LABORATORY 4

Group components: Albert Bausili Fernández, Noa-Yu Ventura Vila

Group identifier: par2301

1/12/2022

Index

multisort-tareador-leaf.c funciona

multisort-tareador-tree.c funciona

multisort-omp-leaf.c funciona

multisort-omp-tree.c funciona

(fins ara fets a classe amb el profe)

multisort-omp-tree-cutoff.c funciona

multisort-omp-tree-cutoff-dependency.c funciona no com el profe vol :(

1. Task decomposition analysis for Mergesort

1.1 “Divide and conquer”

We submitted multisort-seq.c and got the following output. It’s important because we will use them as reference times to check the scalability of the parallel versions in OpenMP we will develop in another section of the project.

//For the deliverable: Take note of the times reported for the sequential execution and use them as reference times to check the scalability of the parallel versions in OpenMP you will develop.

Initialization time in seconds: 0.683093

Multisort execution time: 5.185487

Check sorted data execution time: 0.011297

1.2 Task decomposition analysis with Tareador

//For the deliverable: Include both tareador codes (in the .zip file) and show code excerpts in the report.

In figure 2 we can see how we implemented the leaf decomposition strategy using tareador. We create a task for each base case, so creation of tasks is sequential.

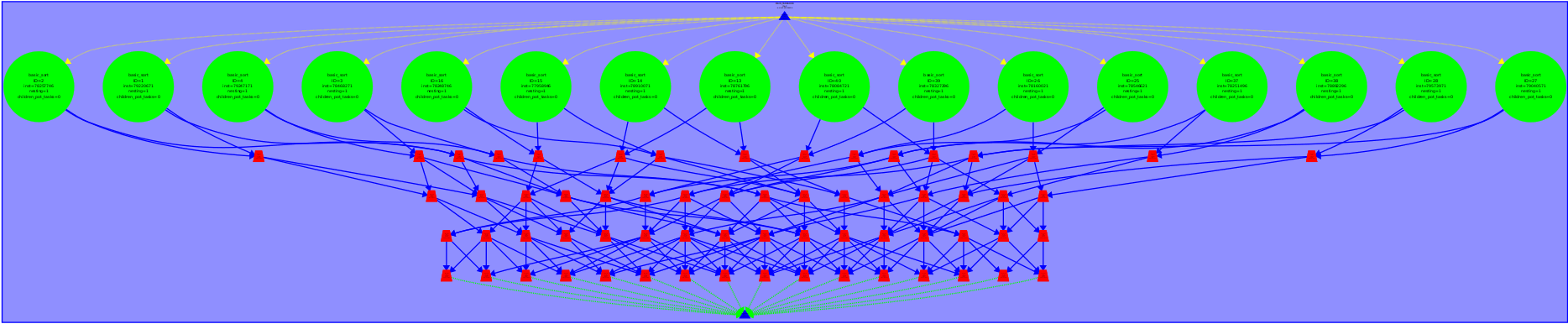


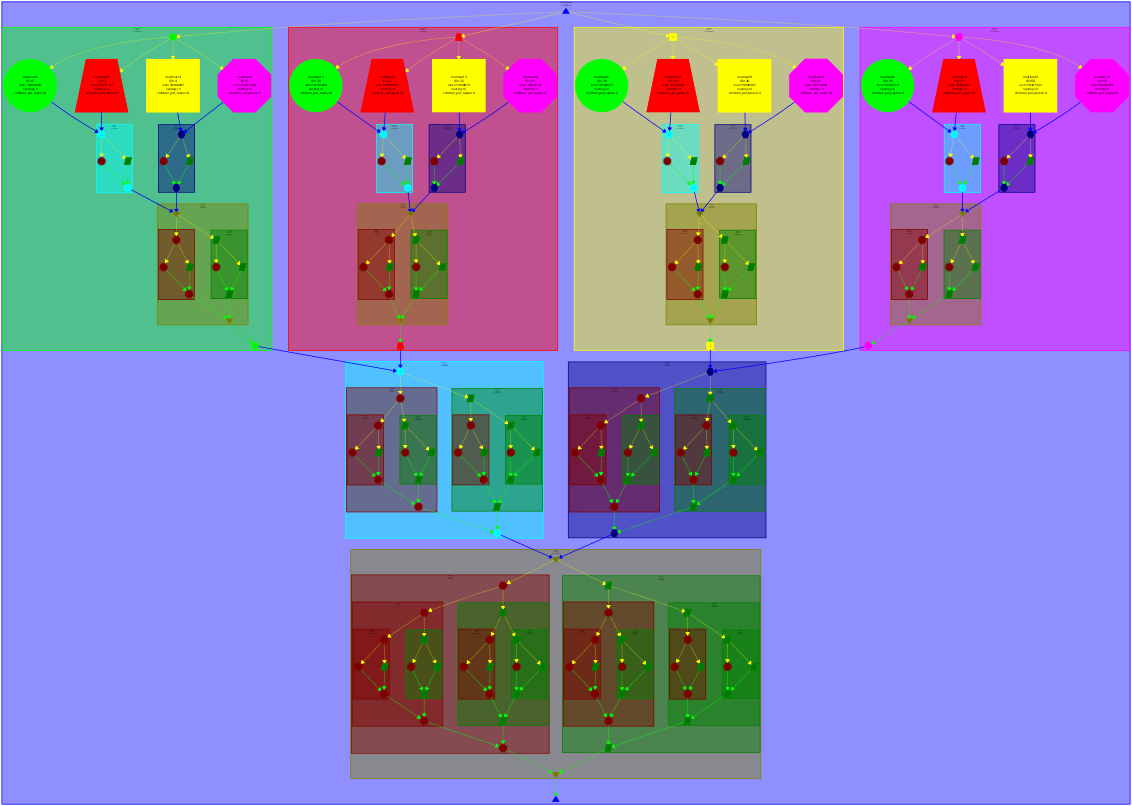


In contrast with leaf strategy decomposition, we have tree strategy decomposition which *tareador* implementation is shown in figure 3. In this case, for each recursive call to merge and multisort we create a task, therefore the creation of tasks is made in parallel.

//For the deliverable: From the task dependence graphs that are generated for leaf and tree, do you observe any major differences in terms of structure, types, number and granularity of tasks, ...? Save the image of the two graphs in order to include them in the report.

The following figure (figure 4) shows the task dependence graph of the leaf strategy, and figure 5 for the tree strategy.







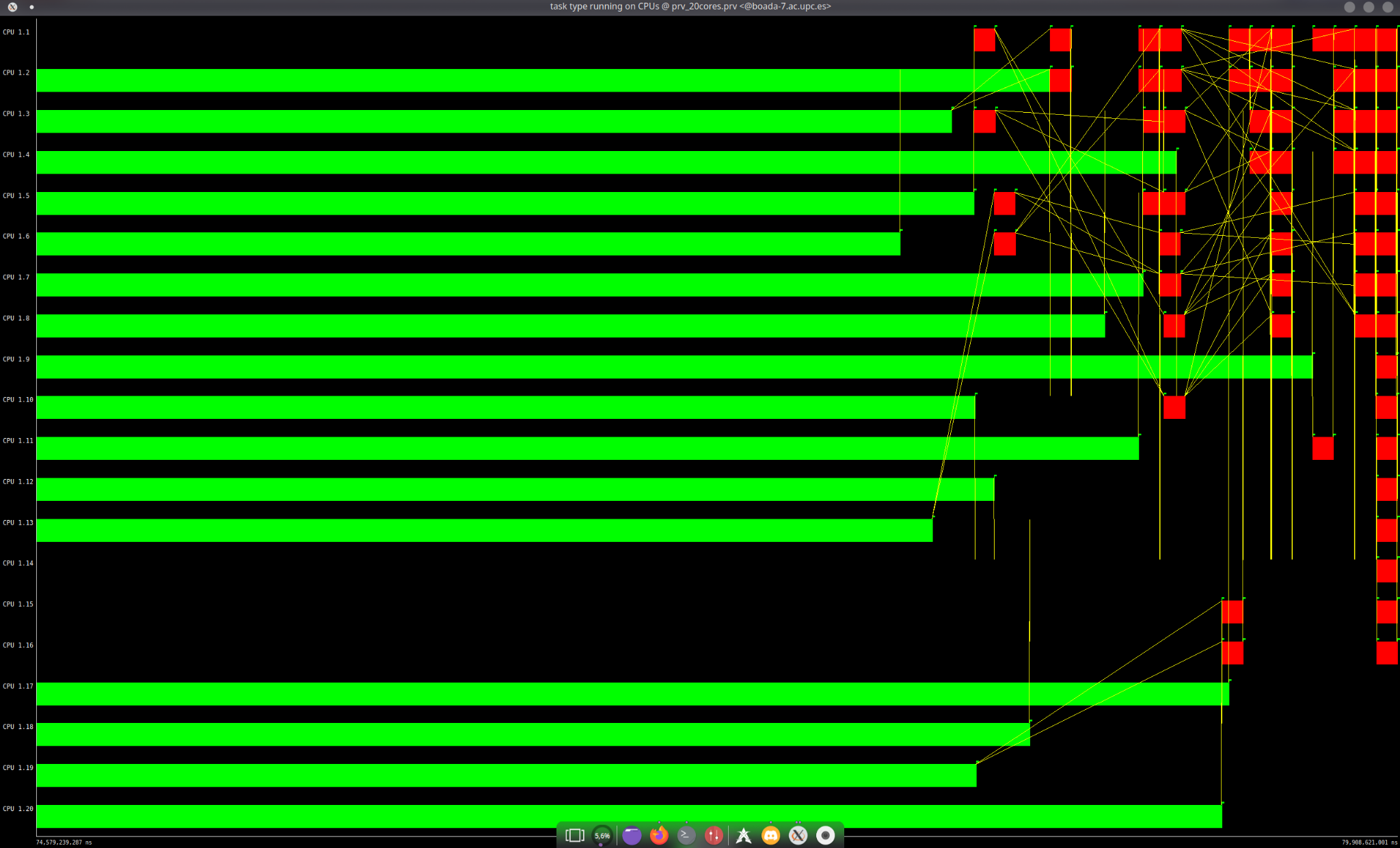
For the leaf strategy we can see that tasks are created sequentially and all of them have a similar granularity, unlike tree strategy where tasks are created in parallel and a task may have a granularity way bigger than another one. It should also be noted that there’s a higher number of tasks for the tree strategy than the leaf strategy, because in the first one we create a task for each recursive call which is made way more times than the number of calls to the base case functions basicmerge and basicsort.

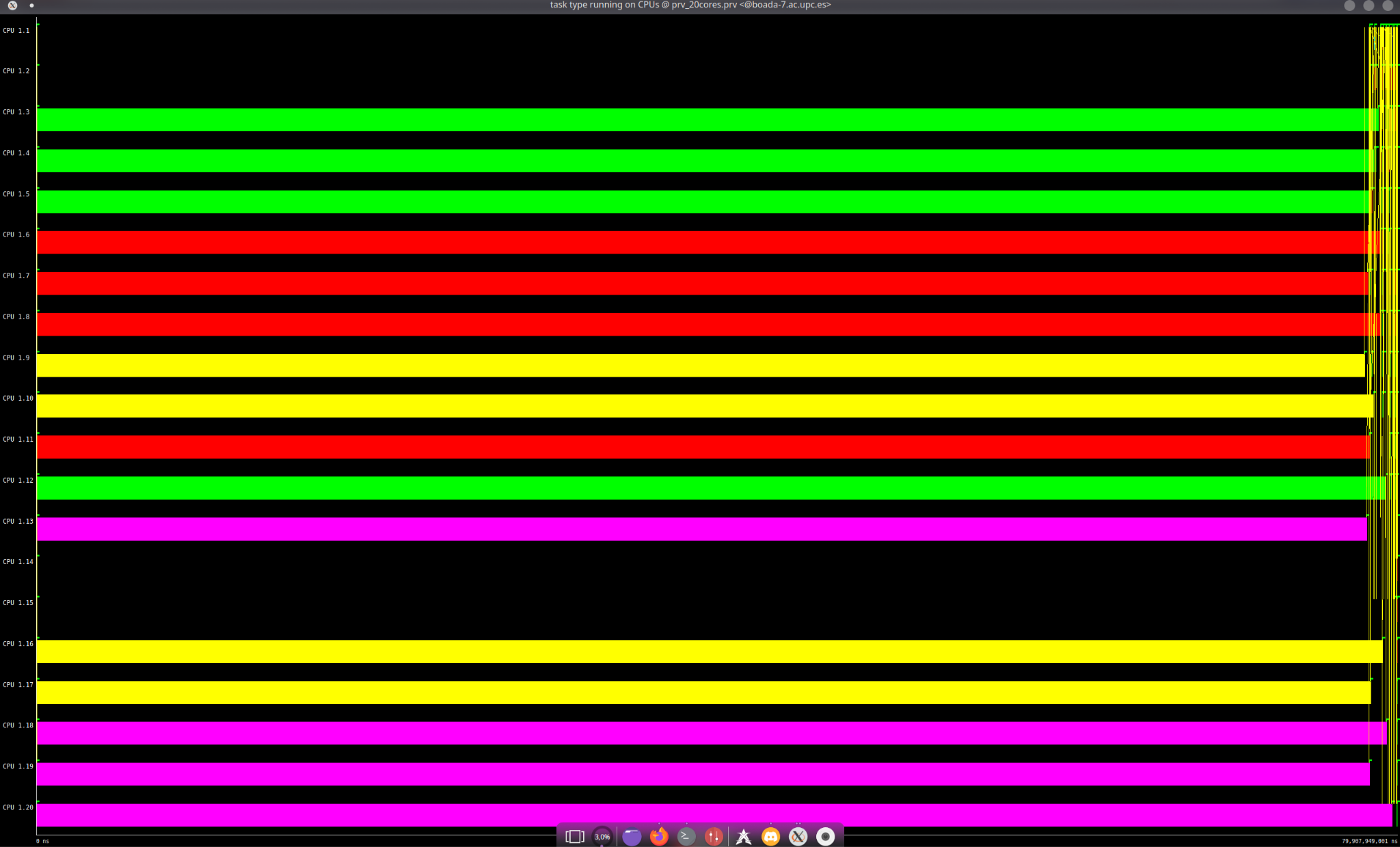
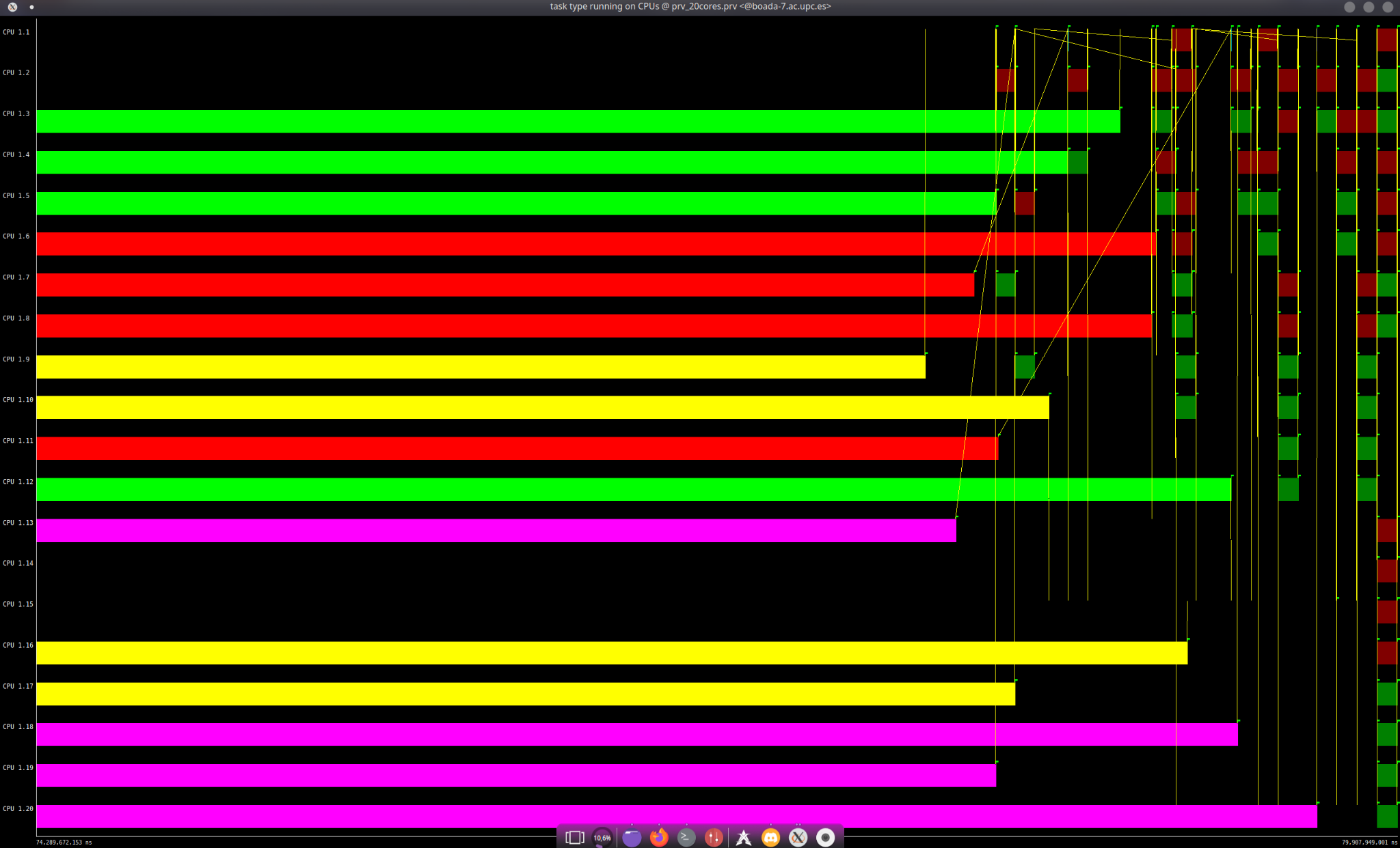
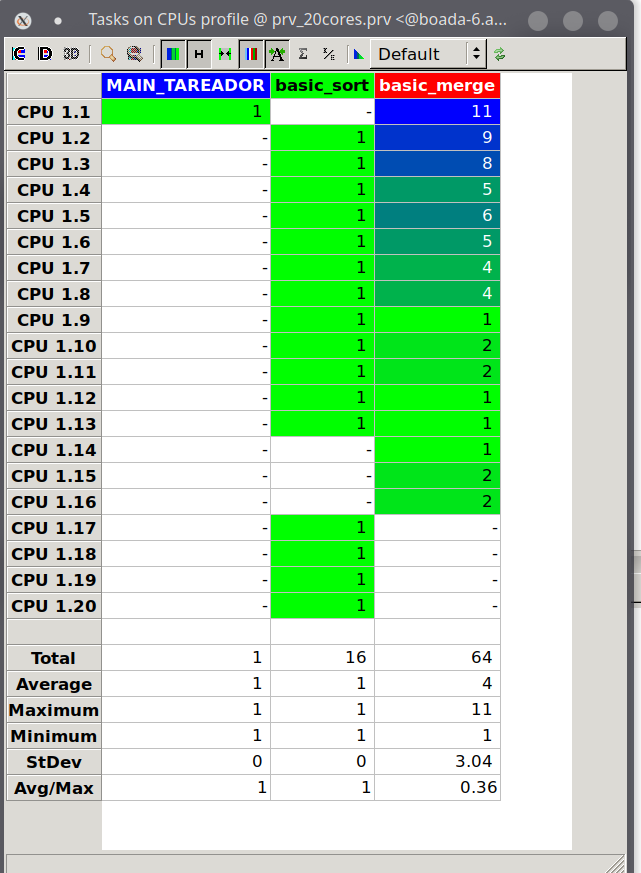
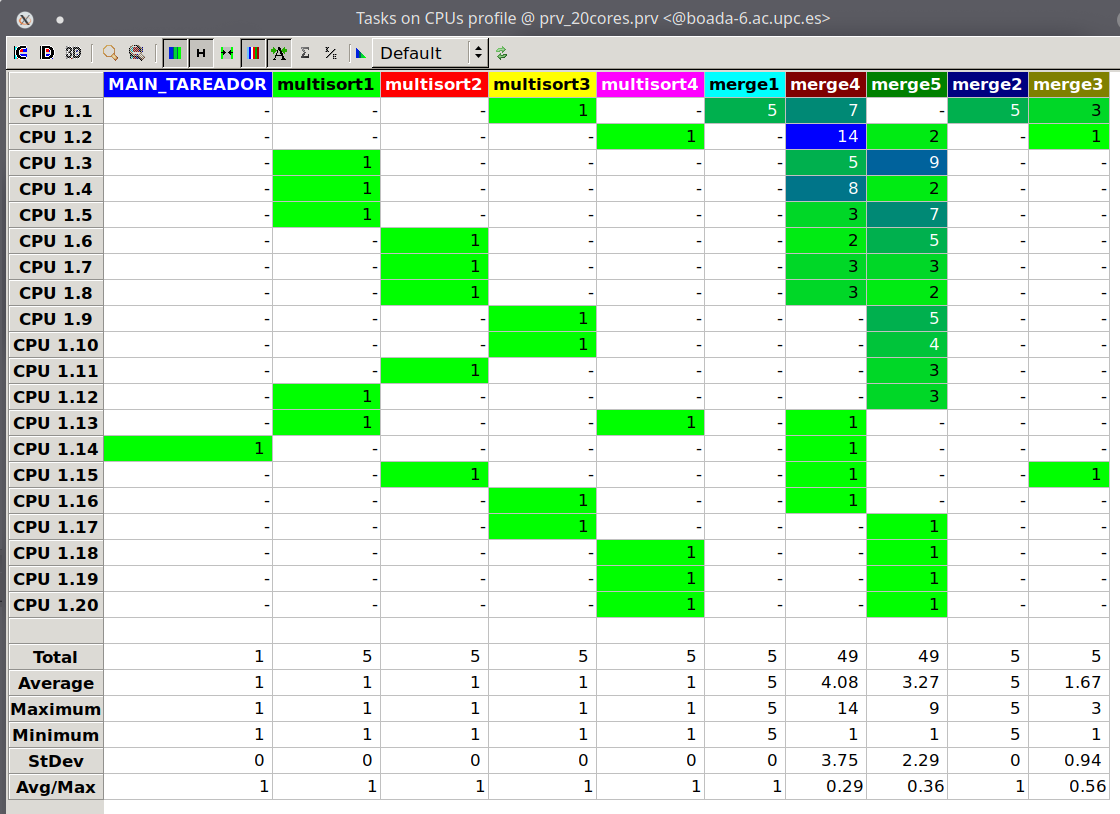
//For the deliverable: Identify the points of the code where you should include synchronizations to guarantee the dependences.

Creating tasks and parallelizing code may influence the result we expected from the function, which is caused by the dependencies between tasks. In this case, we have a strong dependency between the last call to multisort and the first call to merge (see figure 6 dependency 1) and between the second call to merge and the last call to merge (see figure 6 dependency 2). This is due to the nature of the sorting algorithm merge, which needs each of its blocks sorted in order to merge them into a sorted bigger block. This is, we need the four blocks obtained from multisort to be sorted, and then merge two of them together with a function merge getting two blocks, and finally call to the last function merge to sort the last two blocks together.



//For the deliverable: Do simulated executions with 20 processors for each task decomposition. From the Paraver windows, do you notice any differences in terms of how and when tasks doing computation are generated? You will have to zoom at the appropriate parts of the trace in order to observe these differences. Capture the necessary Paraver windows in order to support your explanations in the deliverable.

The following figure helps us to prove this point for the leaf strategy.



2. Shared-memory parallelisation with OpenMP tasks

//For the the leaf and tree strategies with no cut–off mechanism, include the speed–up (strong scalability) plots/tables that have been obtained for the different numbers of threads. Reason about the performance differences that you detect between leaf and tree implementations, including tables, or relevant parts of tables, generated by modelfactors.py that help you to explain and justify the differences between them. Support your explanations with the number of tasks created, their granualarity, and the information of instantaneous parallelism view of Paraver . Finally, explain what is causing the different scalability that is obtained for the whole program and for the multisort function only.

2.1 Leaf strategy

2.1.1 Scalability

//Analyze the scalability of your parallel implementation, threads in the range 1 to 20. Is the speed–up achieved reasonable? (al pdf explica com es fa)

2.1.2 Analysis with modelfactors

2.1.3 Analysis with *Paraver*

//trace the execution of the parallel execution with 1, 2, 4, 6, 8, 10, 12, 14 and 16 threads.

//For the deliverable: Include the three tables generated. Which of the factors do you think is making the parallelisation efficiency so low? Several options to think about: parallel fraction, in-execution efficiency (related with the overheads of sync and sched reported in the third table), number of tasks and their execution time, load balancing, …

//trace generated for the execution with 8 threads

//For the deliverable: Include one or more Paraver visualisations that helps you explain the lack of scalability. Is the program generating enough tasks to simultaneously feed all threads? How many tasks simultaneously execute?

2.2 Tree strategy with OpenMP

//Repeat the previous steps for the alternative tree strategy: implementation, check correctness, overall analysis with modelfactors and detail analysis with Paraver .

2.2.1 Scalability

//same as leaf strategy:

////Analyze the scalability of your parallel implementation, threads in the range 1 to 20. Is the speed–up achieved reasonable? (al pdf explica com es fa)

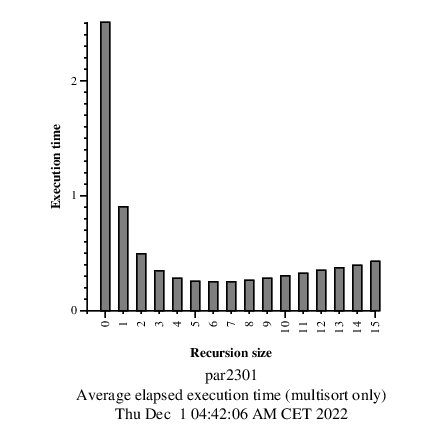
//For the deliverable: Compare the scalability results, table relevant information and traces generated for both strategies and draw the appropriate conclusions. Are those strategies showing a good scalability? Is the granurality of both strategies influencing the parallel performance? What is the number of tasks created?

2.2.2 Analysis with modelfactors

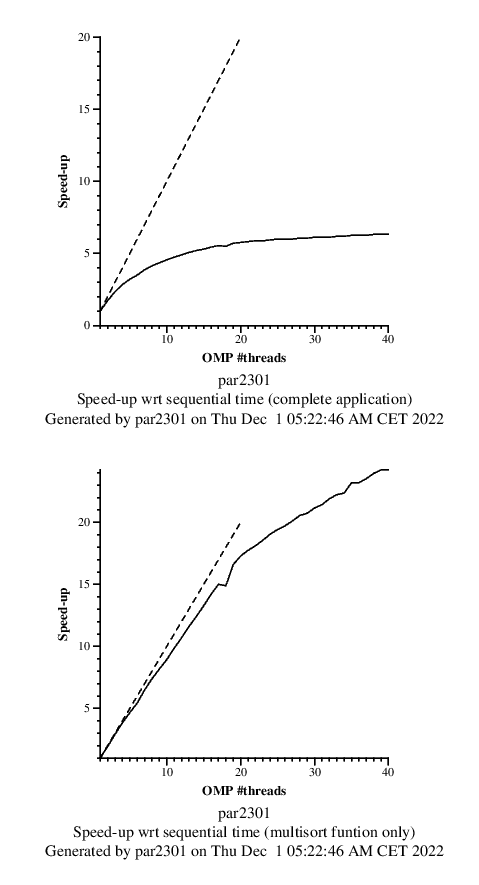
//same as leaf strategy

2.2.3 Analysis with *Paraver*

//same as leaf strategy

2.3 Controlling task granularities: cut–off mechanism

//In this section we propose you to analyze the application to find out the best cut–off level under the point of view of performance of the application.

//For the deliverable: Which is the best value for cut-off? Does it change with the number of threads used? Include the tables of modelfactors for the best cut–off level and the information of the number of tasks generated for each of the cut–off levels used. Reason the number of tasks generated for 0 and 1 cut–off levels.

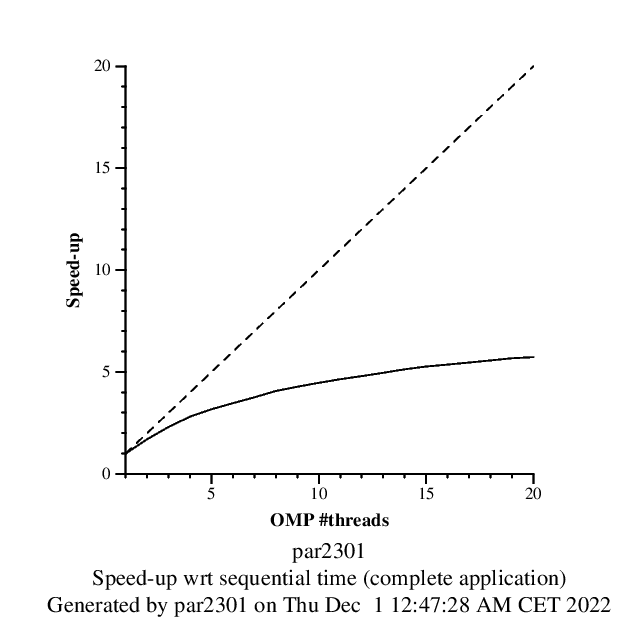
2.4 Optional

//Optional 1: Have you explored the scalability of your tree implementation with cut–off when using up to 40 threads? Why is performance still growing when using more than the 20 physical cores available? Set the maximum number of cores to be used (variable np NMAX) and the cut–off chosen (variable cutoff) by editing the submit-strong-omp.sh script in order to do the complete analysis.

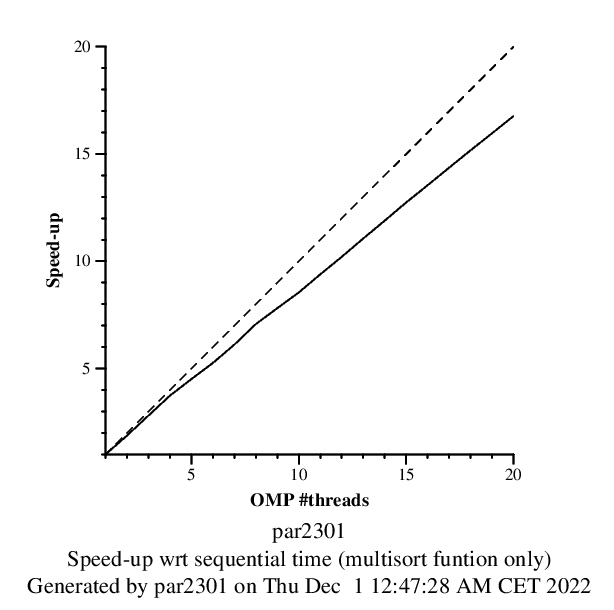
3. Shared-memory parallelisation with OpenMP task using dependencies

3.1 Parallelisation and performance analysis

3.1.1 Scalability plots

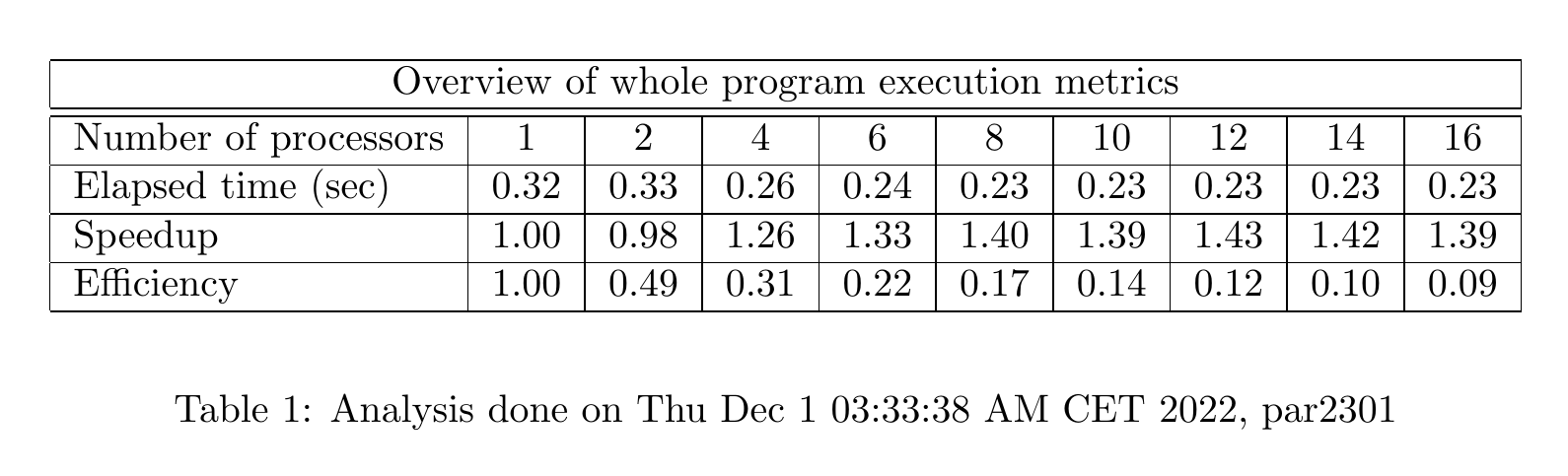
//Analyse its scalability by looking at the strong scalability plots and compare the results with the ones obtained in the previous chapter. Are they better or worse in terms of performance? In terms of programmability, was this new version simpler to code?

The scalability plots for this new version are shown in figure [1 ] and figure [2 ]. As we can see in figure [1], the speedup of the program itself barely improves when increasing the number of threads. This means that excluding multisort the code can be highly parallelized, for example we could parallelize function initialize(). Unlike multisort function which is almost ideally parallelized.

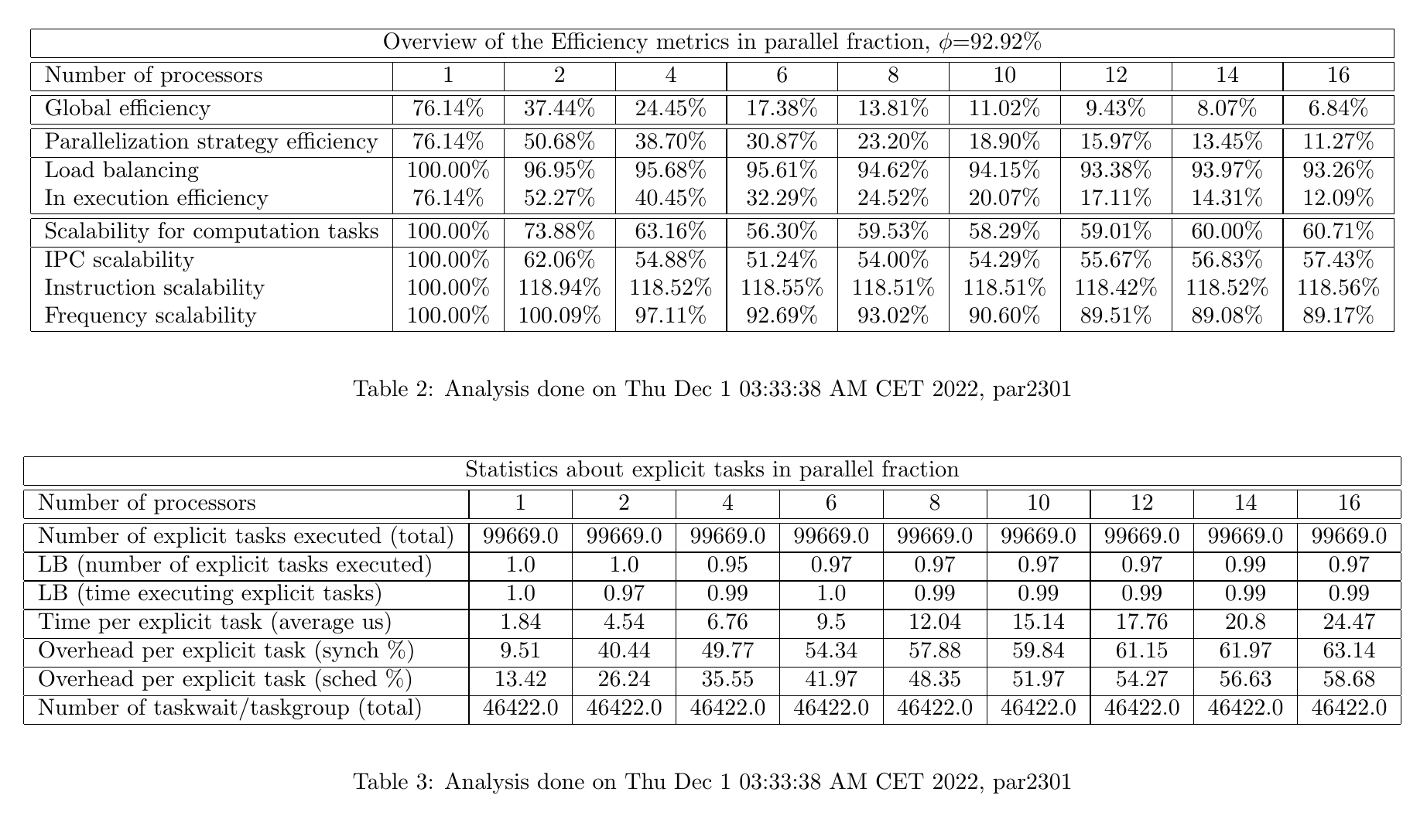
This new version isn’t simpler to code than the previous one, since you need to check what the functions do in order to know if you need an *in*, an *out* or an *inout* dependency. It should be more efficient though, since you don’t block a set of tasks as a group, only the ones that depend on another one, and that one hasn’t finished its execution yet. The scalability plot doesn’t show so, which is because this synchronization provokes overheads, “cancelling” the aforementioned small optimization.

//For the deliverable: Include the tables of modelfactors, scalability plots and Paraver windows to support the comparison.

3.1.2 Analysis with modelfactors

In figure [1] and [2] we can see the modelfactor tables for the version multisort-omp-tree-cutoff-dependency.c file. Its results prove what we mentioned before: the speedup of the whole program remains quite stable, the program can be more parallelized. 

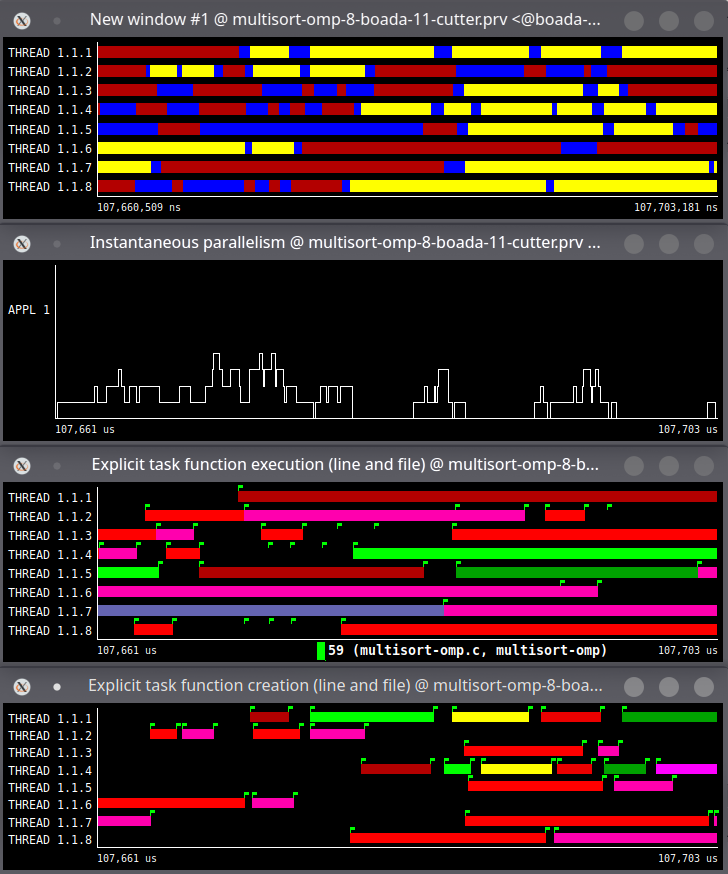
It also shows that creating too many tasks may cause overheads, contrarresting the parallelization from multisort function.



ç

3.1.3 Analysis with *Paraver*

The following figure corresponds to the paraver traces of our version of the code using the depend clause. We can see from the first trace that the overhead and the execution of the program is pretty balanced, which is the reason why its scalability is not ideal.

From the second one we see that sometimes there are no threads being executed, which is a waste of time. But it could also be because of the dependencies generated between the functions as shown in the last two traces.

3.2 Optional

//Optional 2: Complete your parallel implementation of the multisort-omp.c by parallelising the two functions that initialise the data and tmp vectors. Note: the data vector generated by the sequential and the parallel versions does not need to be initialised with the same values, i.e. in both cases, the data 11 vector has to be randomly generated with positive numbers but not necessarily in the same way. Analyse the scalability of the new parallel code by looking at the two speed–up plots generated when submitting the submit-strong-omp.sh script. Reason about the new performance obtained with support of Paraver timelines.

elapsed.txt

0

2.508894

1

0.904477

2

0.494514

3

0.348657

4

0.281740

5

0.257242

6

0.253986

7

0.254919

8

0.266507

9

0.283258

10

0.306118

11

0.328576

12

0.353608

13

0.376454

14

0.396199

1. multisort-omp-leaf.c:

elapsed-global

1 5.88

2 3.55

3 2.62

4 2.14

5 1.89

6 1.73

7 1.59

8 1.46

9 1.38

10 1.32

11 1.27

12 1.23

13 1.19

14 1.16

15 1.13

16 1.10

17 1.08

18 1.07

19 1.07

20 1.03

elapsed-multisort

1 5.163486

2 2.834762

3 1.891487

4 1.416238

5 1.169785

6 1.007540

7 0.865780

8 0.739411

9 0.663943

10 0.607201

11 0.553112

12 0.508106

13 0.469736

14 0.437216

15 0.408081

16 0.383461

17 0.361463

18 0.342256

19 0.339684

20 0.311785

2. multisort-omp-tree.c:

elapsed-global:

1 5.98

2 3.61

3 2.66

4 2.19

5 1.93

6 1.76

7 1.62

8 1.48

9 1.41

10 1.35

11 1.30

12 1.25

13 1.21

14 1.17

15 1.14

16 1.13

17 1.09

18 1.07

19 1.06

20 1.04

elapsed-multisort:

1 5.265662

2 2.898653

3 1.945099

4 1.474251

5 1.210522

6 1.037771

7 0.898143

8 0.768296

9 0.695904

10 0.635983

11 0.578550

12 0.531398

13 0.490967

14 0.457027

15 0.426889

16 0.409402

17 0.378477

18 0.358242

19 0.340259

20 0.324148

3. multisort-omp-tree-cutoff.c:

elapsed-global:

1 5.88

2 3.63

3 2.65

4 2.19

5 1.92

6 1.76

7 1.62

8 1.48

9 1.40

10 1.35

11 1.29

12 1.24

13 1.20

14 1.17

15 1.14

16 1.11

17 1.09

18 1.08

19 1.06

20 1.04

elapsed-global (submit-cutoff-omp):

1 5.88

2 3.55

3 2.63

4 2.15

5 1.90

6 1.73

7 1.59

8 1.47

9 1.40

10 1.34

11 1.29

12 1.24

13 1.20

14 1.17

15 1.14

16 1.11

17 1.09

18 1.07

19 1.05

20 1.04

elapsed-multisort:

1 5.168379

2 2.911083

3 1.925251

4 1.458574

5 1.198125

6 1.037147

7 0.891341

8 0.755605

9 0.684878

10 0.626214

11 0.569596

12 0.522620

13 0.482881

14 0.449852

15 0.420276

16 0.394880

17 0.371974

18 0.352451

19 0.334890

20 0.319687

elapsed-partial (submit-cutoff-omp):

1 5.170998

2 2.839592

3 1.919775

4 1.435348

5 1.184430

6 1.015175

7 0.871294

8 0.759533

9 0.683788

10 0.626131

11 0.570291

12 0.523152

13 0.483180

14 0.449719

15 0.420863

16 0.394643

17 0.377473

18 0.353191

19 0.335371

20 0.319160

4. multisort-omp-tree-cutoff-depend.c:

elapsed-global

1 5.87

2 3.47

3 2.56

4 2.10

5 1.87

6 1.70

7 1.57

8 1.45

9 1.38

10 1.32

11 1.27

12 1.22

13 1.19

14 1.15

15 1.12

16 1.10

17 1.08

18 1.06

19 1.04

20 1.03

elapsed-multisort

1 5.165600

2 2.759719

3 1.847063

4 1.388333

5 1.150367

6 0.987310

7 0.854582

8 0.734766

9 0.665457

10 0.607774

11 0.553160

12 0.508490

13 0.468750

14 0.436806

15 0.407603

16 0.383573

17 0.360686

18 0.342709

19 0.325085

20 0.310682