotic, so we can predict that the best value j is found at 12 and onwards.

4. Conclusions

After finishing this laboratory assignment, and it being our last, we can say that we have learned a lot about parallelizing and how to properly analyse different images and plots provided by the programs we have worked with. Even though the final part of this laboratory has brought many headaches, we managed to work it through and get a proper result that matched our expectations.

To sum it all up, we are very happy with the work we have done as well as with what we have learned both in this assignment and in the whole laboratory part of this subject.