PAR Laboratory Assignment

Lab 5: Geometric (data) decomposition: heat diffusion equation

Sandra Flores Hidalgo

Celia Fernández Pérez

par1215

**Index**

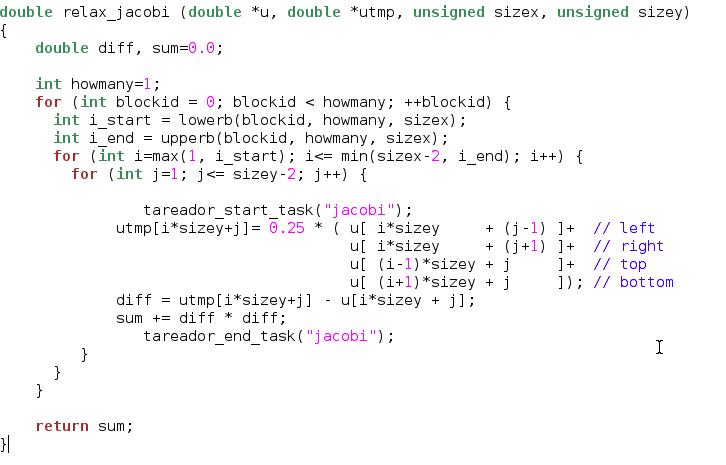
1. Analysis with Tareador

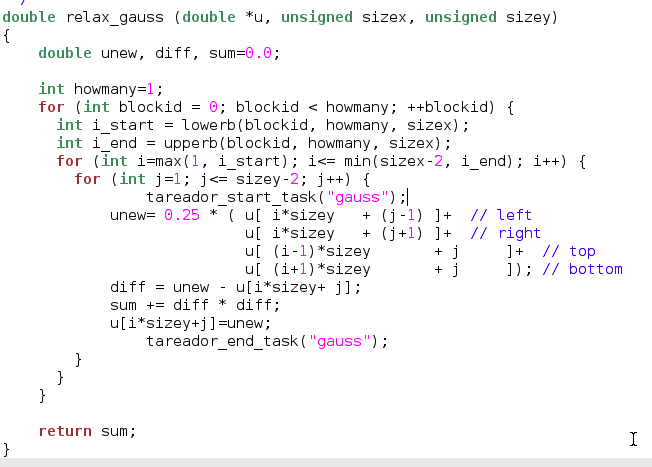
2. Parallelization of Jacobi with OpenMP parallel

3. Parallelization of Gauss-Seidel with OpenMP ordered

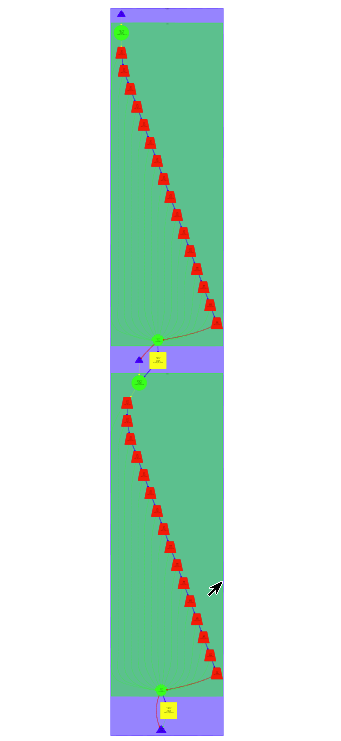
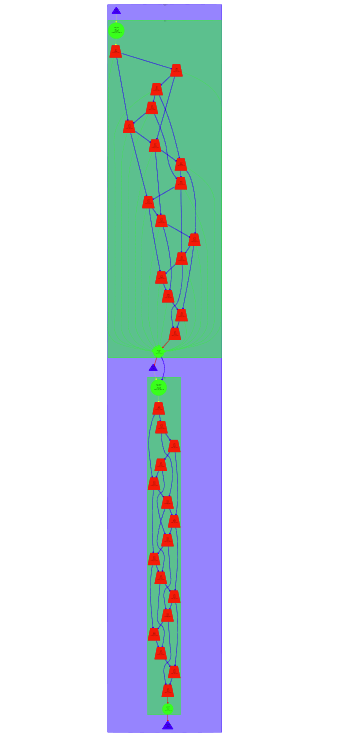
**1. Analysis with Tareador**

To analyze the code with the tareador we add the tareador clauses, tareador\_strat\_task and tareador\_end\_task, in the internal loop. The corresponding code is:

[[1]](#footnote-0)

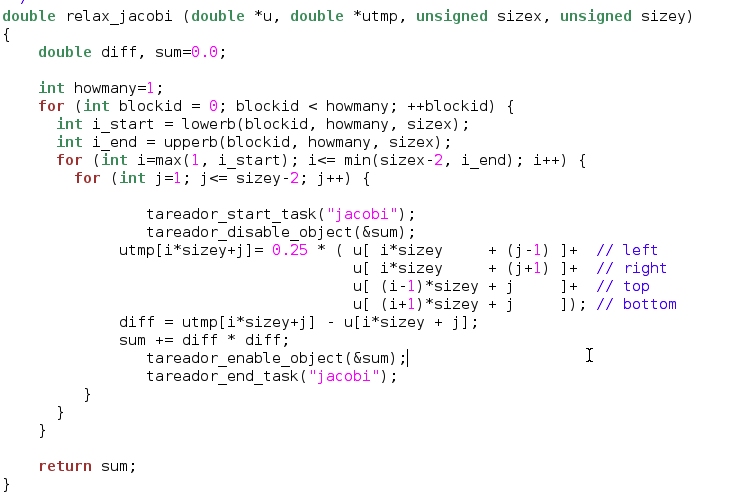
[[2]](#footnote-1)

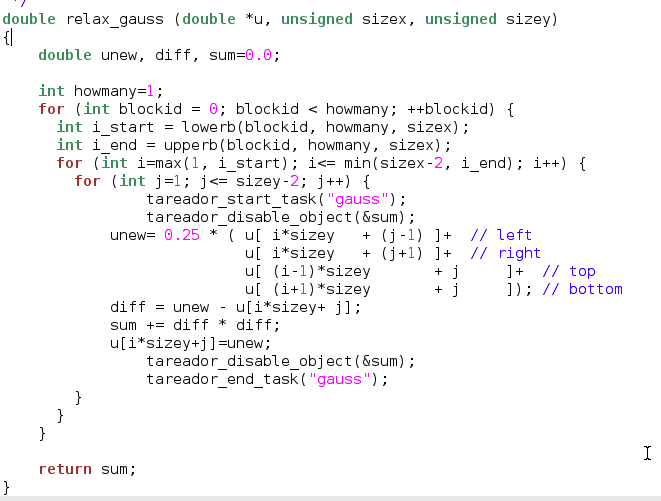
The results obtained are the graphics shown below for both Jacobi and Gauss cases.These end and start with tareador clauses, first jacobi and second gauss. By right-clicking on any of the tasks we can analize de dependences on the Dataview. We realize that the variable sum is causing the dependences.

[[3]](#footnote-2)[[4]](#footnote-3)

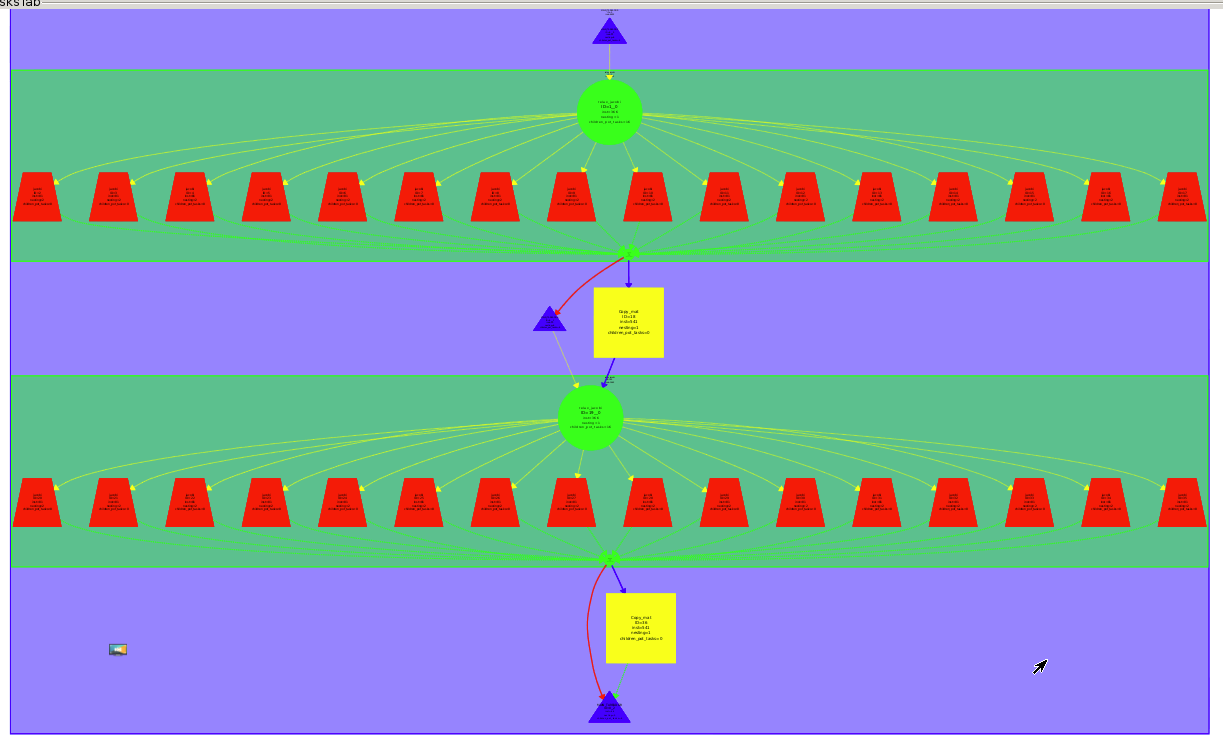
To avoid the serialization of the tasks we disable and enable the variable by adding ‘tareador\_disable\_object(&sum)’ and ‘tareador\_disable\_object(&sum)’, in both cases.

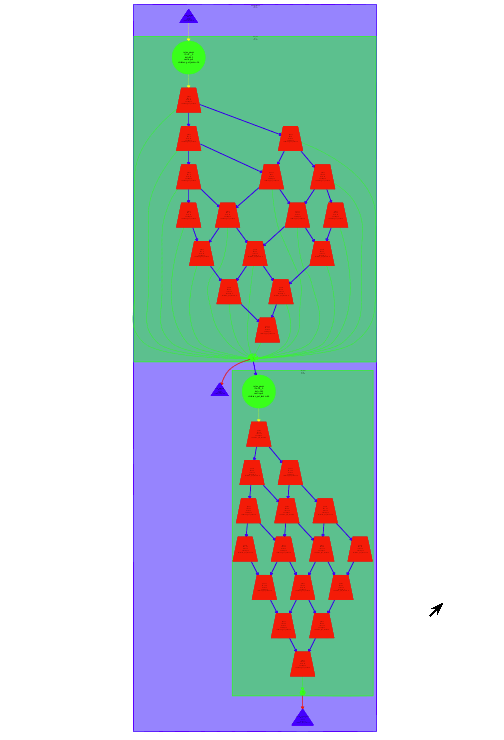
The code of this is:

[[5]](#footnote-4)

[[6]](#footnote-5)

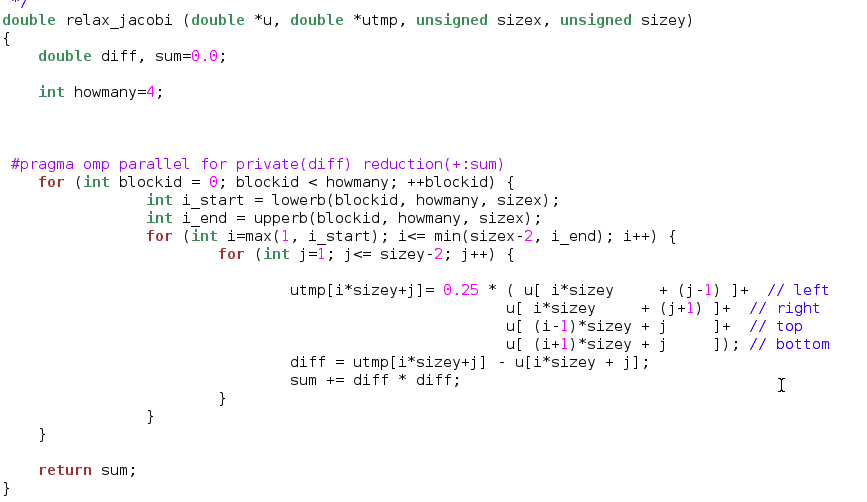
As we can appreciate on the graphics, in the first case, the Jacobi’s implementation, the dependences can be erased by disabling the variable sum. In the second case, the Gauss one, we can appreciate an increasing on the parallelism, but the dependencies are not eliminated. This is because the u matrix is been written and read while, in the other case it is only been read and then a temporal matrix utmp is used to write on it.

[[7]](#footnote-6)

[[8]](#footnote-7)

**2. Parallelization of Jacobi with OpenMP parallel**

To achieve the parallelization of the jacobi function we added a parallel statement to the external loop of the code, as we see in the Image 9.

[[9]](#footnote-8)

Based on the results showed on the tareador it has been distributed the computation to a block level distribution, so each of the processors could compute an entire block. This is achieved with the OpenMP parallel for, just before the external for-loop. To work with this statement we have to privatize the diff variable to each thread. Also, we must transform the sum variable into a local variable to each processor by using the reduction clause. Then a summatory operation is performed over the obtained results of the sum variable in each one of the blocks.

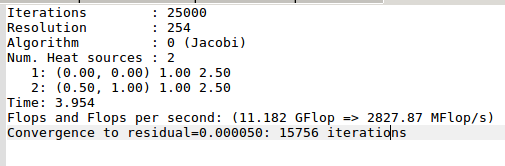
By using this parallelization we obtain the same draw as before, as we can see in Image 10, but with optimized results. To prove it we can compare the results represented in Image 11 and Image 12.

To analyze the new parallel function we use the Paraver. The traces obtained are executed by the program getting the images below. As we can see in Image 13 and Image 16 the main thread, thread 1.1.1, starts to run. When it gets to the parallelized fragment it starts a scheduling and fork-join fase, creating all the other threads. Then the rest starts to run until the parallel construction finishes. At that point the main thread starts to run again while the others stay in a idle state.

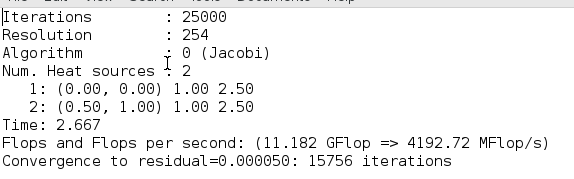
The images 14 and 15 allows us to appreciate the successive signals for the entry and exit points of the different OpenMP activities.

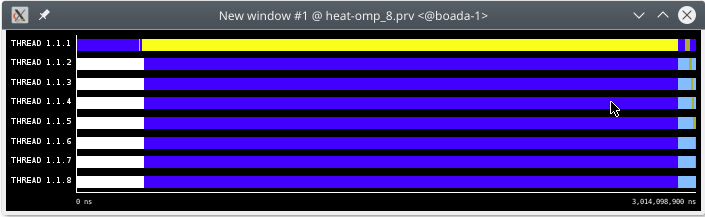
[[10]](#footnote-9)

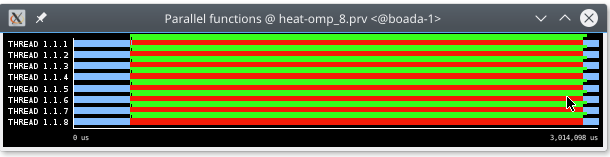
The initial version results:

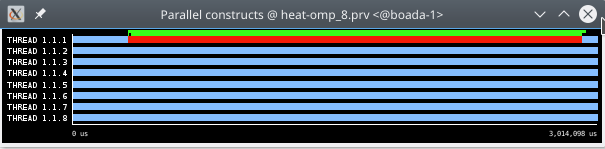
[[11]](#footnote-10)

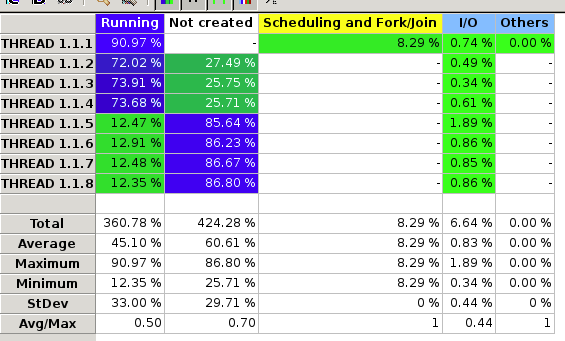
The new version results:

[[12]](#footnote-11)

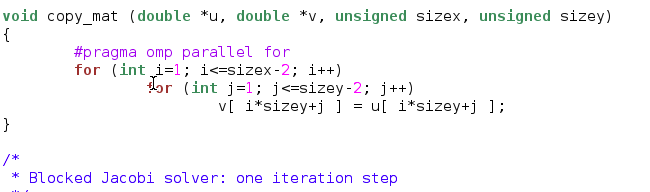
[[13]](#footnote-12)

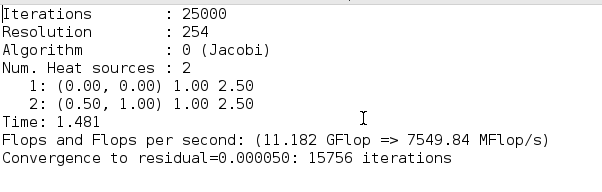
**[[14]](#footnote-13)**

**[[15]](#footnote-14)**

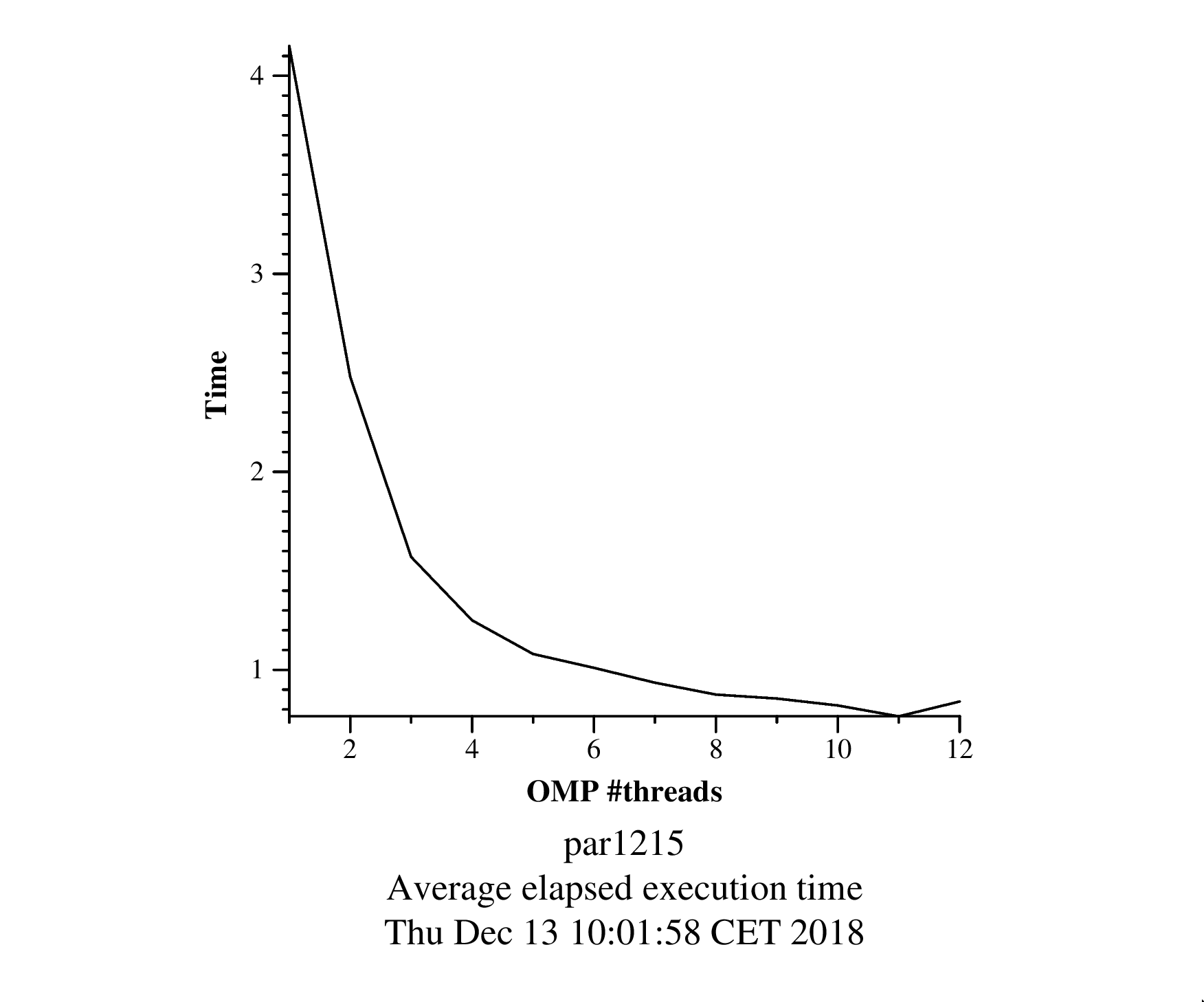
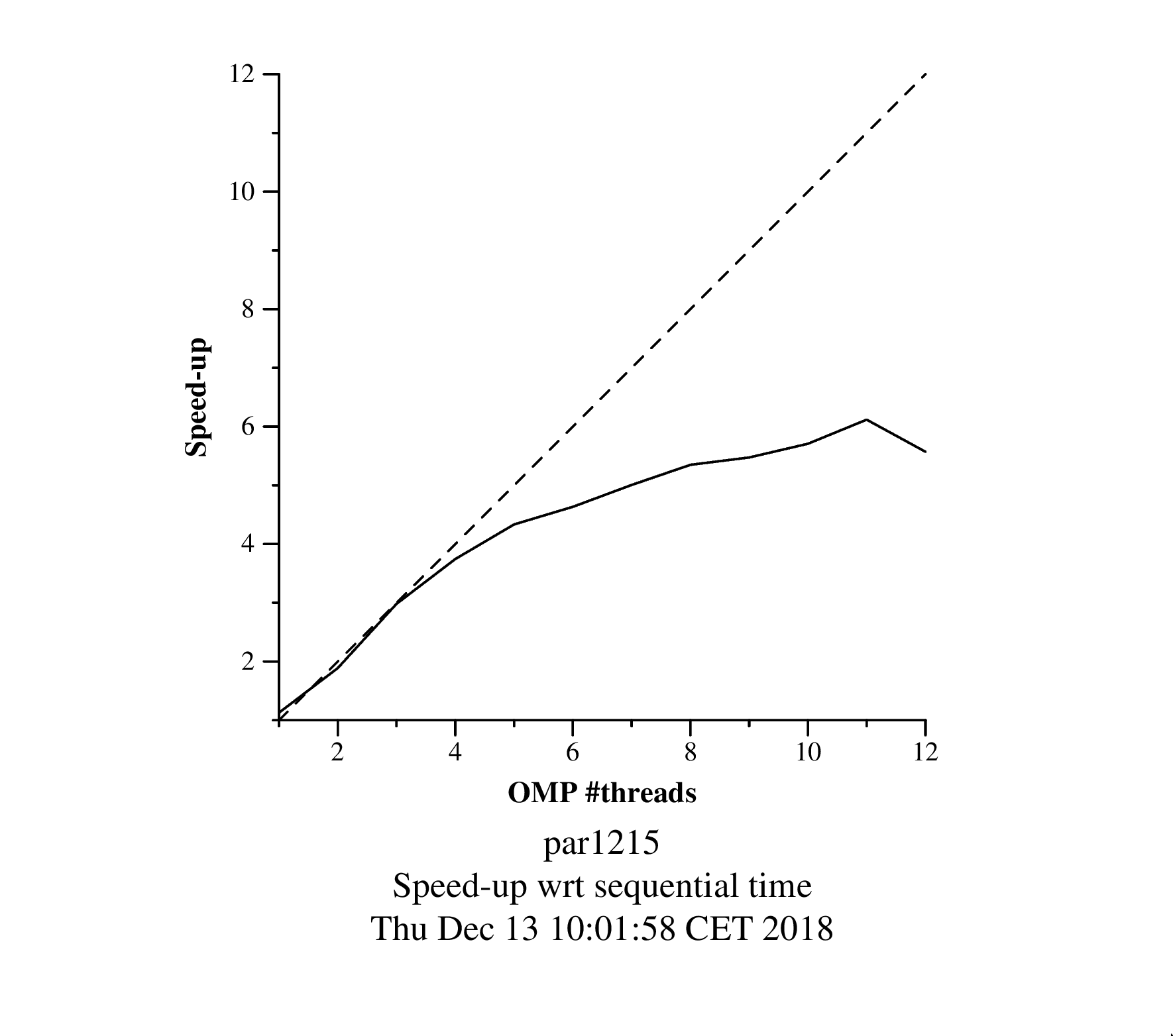
**[[16]](#footnote-15)**

To get even better results we decide to apply another parallel for statement at the copy\_mat function. With these change the results are better, as we can see at the image.

[[17]](#footnote-16)

[[18]](#footnote-17)

Until four threads it accelerates on a linear way. Above this number the speed up decreases because there are not enough tasks to feed the threads, because, as we see in the code, we are dividing it in four blocks with the variable howmany.

[[19]](#footnote-18)[[20]](#footnote-19)

**4.Parallelization of Gauss-Seidel with OpenMP ordered**

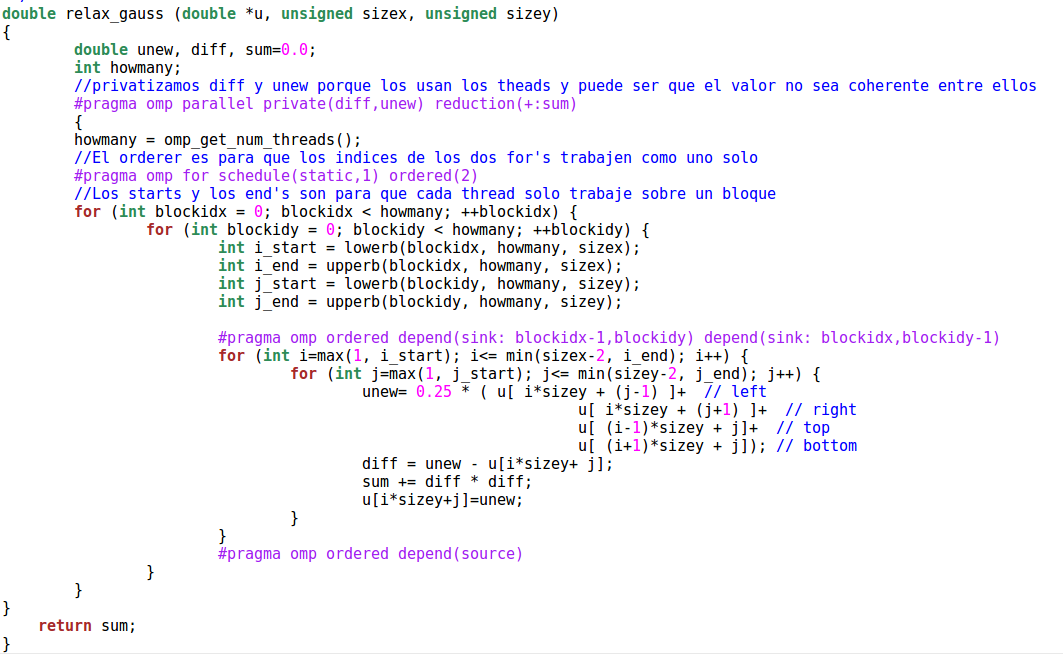
To parallelize the gauss function we had to add multiple pragma omp statements, as we can appreciate in the code from Image 21.

The schedule(static, 1) distribute the blocks among the threads one at a time, so blocks that are in diagonal position between them are always processed by differents threads. This way, they can be executed in parallel getting an optimal parallelization.

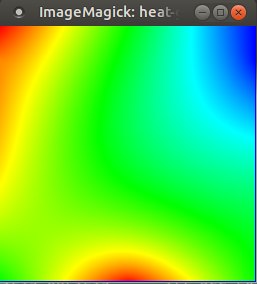
The depend clauses assure that when a block is been processed its upper and left neighbours had been already computed, because they are dependent.

The private(diff, unew) clause make sure that there are not interference between the differents values from the diff and unew variables on each one of the threads, so each process has its own private variables.

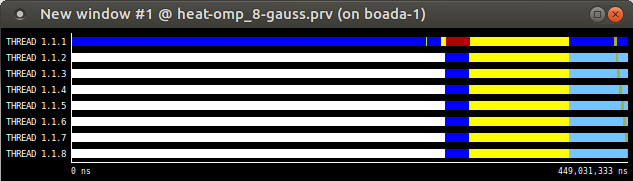
Finally the reduction(+:sum) update the sum value through the values generated on each process.

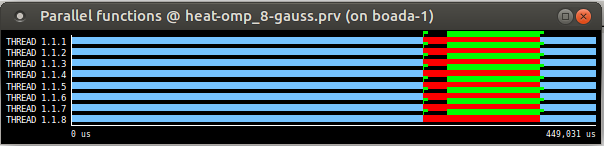
[[21]](#footnote-20)

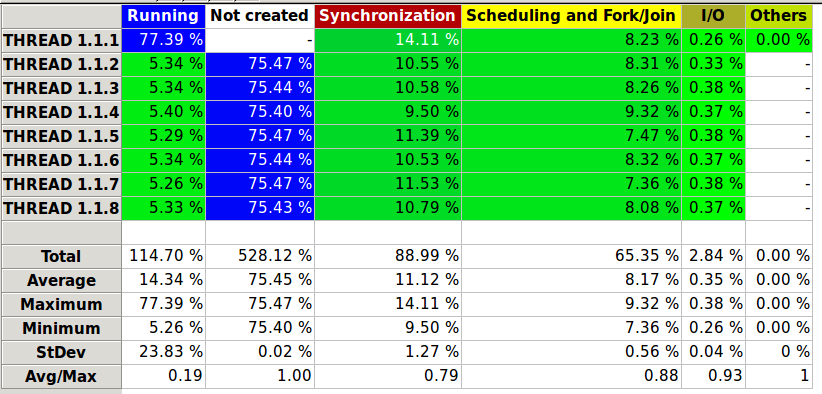
In this case, after applying the parallel code to the gauss function the result equals to the image obtained with the original code, as we can see in the Image 22.

 [[22]](#footnote-21)

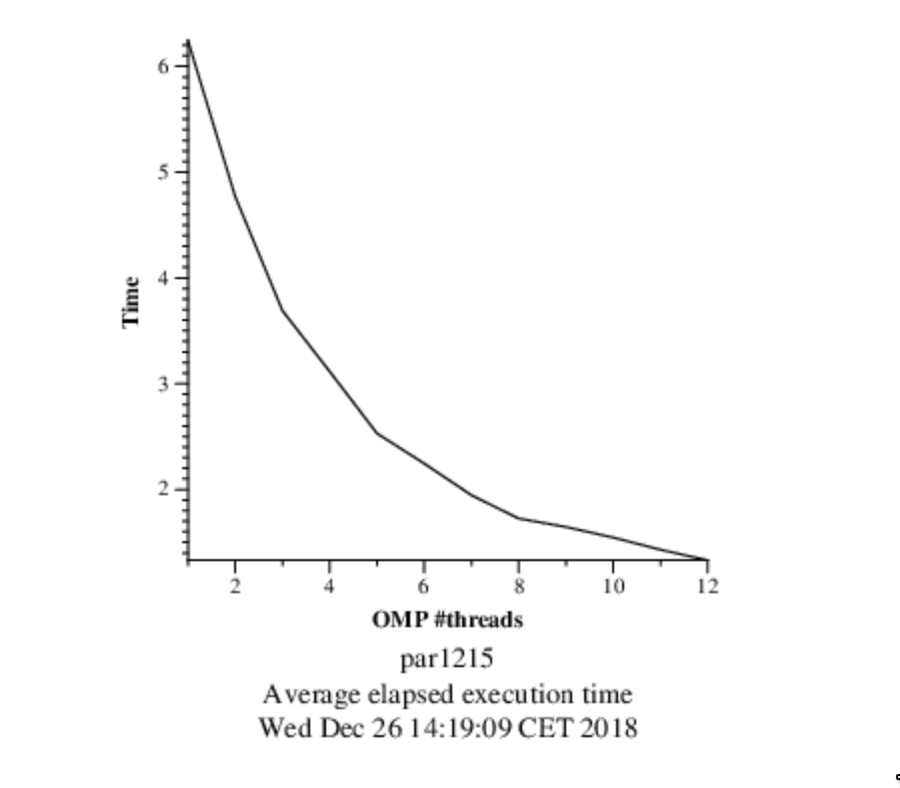
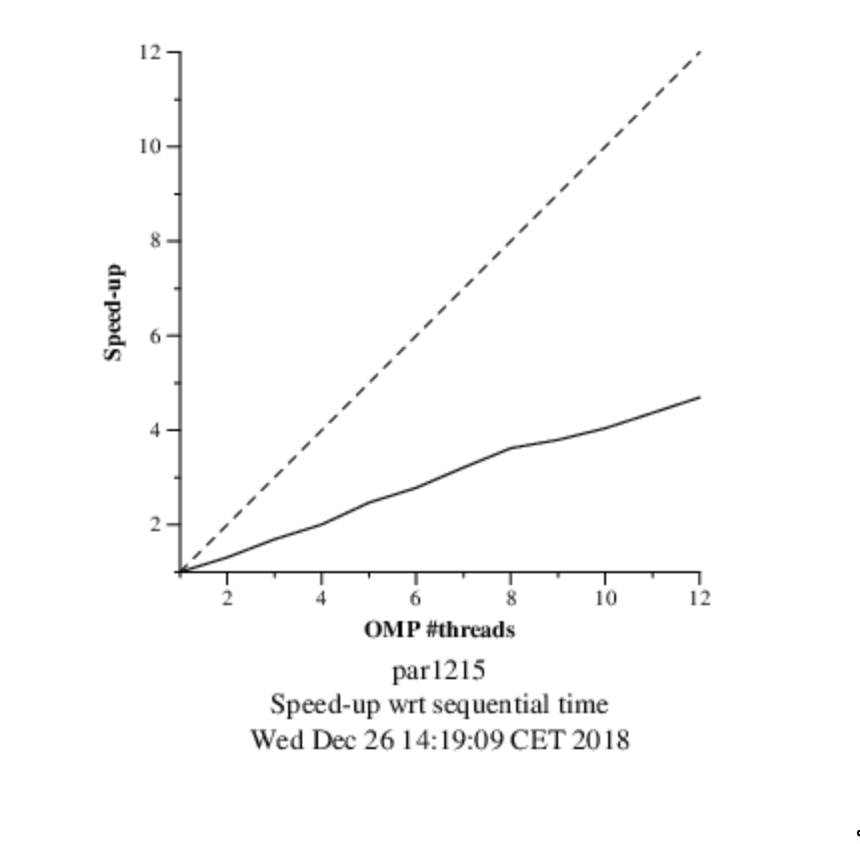
The execution with the paraver is made with 250 iterations, because with the defaults ones there is not enough space to generate the traces. the results in the paraver allow us to see the action of the threads. The main thread creates the others when it enters in the synchronization phase, as we see in Image 23. Then they will be running until the scheduling phase is reached. In the next image we can observe the entry and exit points of the different OpenMP activities mark with the flags.

[[23]](#footnote-22)

[[24]](#footnote-23)

[[25]](#footnote-24)

The time and speed up graphics are made with 8 threads. As we can see on this figures he results are not as good as in the jacobi case because there are dependencies among the iterations.

[[26]](#footnote-25)[[27]](#footnote-26)

1. Imagen 1 [↑](#footnote-ref-0)
2. Imagen 2 [↑](#footnote-ref-1)
3. Imagen 3 [↑](#footnote-ref-2)
4. Imagen 4 [↑](#footnote-ref-3)
5. Imagen 5 [↑](#footnote-ref-4)
6. Imagen 6 [↑](#footnote-ref-5)
7. Imagen 7 [↑](#footnote-ref-6)
8. Image 8 [↑](#footnote-ref-7)
9. Image 9 [↑](#footnote-ref-8)
10. Image 10 [↑](#footnote-ref-9)
11. Image 11 [↑](#footnote-ref-10)
12. Image 12 [↑](#footnote-ref-11)
13. Image 13 [↑](#footnote-ref-12)
14. Image 14 [↑](#footnote-ref-13)
15. Image 15 [↑](#footnote-ref-14)
16. Image 16 [↑](#footnote-ref-15)
17. Image 17 [↑](#footnote-ref-16)
18. Image 18 [↑](#footnote-ref-17)
19. Image 19 [↑](#footnote-ref-18)
20. Image 20 [↑](#footnote-ref-19)
21. Image 21 [↑](#footnote-ref-20)
22. Image 22 [↑](#footnote-ref-21)
23. Image 23 [↑](#footnote-ref-22)
24. Image 24 [↑](#footnote-ref-23)
25. Image 25 [↑](#footnote-ref-24)
26. Image 26 [↑](#footnote-ref-25)
27. Image 27 [↑](#footnote-ref-26)