PAR Laboratory Assignment

Lab 3: Divide and Conquer parallelism with OpenMP: Sorting

Sandra Flores Hidalgo

Celia Fernández Pérez

par1215

**INDEX**

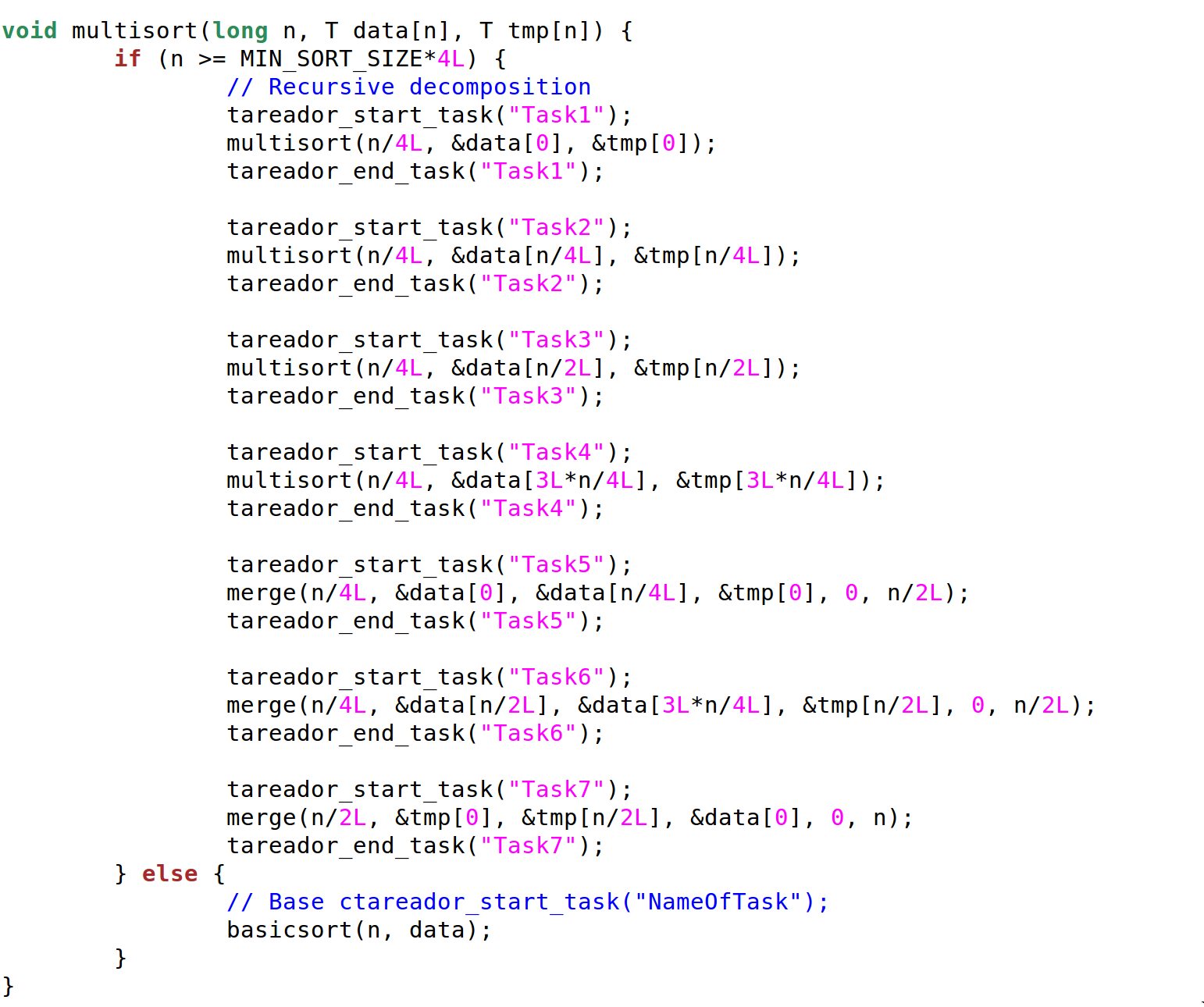
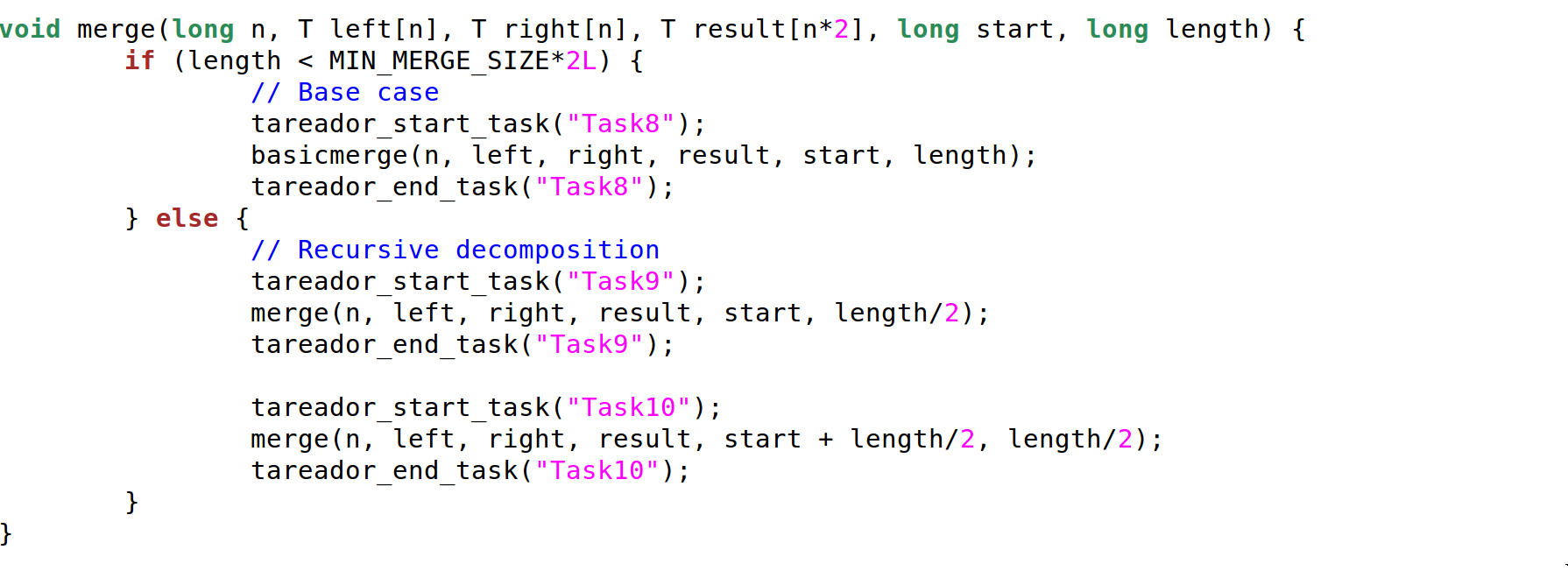
**1. ”Divide and conquer”, Analysis with tareador**

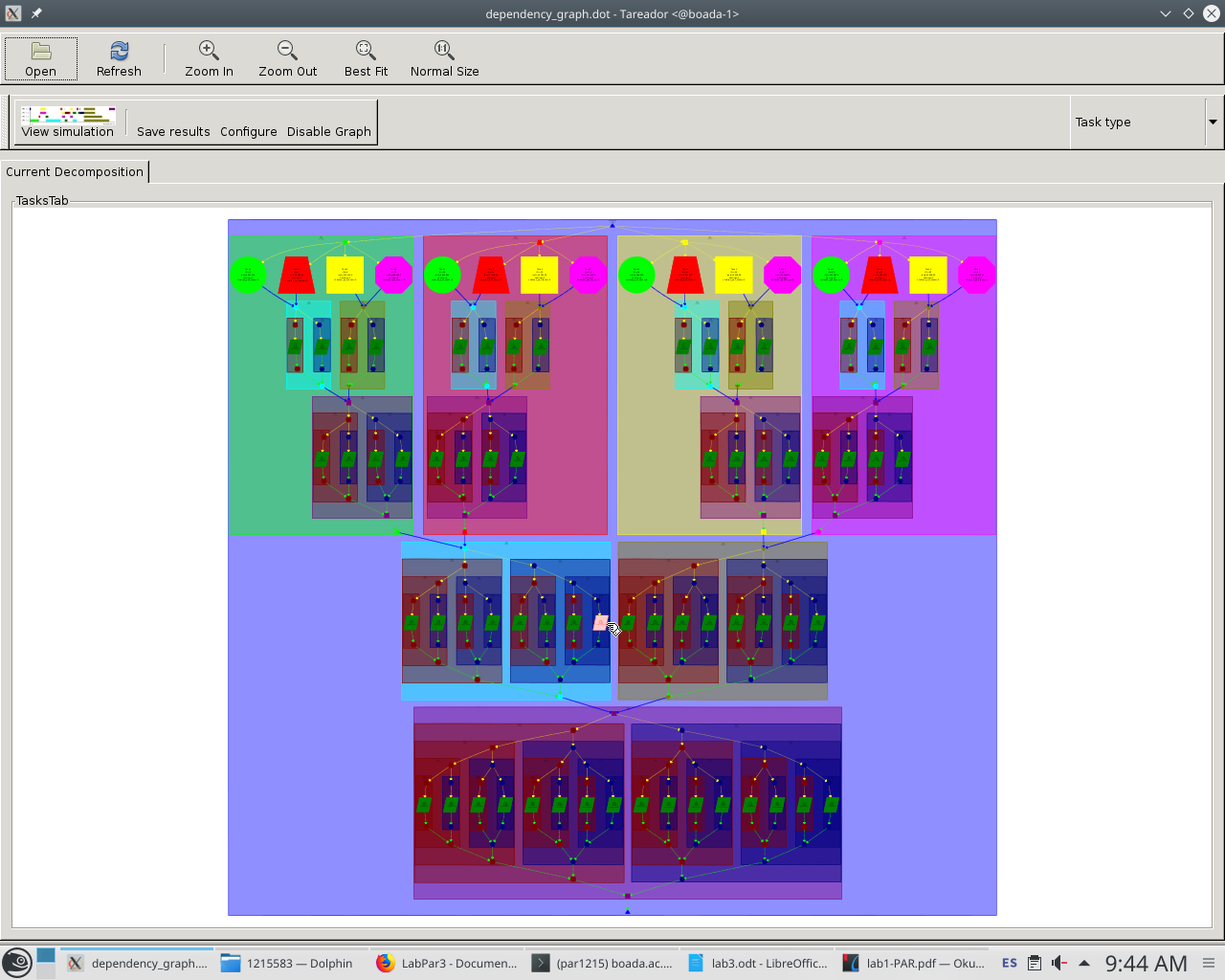
**2. Shared-memory parallelization with OpenMP tasks**

**3. Using OpenMP task dependencies**

**1.”DIVIDE AND CONQUER”, ANALYSIS WITH TAREADOR**

In this part we are going to work with the “divide and conquer” strategy. To obtain a graphic representation of the process we have to insert Tareador instrumentation in our code, as we show below.

****



The “Divide and conquer” strategy is shown in this image, each of the tasks have assigned a different color:

* The green color correspond to task1
* The pink color correspond to task2
* The yellow color correspond to task3
* The light purple color correspond to task4
* The light blue color correspond to task5
* The grey color correspond to task6
* The dark purple color correspond to task7
* The dark green color correspond to task8
* The brown color correspond to task9
* The dark blue correspond to task10

We analyze the code and observe that the tasks 1, 2, 3 and 4 can do in parallel because each one trate a quarter part of the array, task 5 and 6 can also can do in parallel because each one trate the half part of the array and the task7 have to do alone because it trate al the array. Task1 and task2 have dependence with task5 because trate the same part of the array. Task3 and task4 have dependence with task6 by the same reason. And task5 and task6 have dependence with task7.

In this table we can see our code modified to be able to work with the Tareador. The calls to the Tareador API are place in each call to both multisort and merge function, so this allow us to appreciate which of the calls to this functions can be realized in parallel and which dependences exists.

Analyzing the graph obtained we see that tasks 1, 2 ,3 and 4 can be done in parallel because, even they modify the same arrays, they work in differents parts of the array data and tmp, a quarter part of each is distributed to each tak. This tasks call recursively to the multisort function. Then the tasks 5 and 6 split the data and tmp arrays between them, so they can be executed in parallel too. Both of them call the function merge on a recursive way. Finally the task 7 calls the the merge function and works with the whole data and tmp arrays.

The dependences exists between this three groups, tasks 1, 2, 3 and 4, taks 5 and 6 and task 7, because the modify the data and tmp arrays.

Because of the calls to the merge function the tasks 8, 9 and 10 are executed too. In the graph we can appreciate that tasks 9 and 10 can be done in parallel but task 8 no, because there are dependences between this task and the other two.

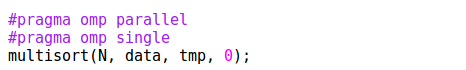
The table with time execution and speedups are the next:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 4 | 8 | 16 | 32 | 64 |
| Execution time(s) | 20.334 | 10.174 | 5.068 | 2.550 | 1.290 | 1.2848 | 1.2898 |
| SpeedUp | 1 | 2 | 4 | 8 | 15.7643668 | 15.7649136 | 15.7649136 |

The speedup don not go to the ideal case, because at the end in the 32 and 64 cores is the same.

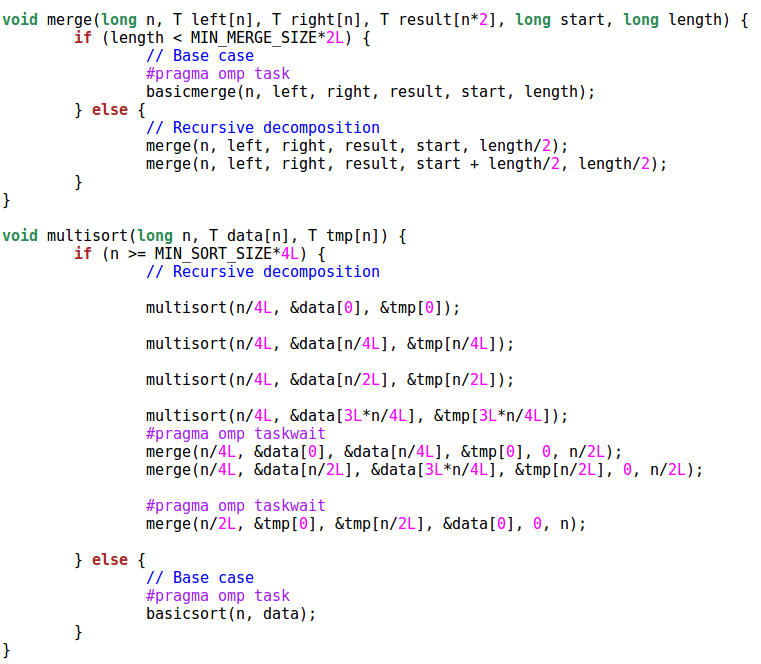
**2.SHARED-MEMORY PARALLELIZATION WITH OPENMP TASKS:**

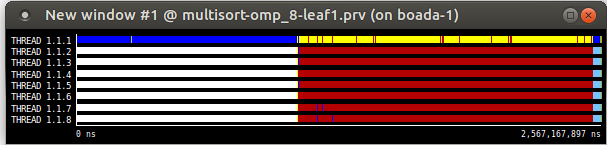
The follow code are the main part, it will be the same in all the versions codes (leaf, tree and tree with cut-off).

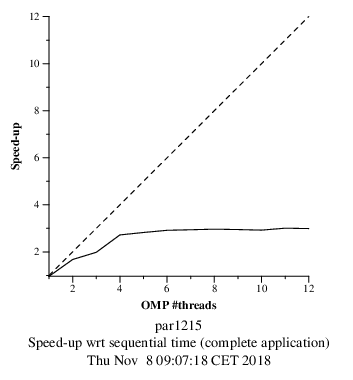
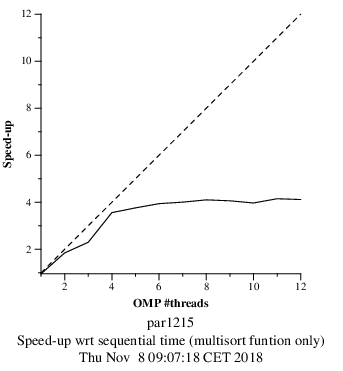
****

**Leaf version without cut-off:**

To write the leaf version of the code we assign a parallel composition to the multisort call, so it will be executed in parallel. First, because of the #pragma omp single, the call is going to be executed by one thread at a time. This task will execute the first four recursive calls to multisort, modifying the whole data and tmp vectors. Then it is going to find a barrier, as we can see, the #pragma omp taskwait, so it is going to wait, as all the other tasks until all of them finish. Then each one of the tasks call to the merge function, the first call allow us to merge

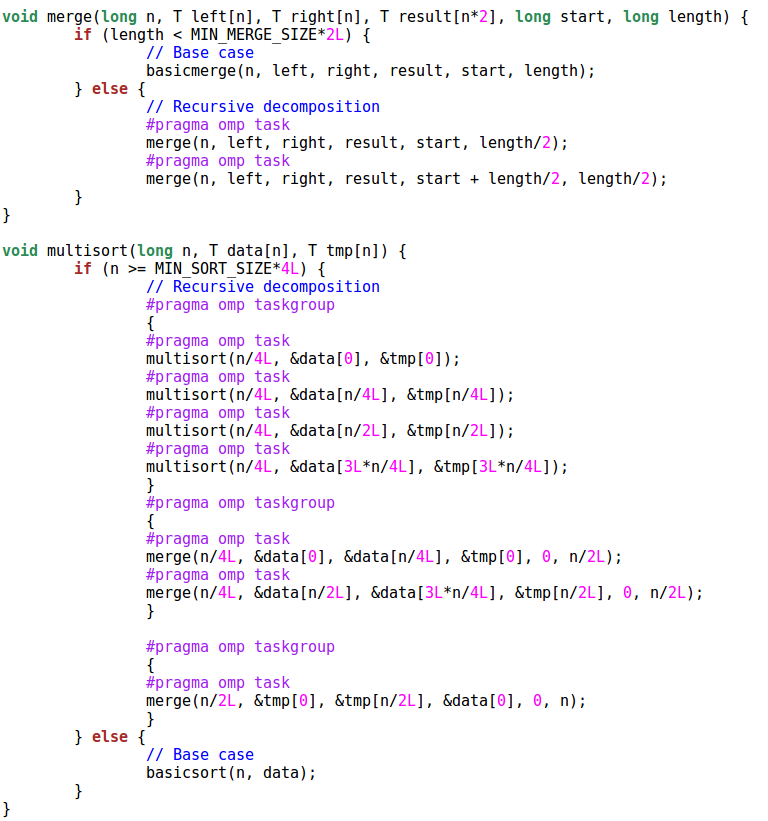


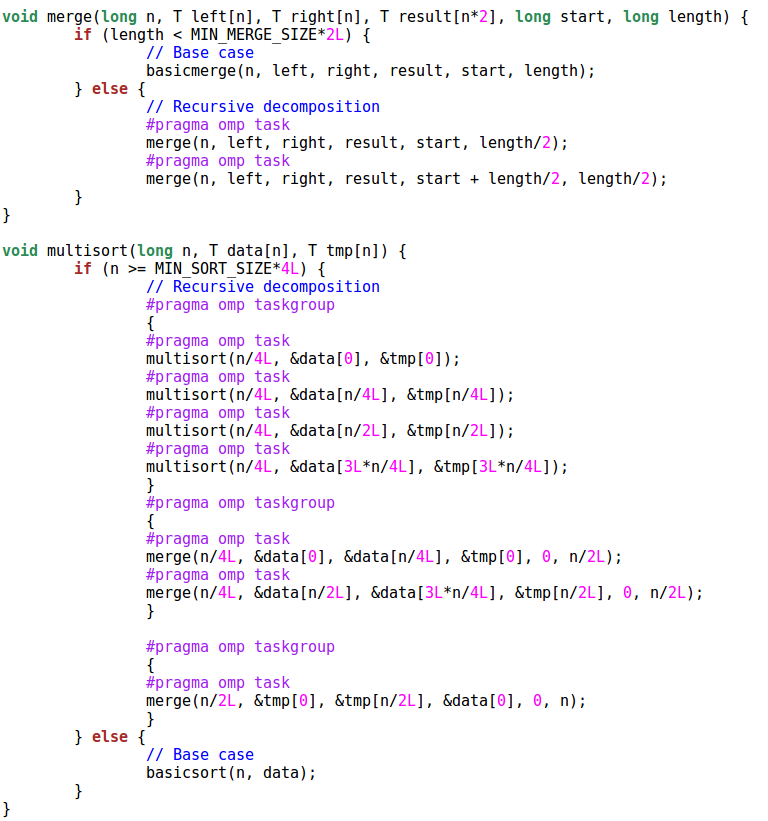




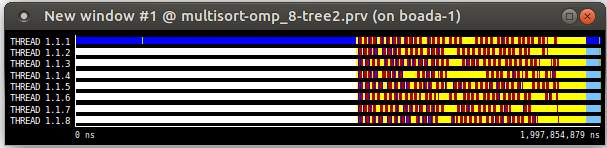
**Tree version without cut-off:**

To write the tree version of the code we write #pragma omp task before all the calls to merge and multisort recursive functions. And for avoid the dependencies between the three parts of the code than have it (we explain this in the first headland) we write #pragma omp taskgroup this cause that the calls to recursion functions wait for the previous tasks.

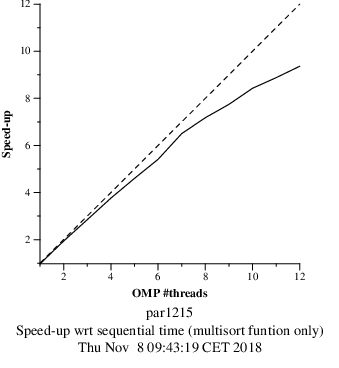
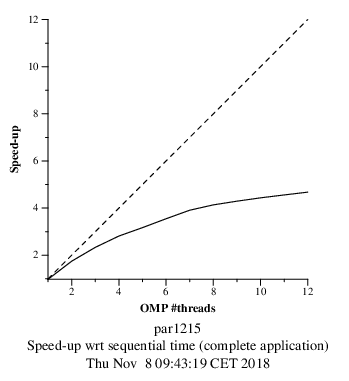
****

****

In this graphic we can view the creation of the tasks in each recursion call.

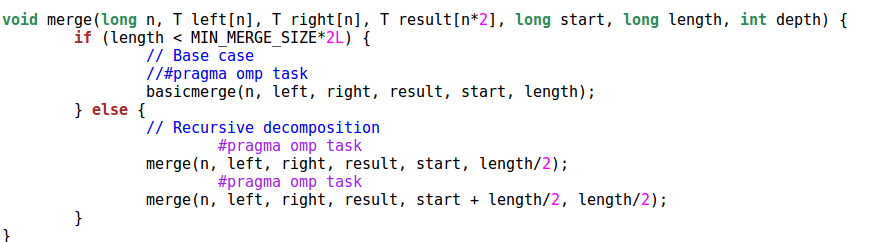


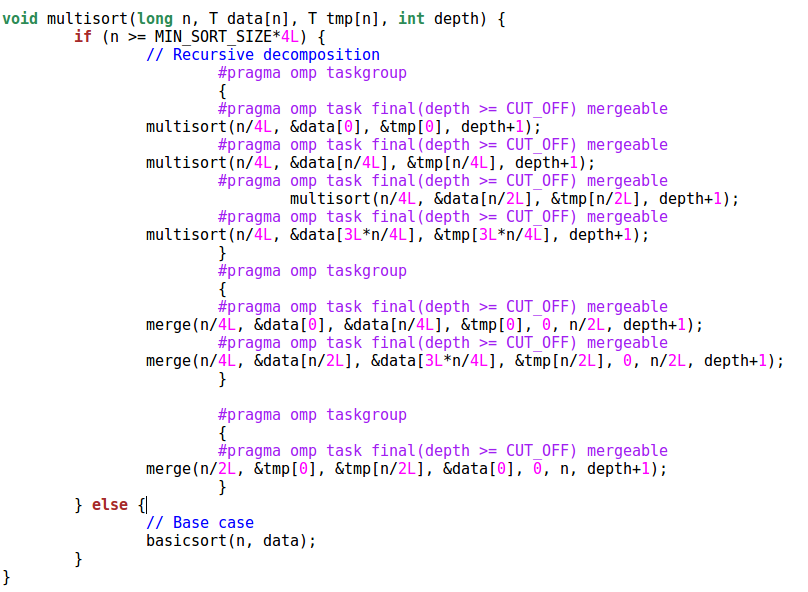
The plots shows that the speedup in better in this version than the leaf, because the task are not created at the end of all the calls, they are created while the calls are occurring.

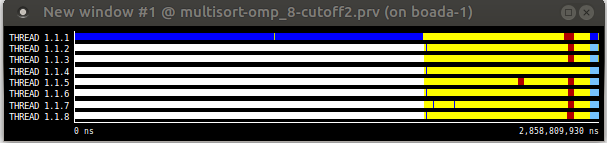


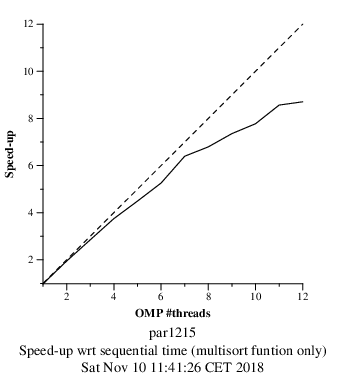
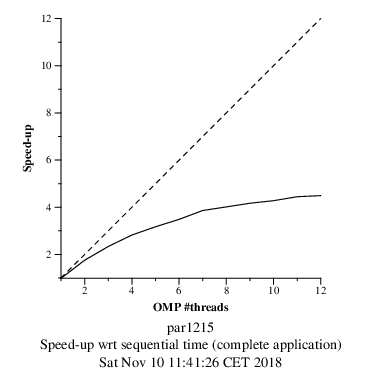
**Tree version with cut-off:**

To write the cut-off version we write the clause “first” with a condition inside (depth >= CUT\_OFF) this makes that does created task until the condition are true. And we add the clause mergeable because if the task is undeferred task or included task, amerge task might be generated.

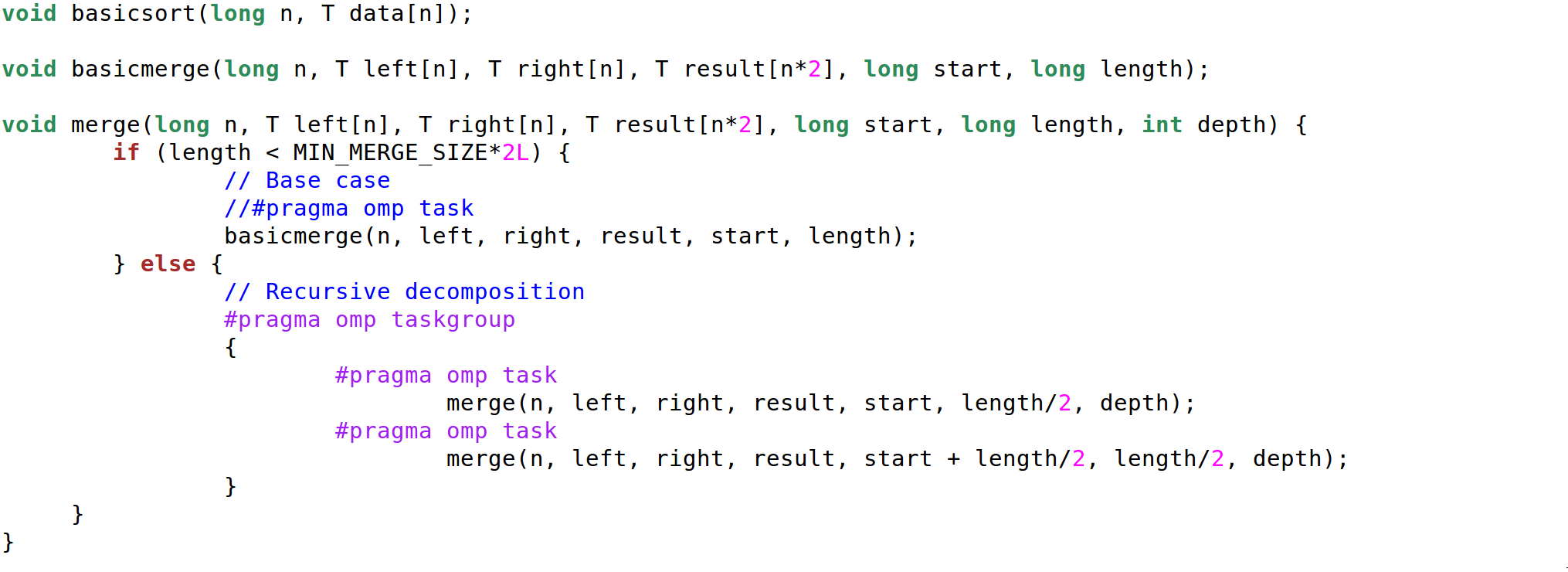
****

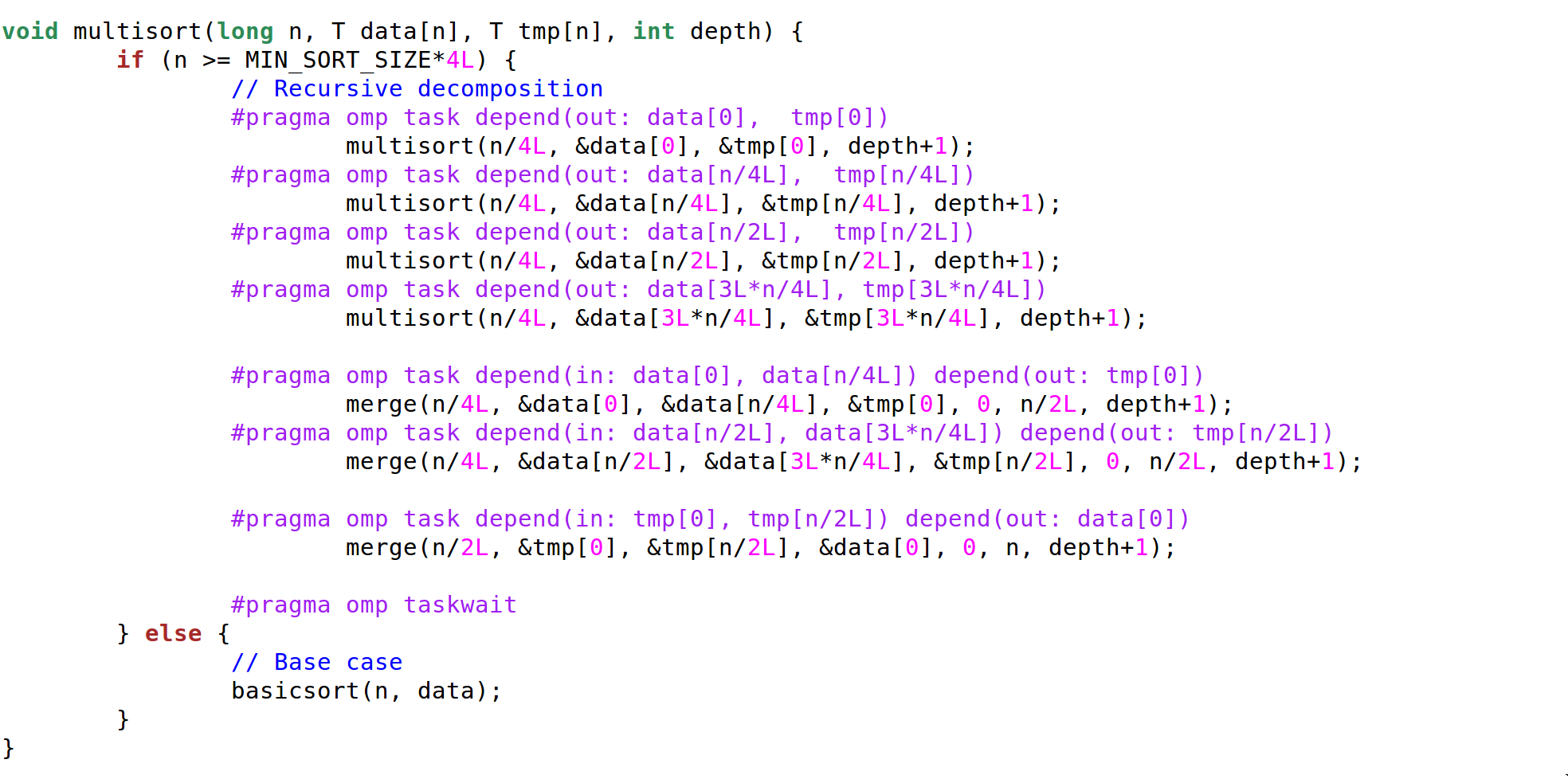
****

The graphic shows that the synchronization are at the end of the calls, thats is because is when the task have stopped creating.****

****

**3. Using OpenMP task dependencies**

****

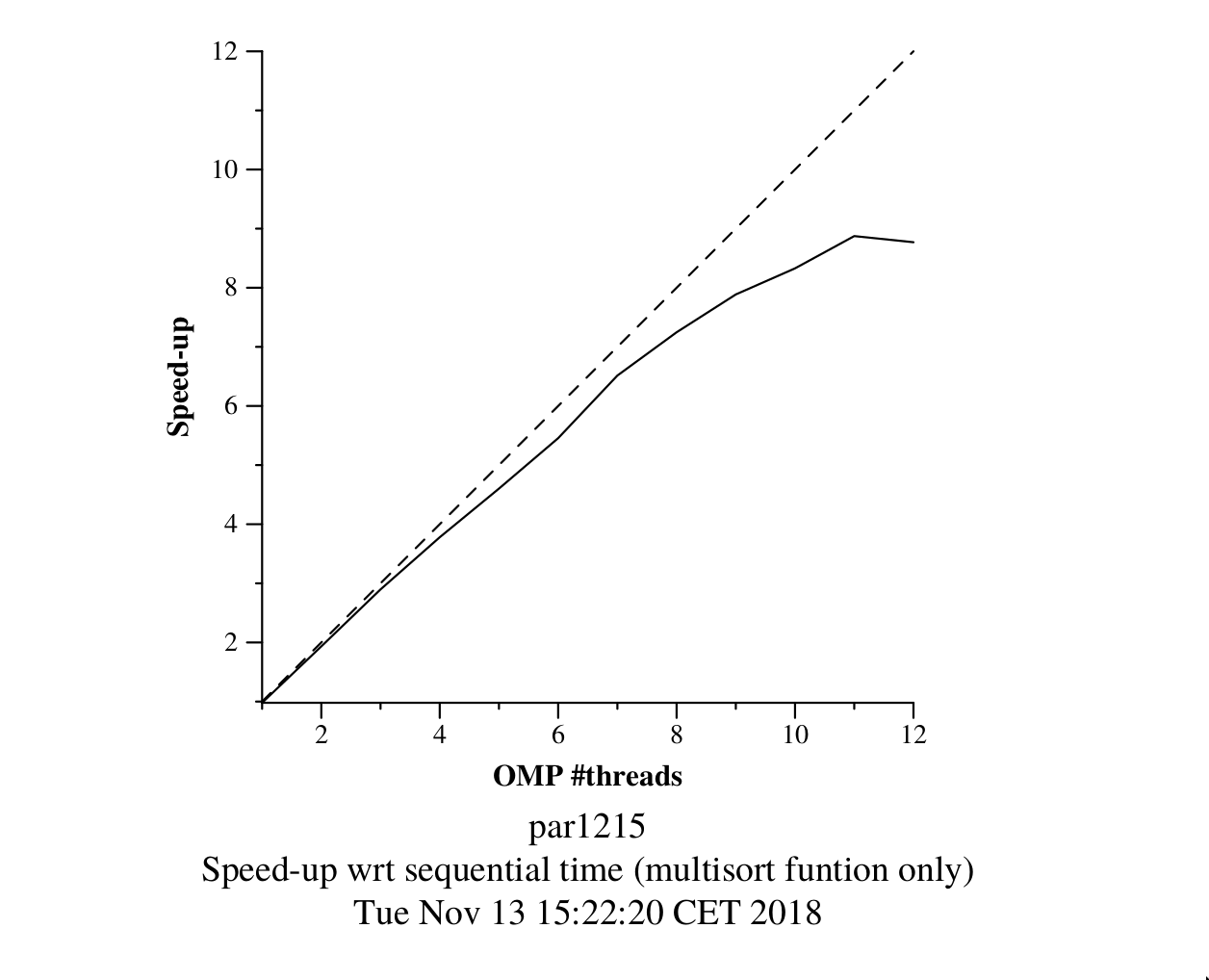
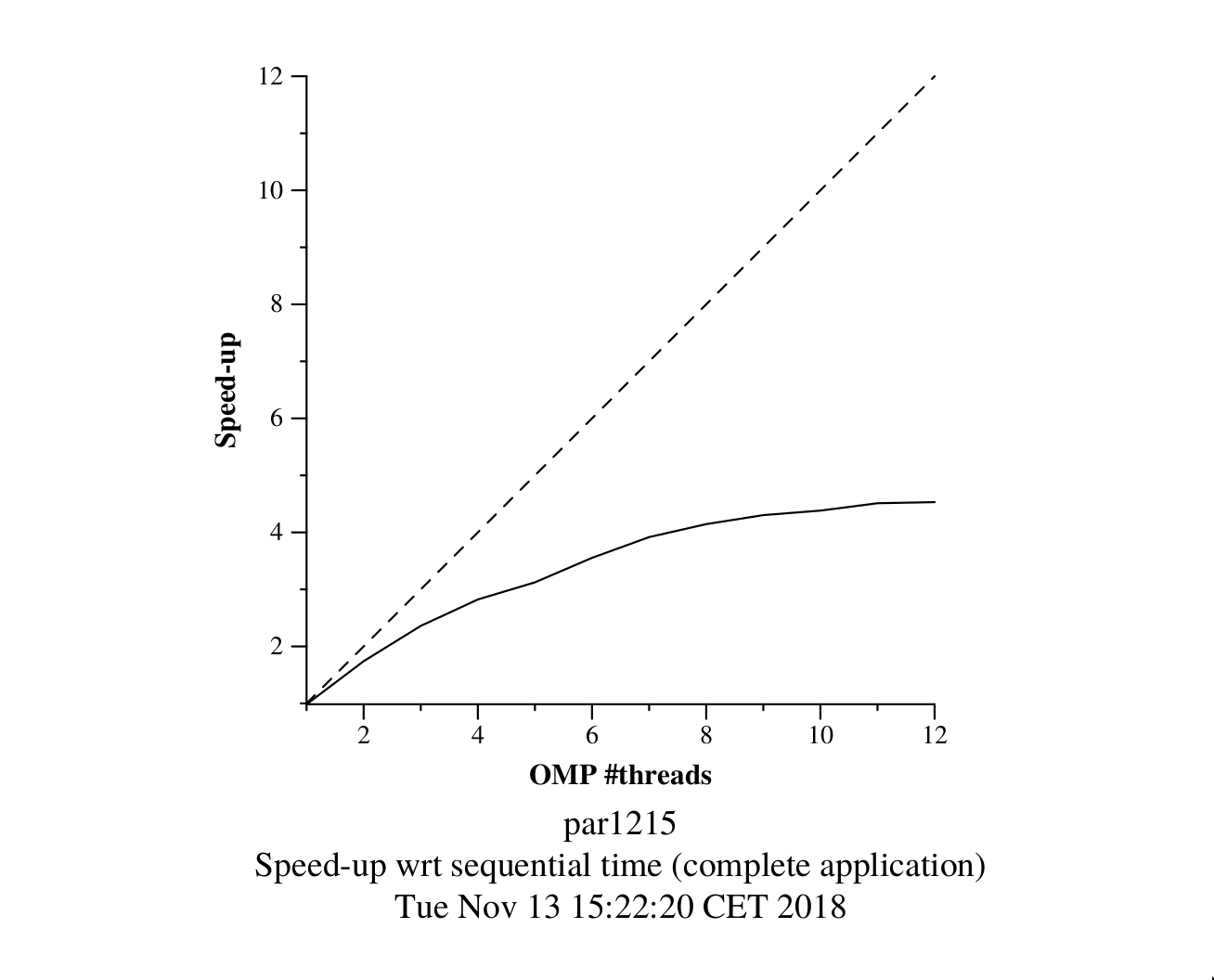
****

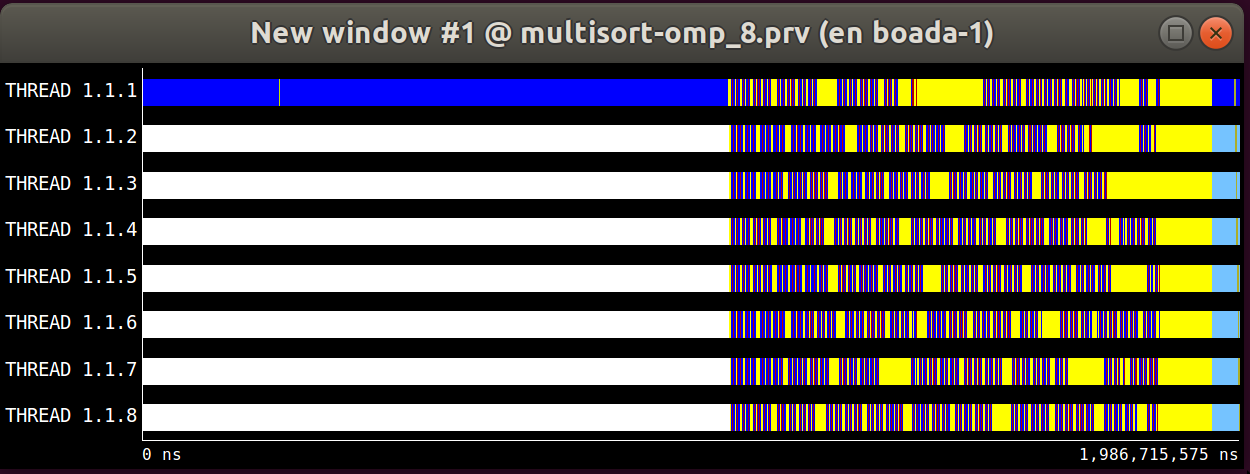
As it says on the practice statement, we design this code defining dependencies between the tasks to avoid using the taskgroup.

On the merge function we added a taskgroup call to ensure that all the tasks end before return to the parent function.

On the multisort code we create dependencies between the four multisort calls and then between the merge calls too. These dependencies ensure that the tasks, in the multisort calls case, are going to wait until all four of them finishes. The same happens with the first two calls to merge. Finally we use a taskwait to guarantee that any task is going to finish the function call before each one of them ended up.

The execution of this code and later the use of Paraver allow us to obtain the graphic representation of the results as we can see in the next images.

****

****

The results shown in these two graphics are very similar to the results obtained when using the taskgroups. As we can see in the tareador image and analyzing the code, there are many situations in which the threads have to be waiting for other threads to finish, these dependencies limits the parallelism of our code so we obtain not such good results.