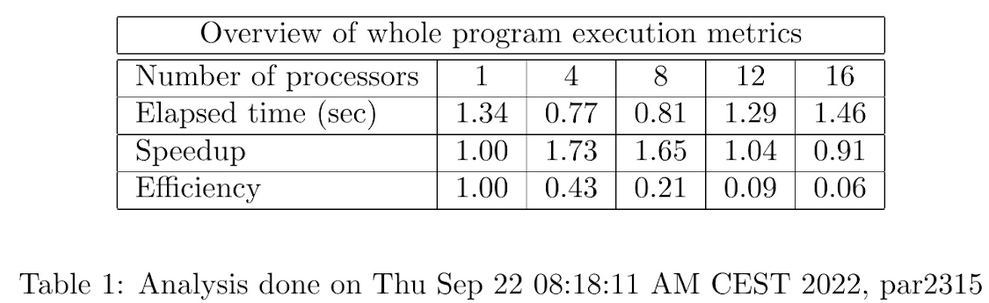
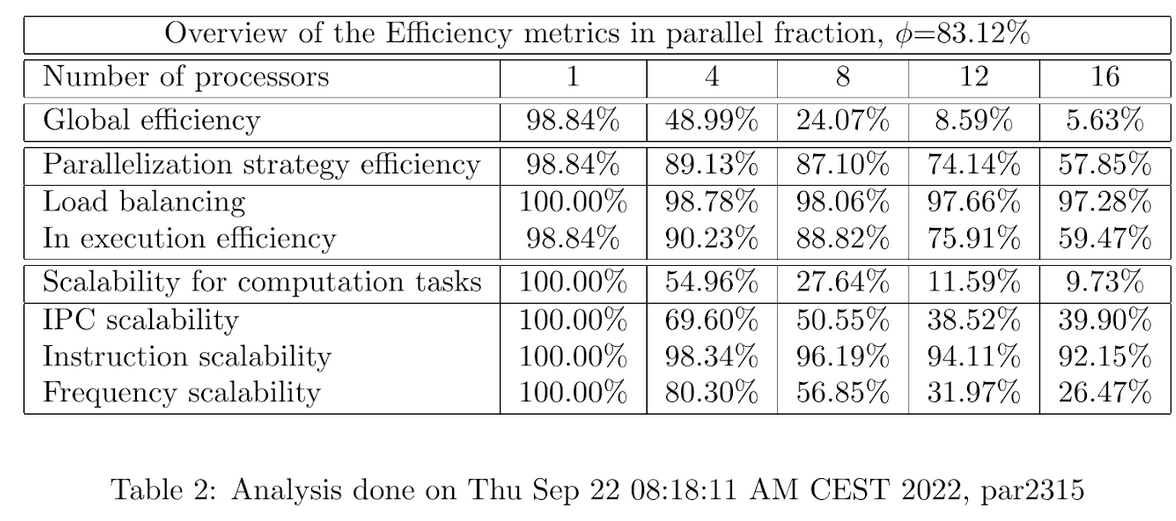
4.1 Discovering modelfactors: Overall analysis

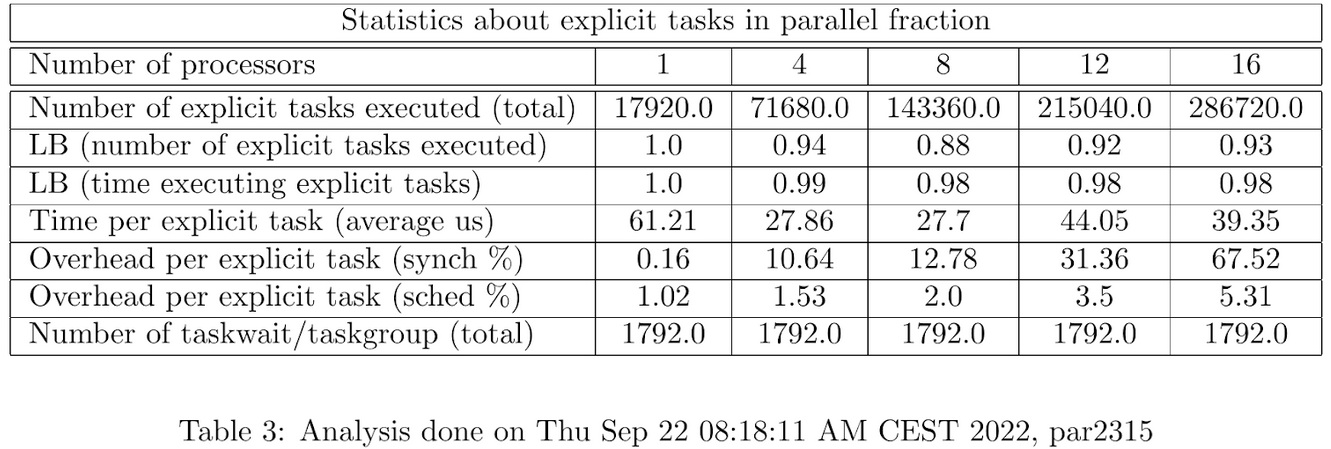
After a deep analysis of the modelfactors metrics we have reached some conclusions. The first one being that the scalability was really bad, as we can see in table 1, after the four threads execution the elapsed times just go worse and worse when adding more threads, even having a slower execution with sixteen threads than the sequential one.

The second conclusion was deduced from the first one, as we realized that the overheads due to synchronization are not negligible, as when the number of threads increased the synchronization time increased proportionally.

Then we computed the parallel fraction of the program using the formula:

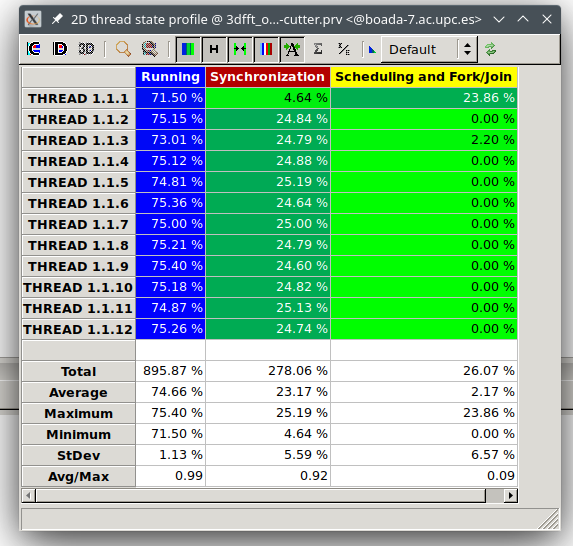
Once we knew the parallel fraction of the program we realized that the efficiency for the parallel regions is not quite right as we can see in the next figure:

After seeing the huge difference between the expected efficiency and the real one we decided to find out the factor that was most important in terms of efficiency loss, and we concluded that that factor was the synchronization time, that represents, in the worst case a 67% of the whole execution time as we can see in the third figure:



4.2 Discovering Paraver (Part I): execution trace analysis

As we stated before, the overhead problem increases in function of the number of threads going from just a 0.16% with one thread to a 67% with sixteen threads. We find this gradual increase justified as the number of tasks increase proportionally to the number of threads but the amount of work done in each task is less and less, so in the end there are a huge amount of little tasks that we need to synchronize while the work done by each of them is small despite having the same synchronization time.



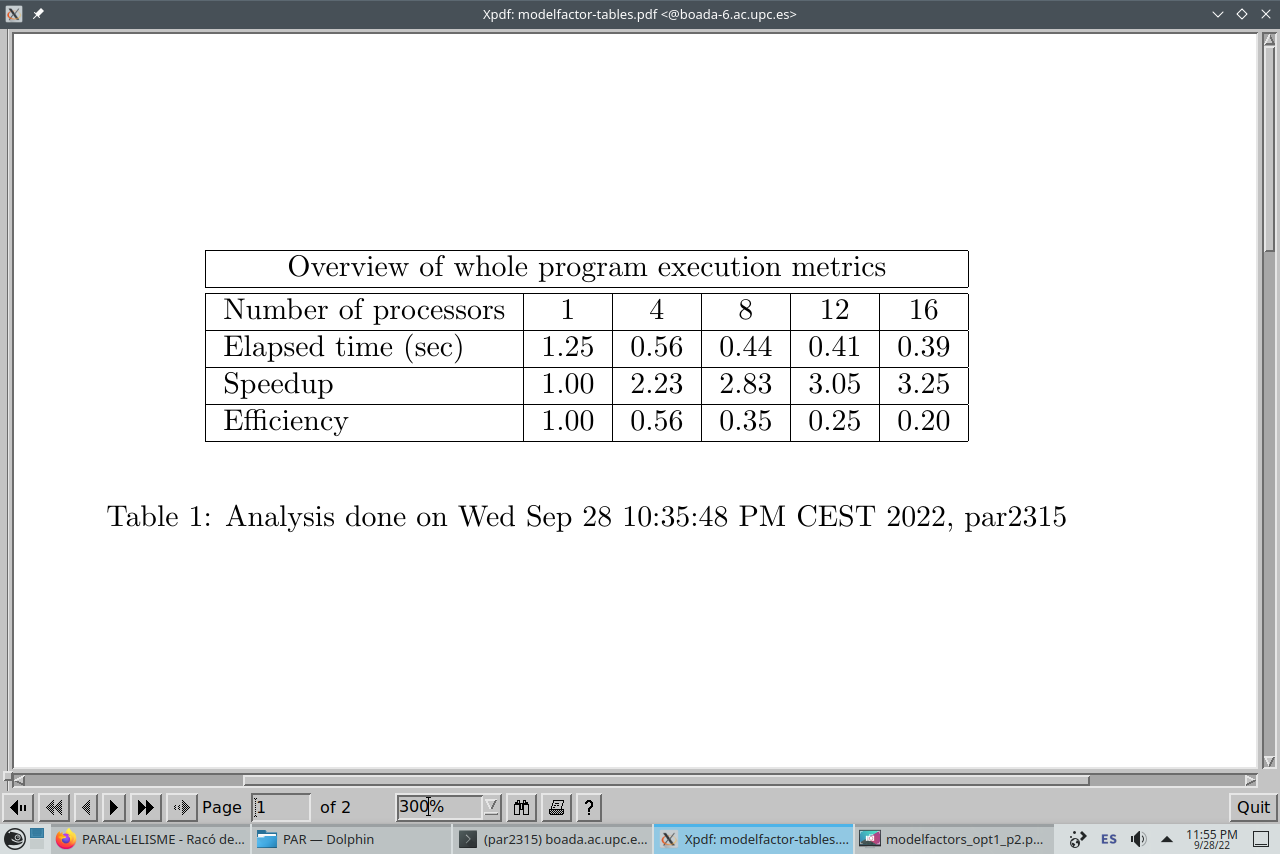
4.3 Discovering Paraver (Part II): understanding the parallel

execution

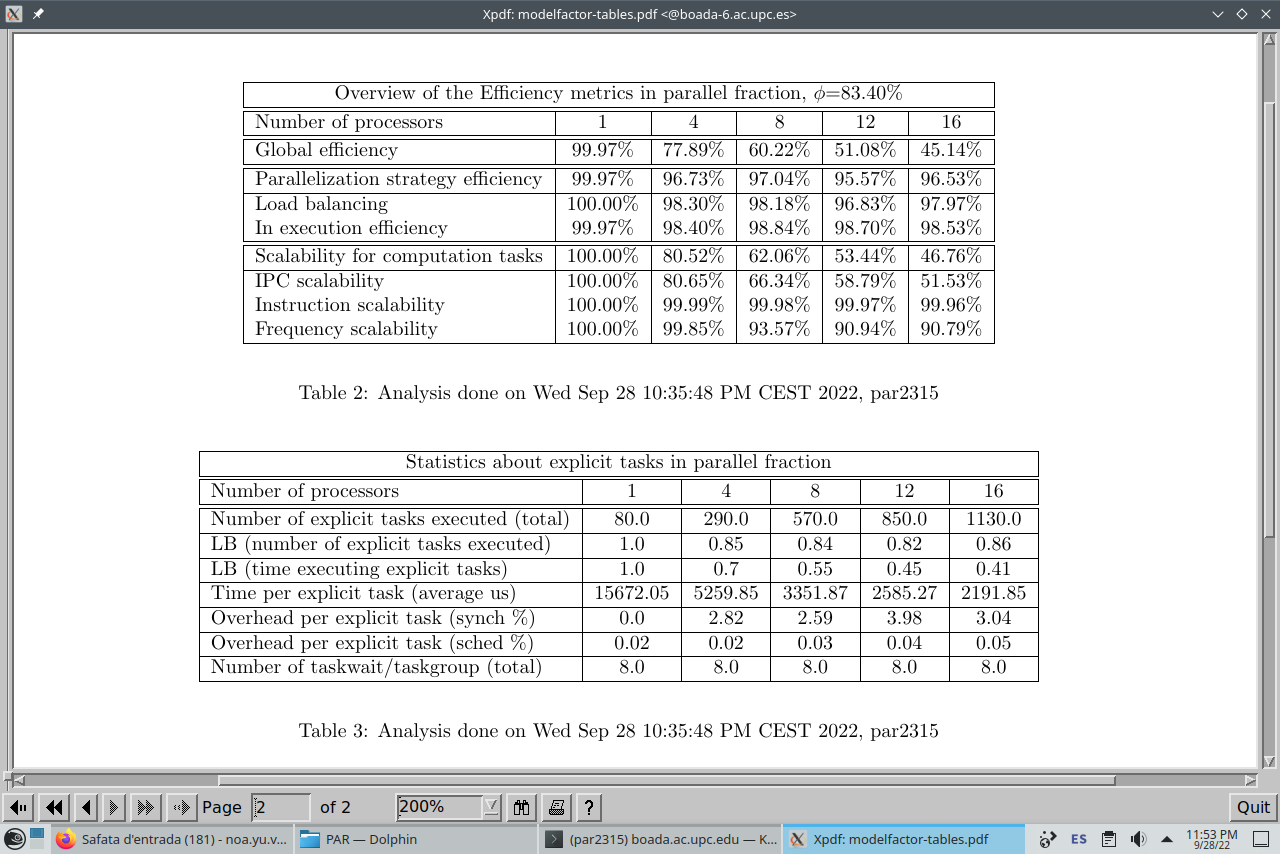
4.3.1 Reducing Parallelisation Overheads and Analysis

After reviewing the histogram and timeline and as we were expecting from the previous results, we have concluded that the granularity of this program is really fine, or in other terms, there are a huge number of relatively small tasks.

Then we have analyzed the modelfactor output and from that we have drawn the following conclusions. The overall performance has improved in a highly sensible way, and can be seen especially in the speedup of table 1, which not only has not decreased as before, but now is more than three times higher than the sequential time. There are no slowdowns as there were before.

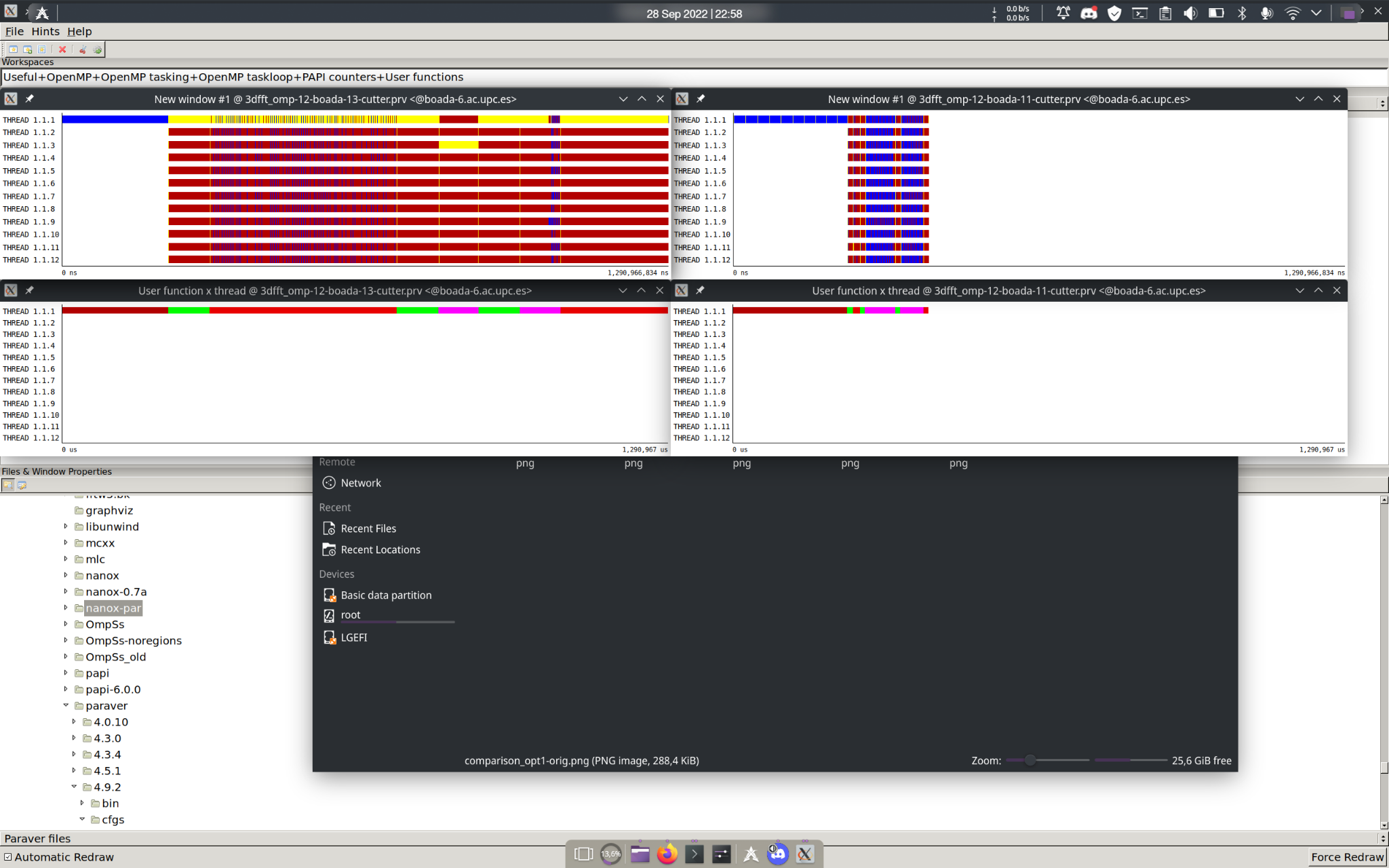
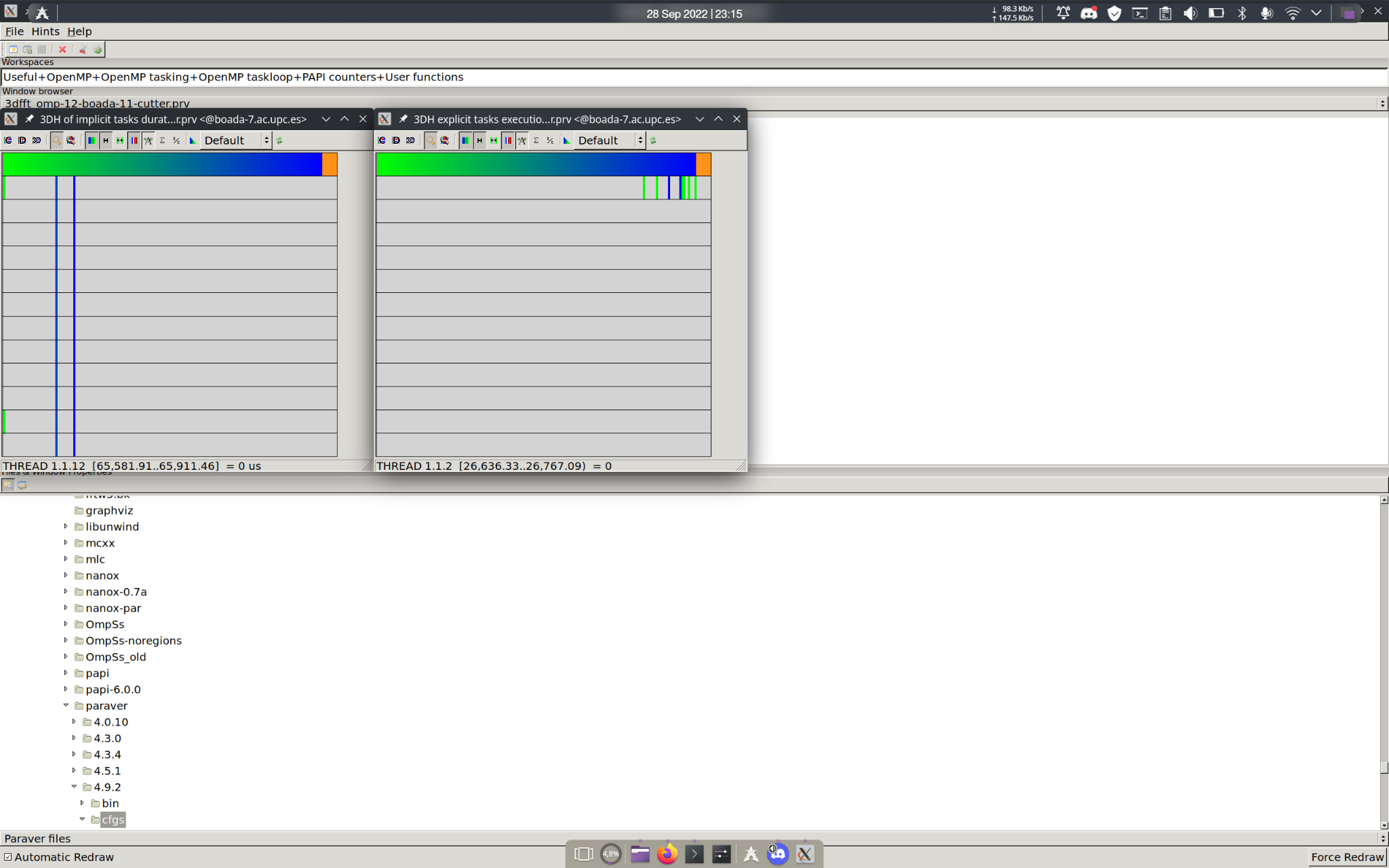


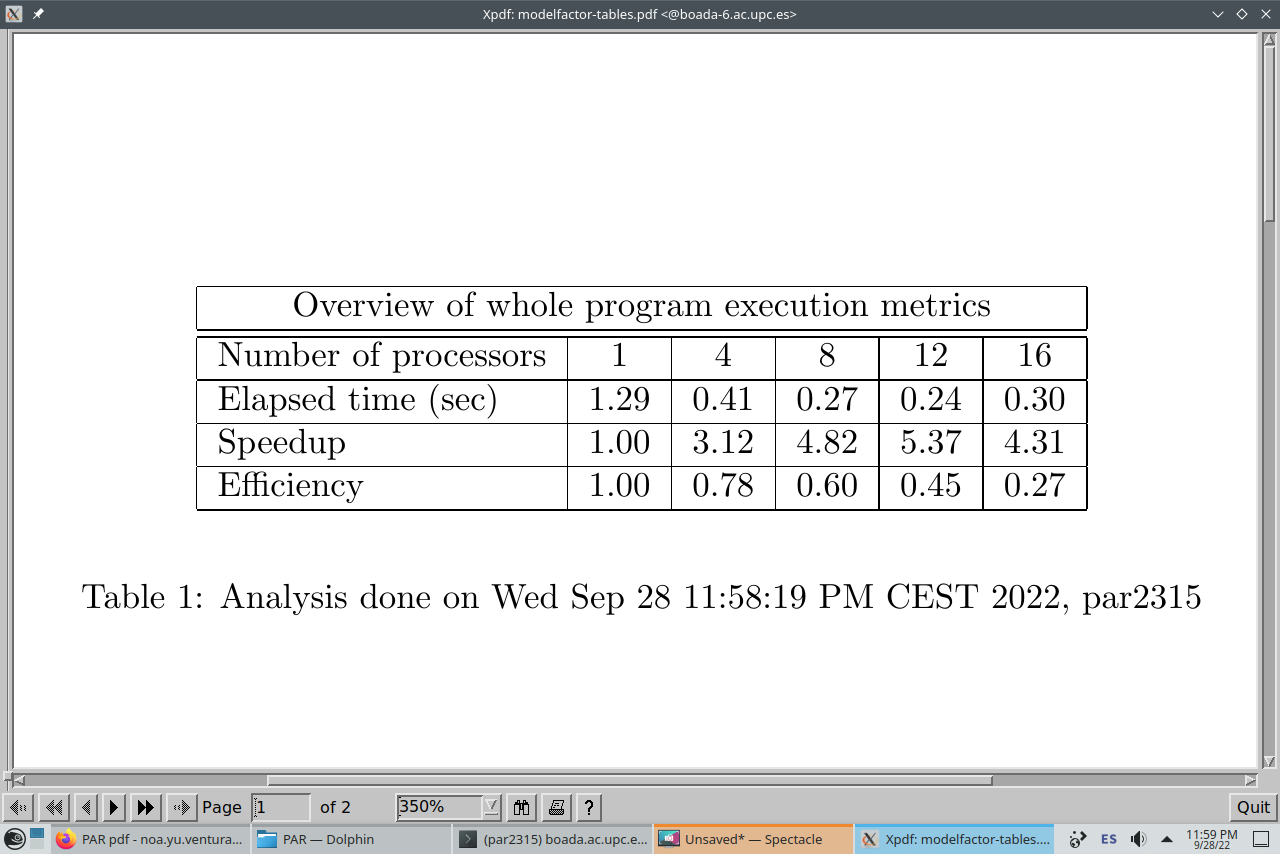
In the overheads is where we find the key of this efficiency improvement, as what before was a 67% and a 5% of synchronization and scheduling times respectively now these times have reduced to a 3% and a 0.05% each one of them as we can see in table 3.



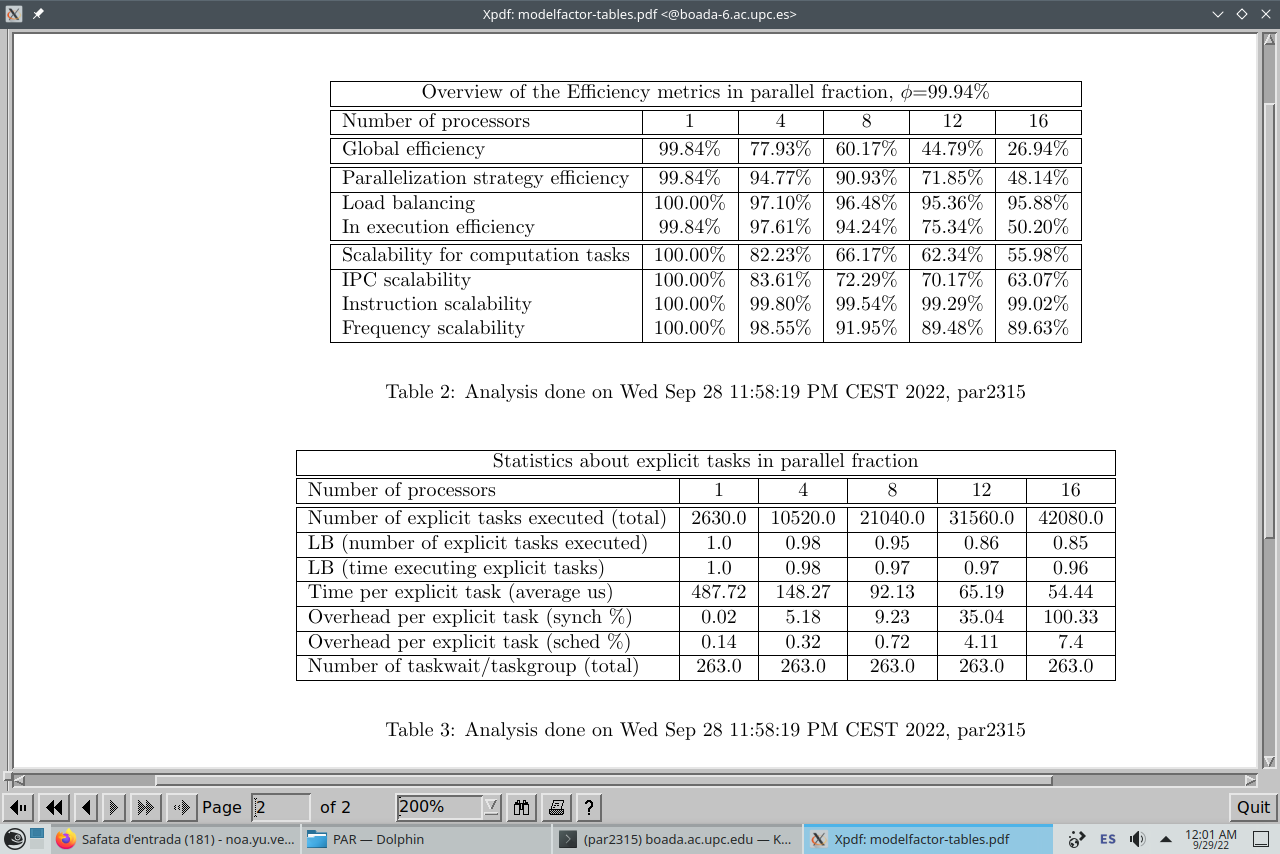
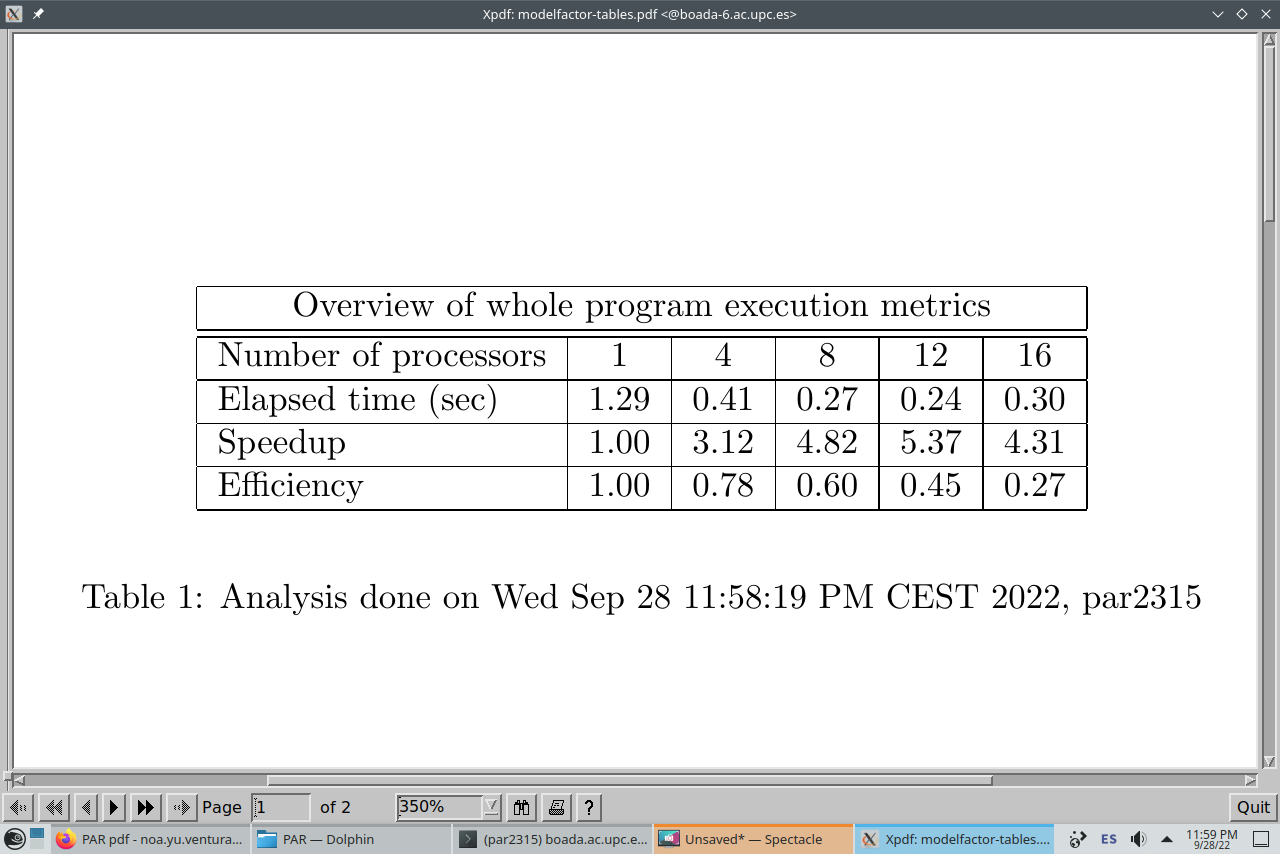
We have obtained that the parallel code section represents the 73.4% of the code, then aplying the amdahl law:

Which tells us that the maximum speedup achievable with infinite processors would be of 3.76, and the one we have attained with 16 processors is 3.25 is quite close to that maximum, so as it has an asymptotically behavior, the amount of speedup gained while increasing the amount of processors reduced proportionally.

4.3.2 Improving φ and Analysis



Analyzing the last modelfactor output we can see that the function that limits the speedup the most is the synchronization overheads, as in the 16 core execution represents a 100.33% of the task time. All the functions are parellized, the only parts that are not parallized are the main code part which represents about 12.2% of the code.



| Version |  | ideal |  |  | real |
| --- | --- | --- | --- | --- | --- |
| initial version in 3dfft\_omp.c | 0.54 | 2.17 | 1.34 | 1.29 | 1.04 |
| new version with reduced parallelisation overheads | 0.73 | 3.7 | 1.25 | 0.41 | 3.05 |
| final version with improved | 0.88 | 8.33 | 1.29 | 0.24 | 5.37 |

As we can see in the two graphs on the top, there is a non-parallelized code at the beginning of the execution, which corresponds to the init\_complex\_grid function, but once we parallelize this function, we see a big improvement on the execution time of the last version (the graph at the bottom). In the original code we have a lot of small tasks because the tasks are defined on the inner loops, so there are a lot of context switches and that’s why synchronization takes so much of the execution time. In the first and the second optimized versions the synchronization time greatly decreases because the granularity of the tasks is bigger and the number of tasks is reduced, which we achieved by uncommenting the outer loops of the other functions except the init\_complex\_grid function.