PAR Laboratory Assignment

Lab 1: Experimental setup and tools

Group 13-03

Carlos Barcenas Holguera

Miquel Gómez i Esteve

### Facultat d’Informàtica de Barcelona - UPC

### March 2018

### Spring 2017-2018

#### Node architecture and memory

|  |  |  |  |
| --- | --- | --- | --- |
|  | **boada-1 to boada-4** | **boada-5** | **boada-6 to boada-8** |
| **Number of sockets per node** | 2 | 2 | 2 |
| **Number of cores per socket** | 6 | 6 | 8 |
| **Number of threads per core** | 2 | 2 | 1 |
| **Maximum core frequency** | 2395 MHz | 2600 MHz | 1700 MHz |
| **L1-I cache size (per-core)** | 32 KB | 32 KB | 32 KB |
| **L1-D cache size (per-core)** | 32 KB | 32 KB | 32 KB |
| **L2 cache size (per-core)** | 256 KB | 256 KB | 256 KB |
| **Last-level cache size (per-socket)** | 12288 KB | 15360 KB | 20480 KB |
| **Main memory size (per socket)** | 12 GB | 31 GB | 16 GB |
| **Main memory size (per node)** | 24 GB | 62 GB | 32 GB |

All the information shown in the previous table has been obtained using lstopo and lscpu commands.

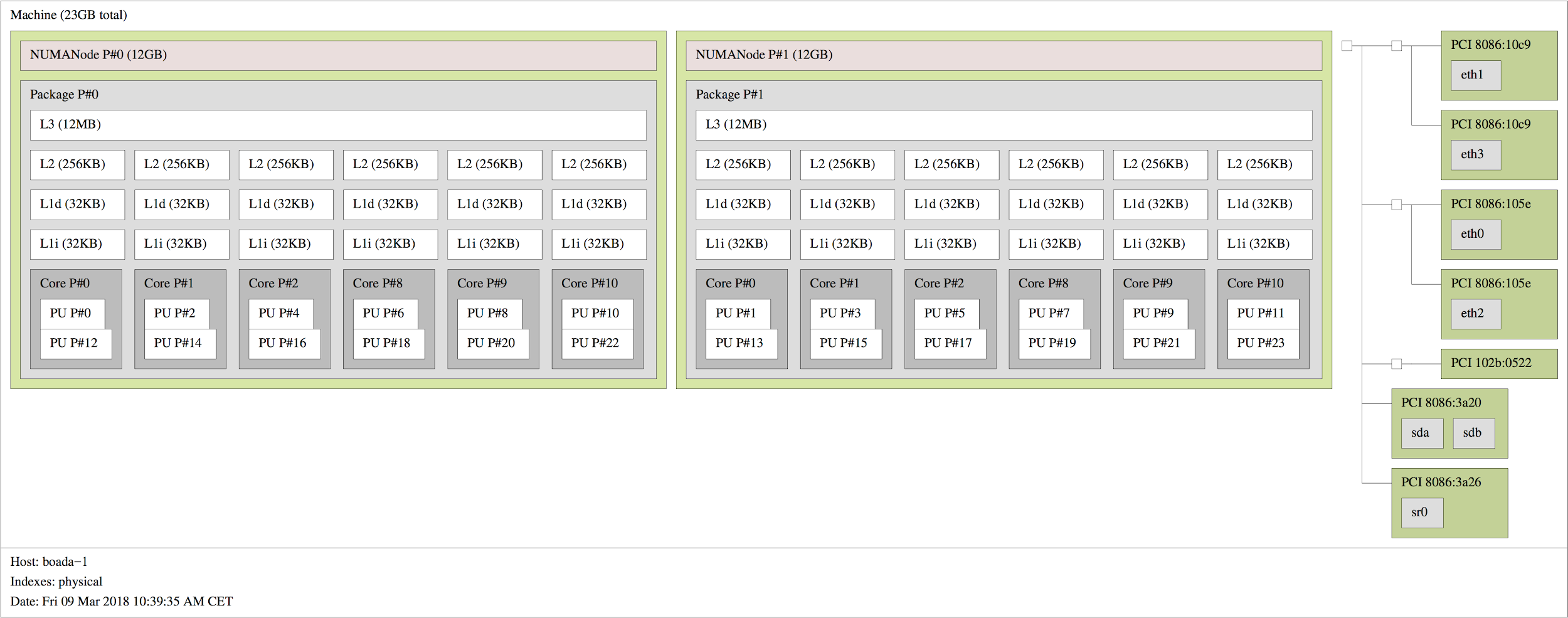
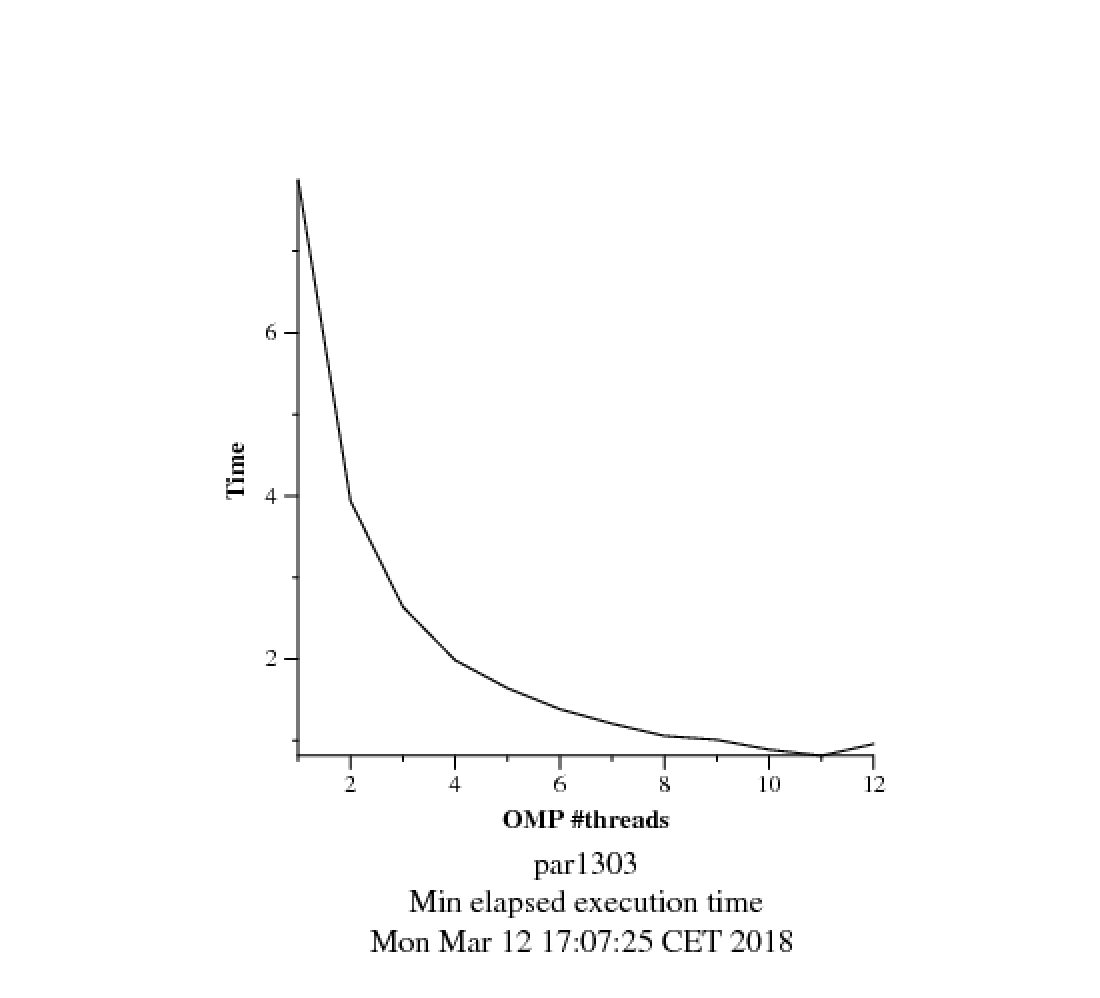
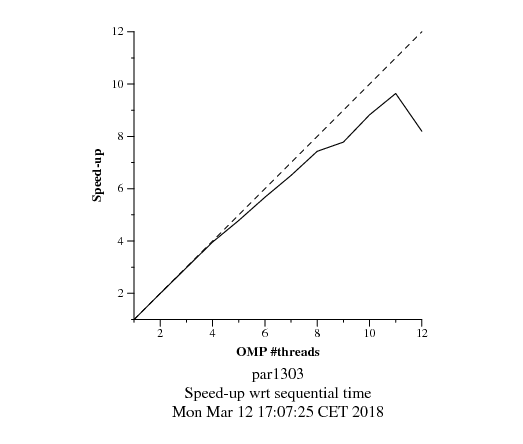


Figure 1.1: Image obtained with lstopo, showing boada-1 to boada-4 architecture.

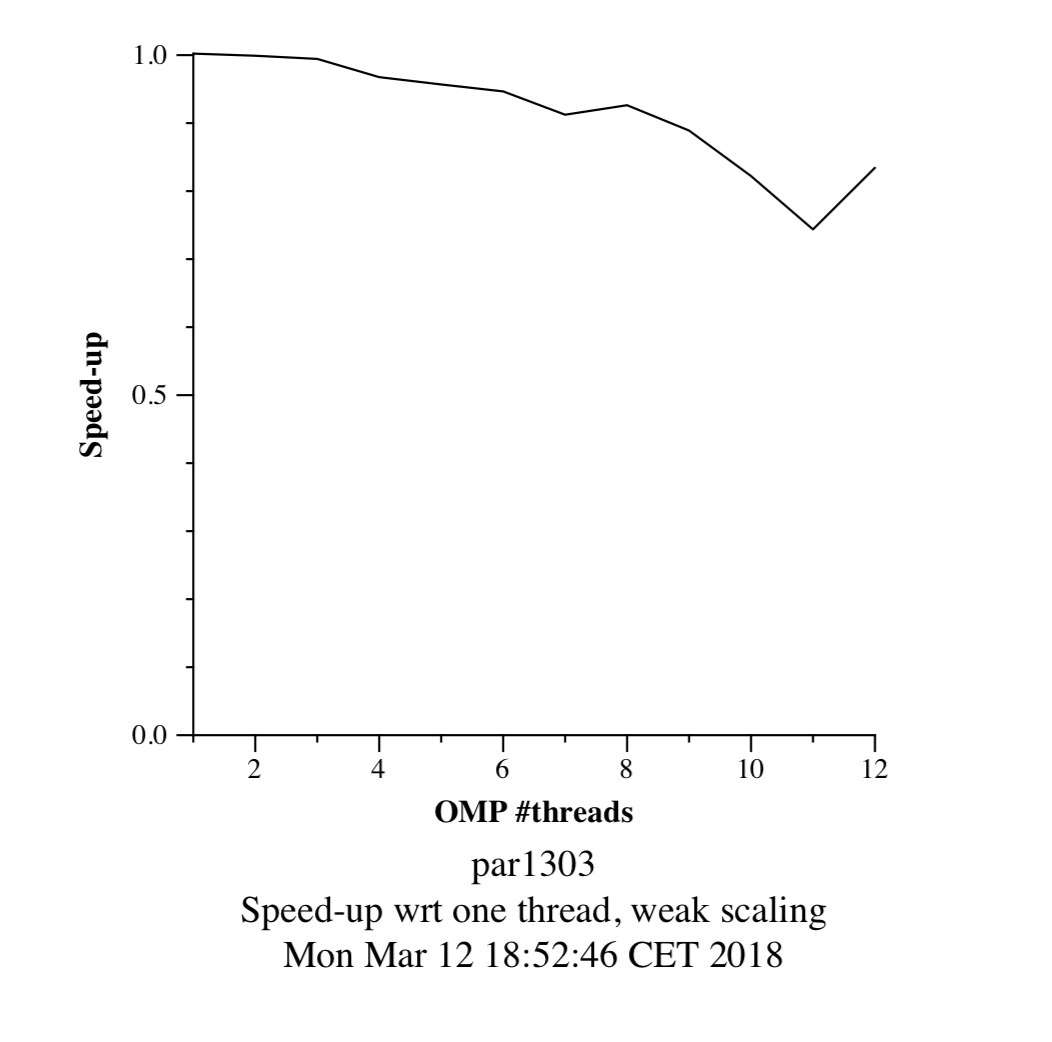
#### 

#### Timing sequential and parallel executions

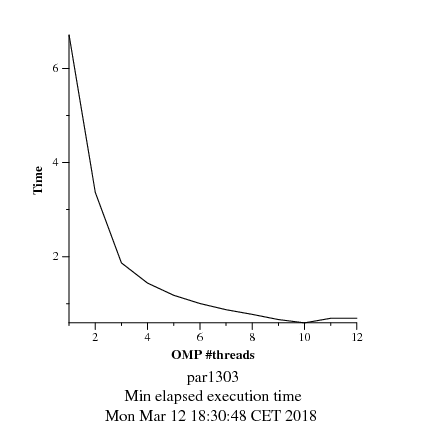
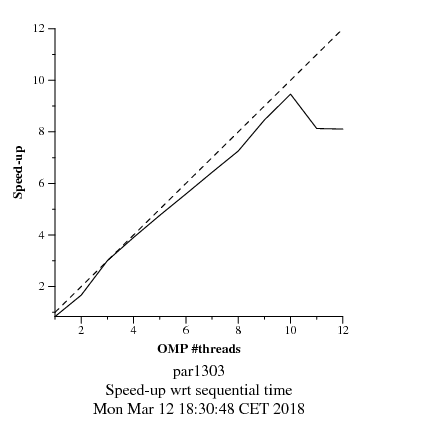
Strong Scalability boada 1-4:



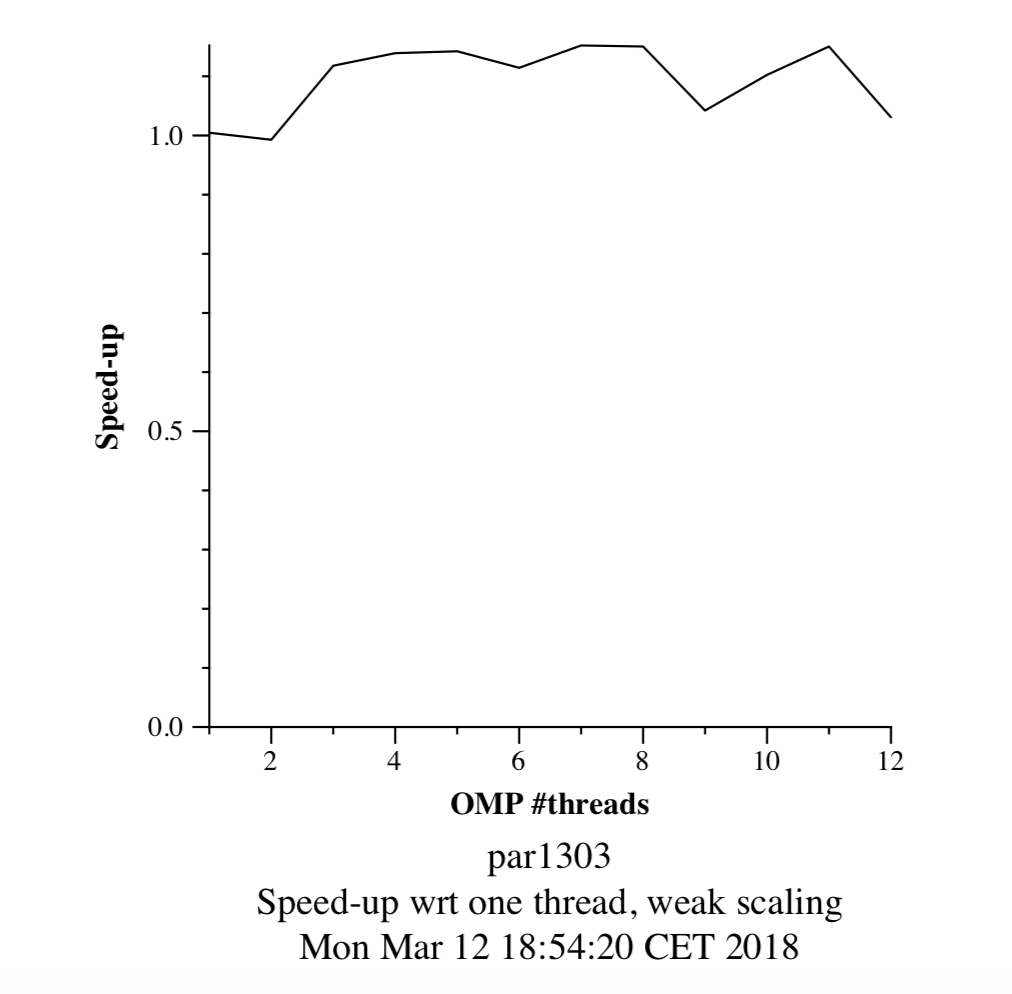
Weak Scalability boada 1-4:



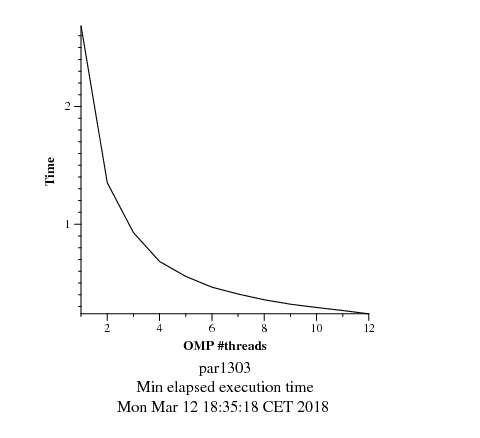
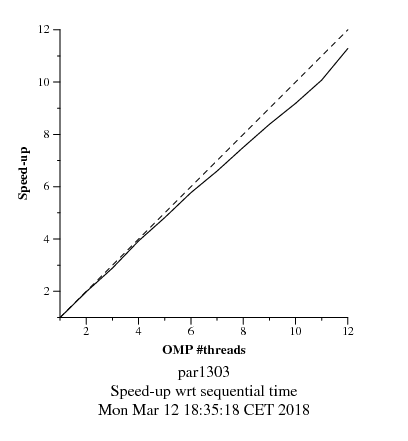
Strong Scalability boada 5:



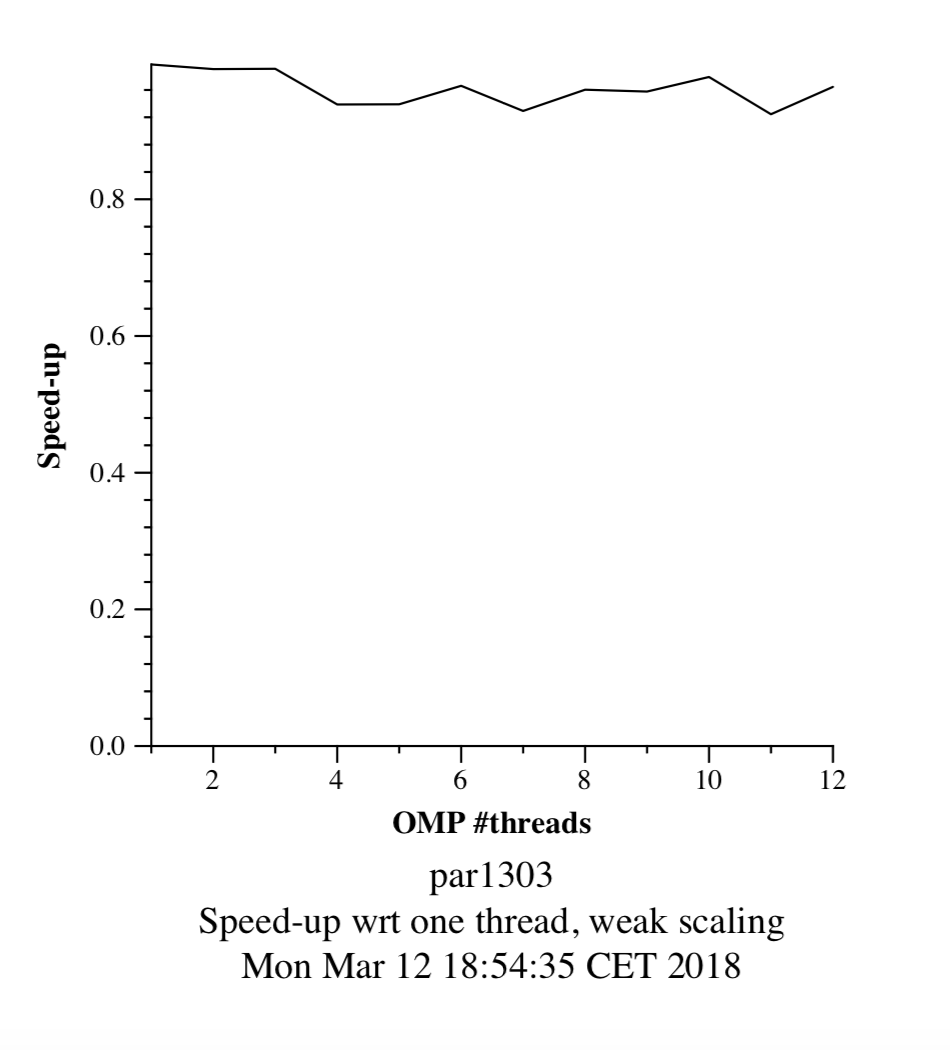
Weak Scalability boada 5:



Strong Scalability boada 6-7:



Weak Scalability boada 6-7:



All the previous graphics have been obtained running the different scripts to check weak and strong scalability of the program which calculates pi on the differents nodes from boada.

Explanation:

We can clearly see that in most cases, while checking the strong scalability of the program, the more threads we have, the less the execution time is. The speed-up of every…...

In some cases when the number of threads reaches a higher number, the execution time doesn’t shorten anymore and rises. That’s because the overhead associated to the task creations and synchronization times gain importance and even if we can have a lower execution time of the tasks themselves we cannot improve the overall time.

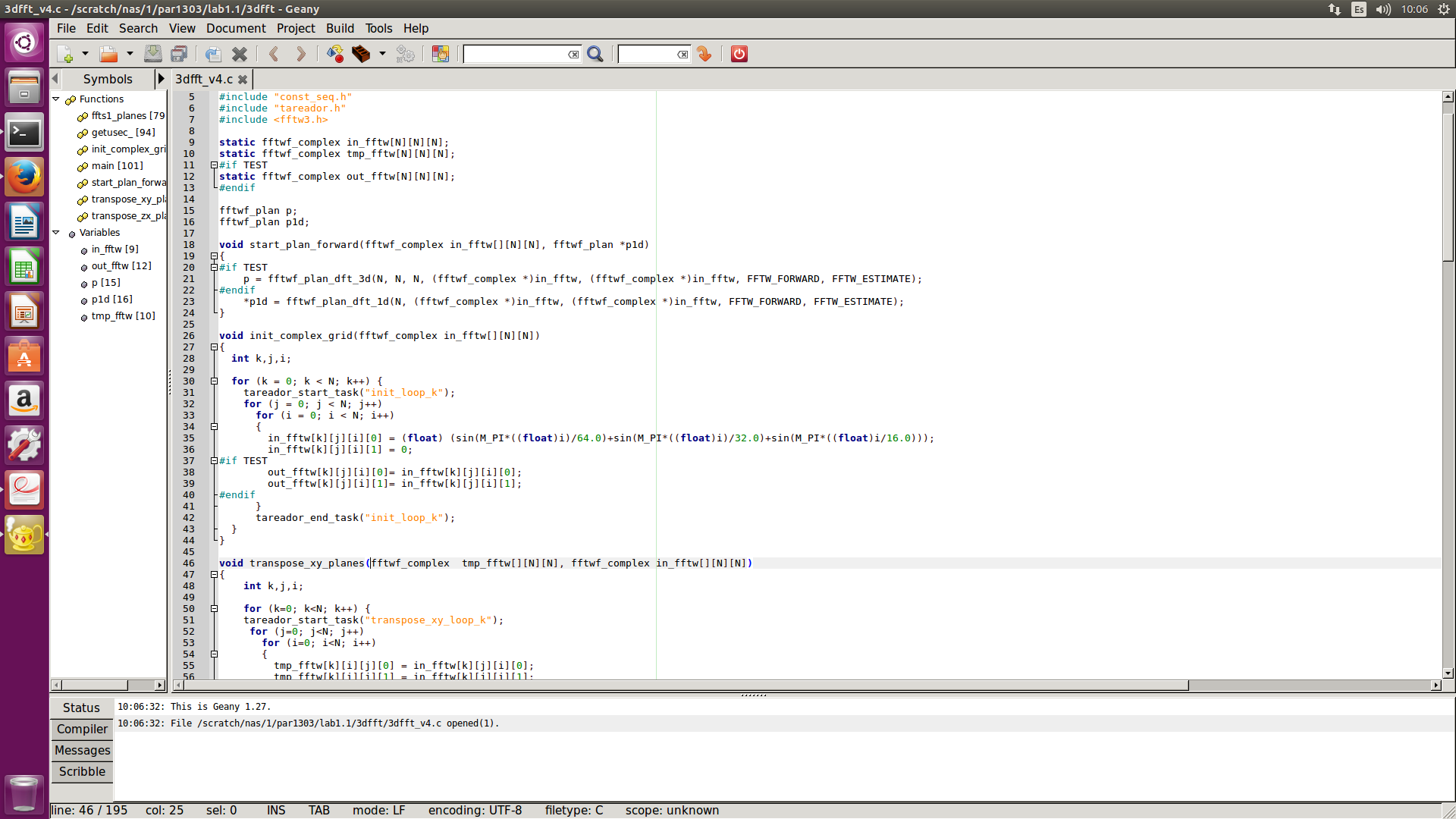
When we check the weak scalability we can see that no matter how big we make our program, the speed-up is always nearly the same with just some small variations. That would allow us to do much more work just by adding more available threads for the program execution.

# 

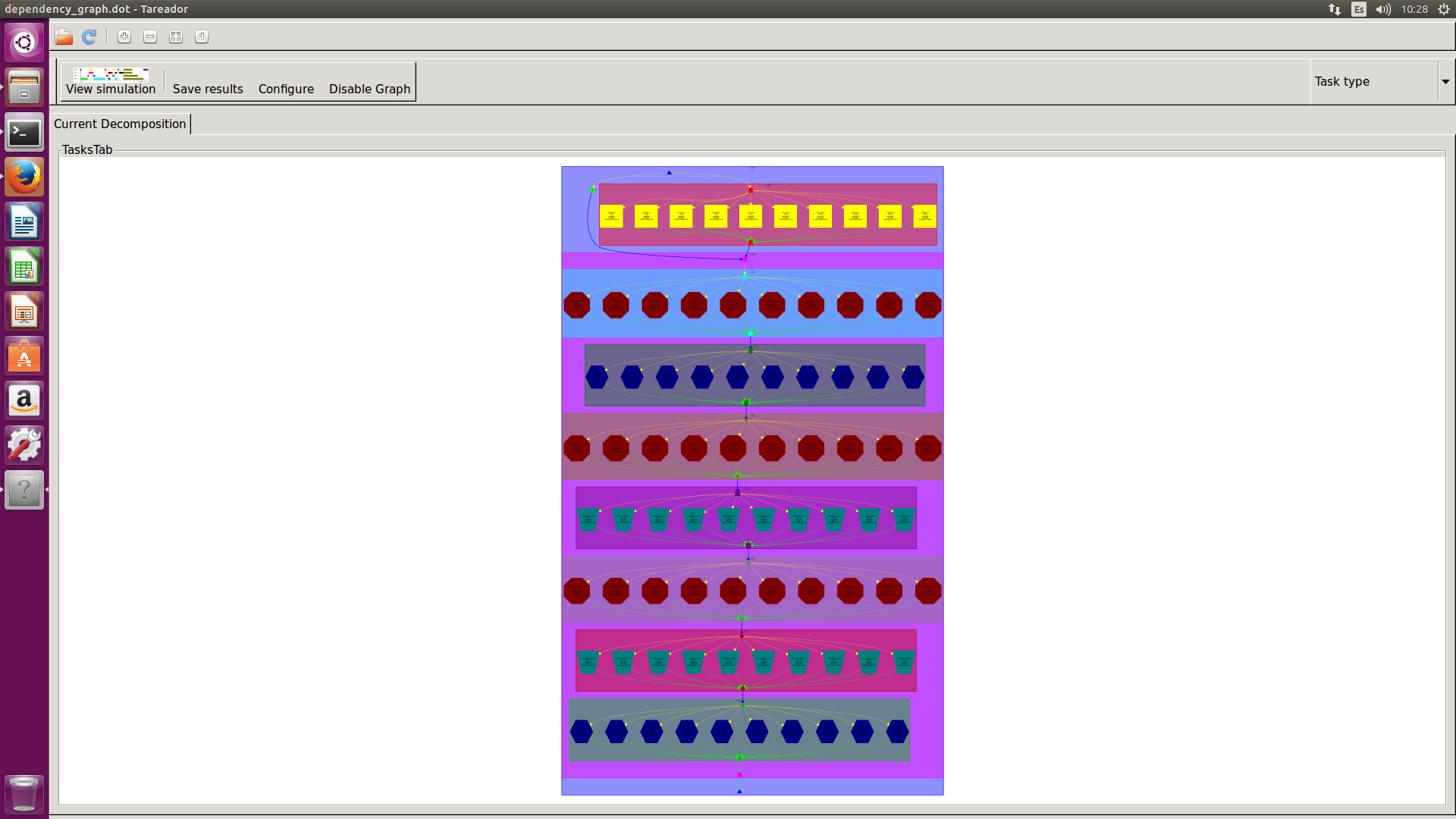
#### 

#### Analysis of task decomposition with *Tareador*

In version 4 we have included this parts to the init function to decompose totally all the tasks in fine-grained tasks.



We have added the tareador\_start\_task and tareador\_end\_task into the loop k of the init function.

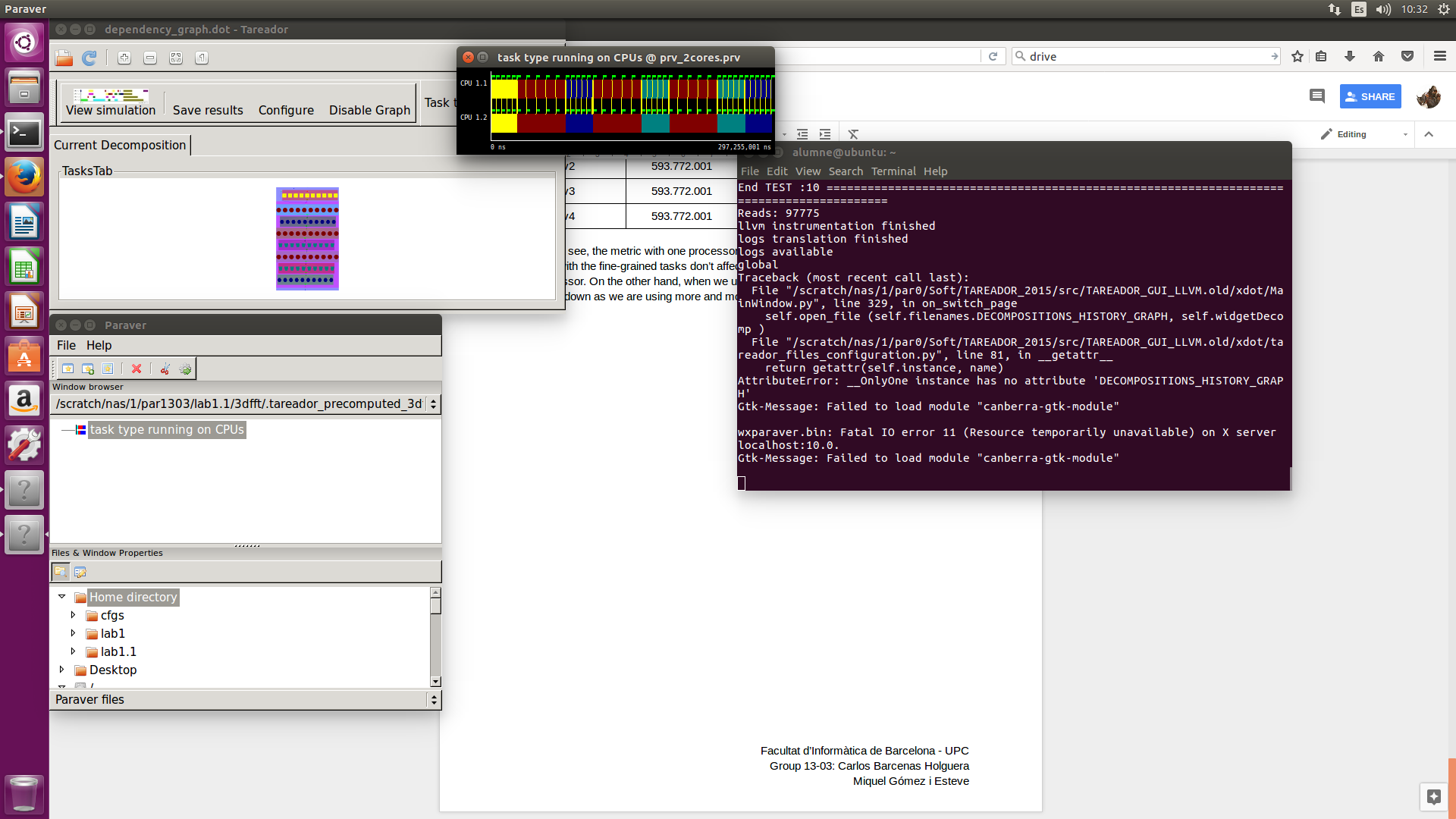


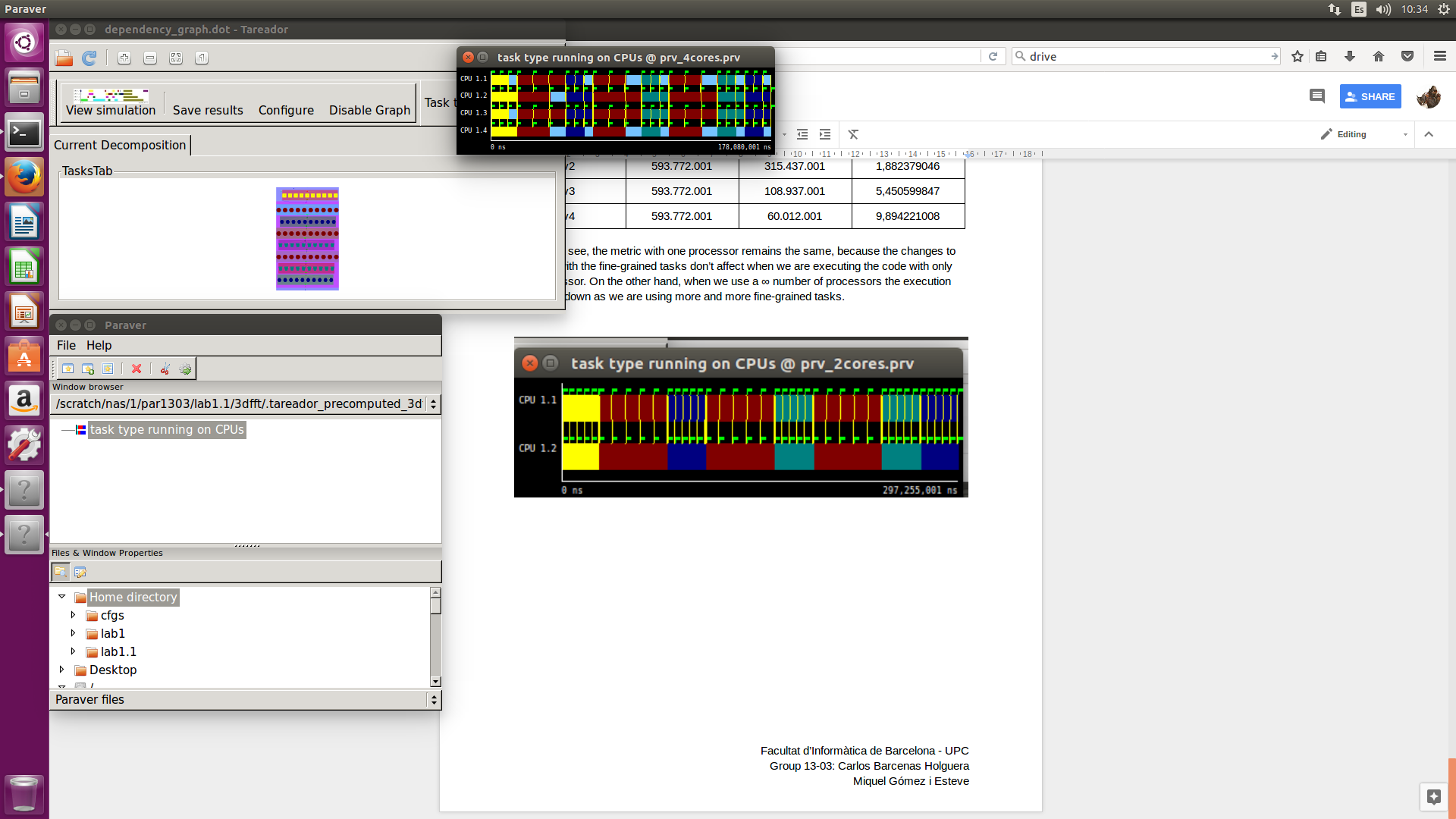
We can see as all the big tasks are fine-grained now.

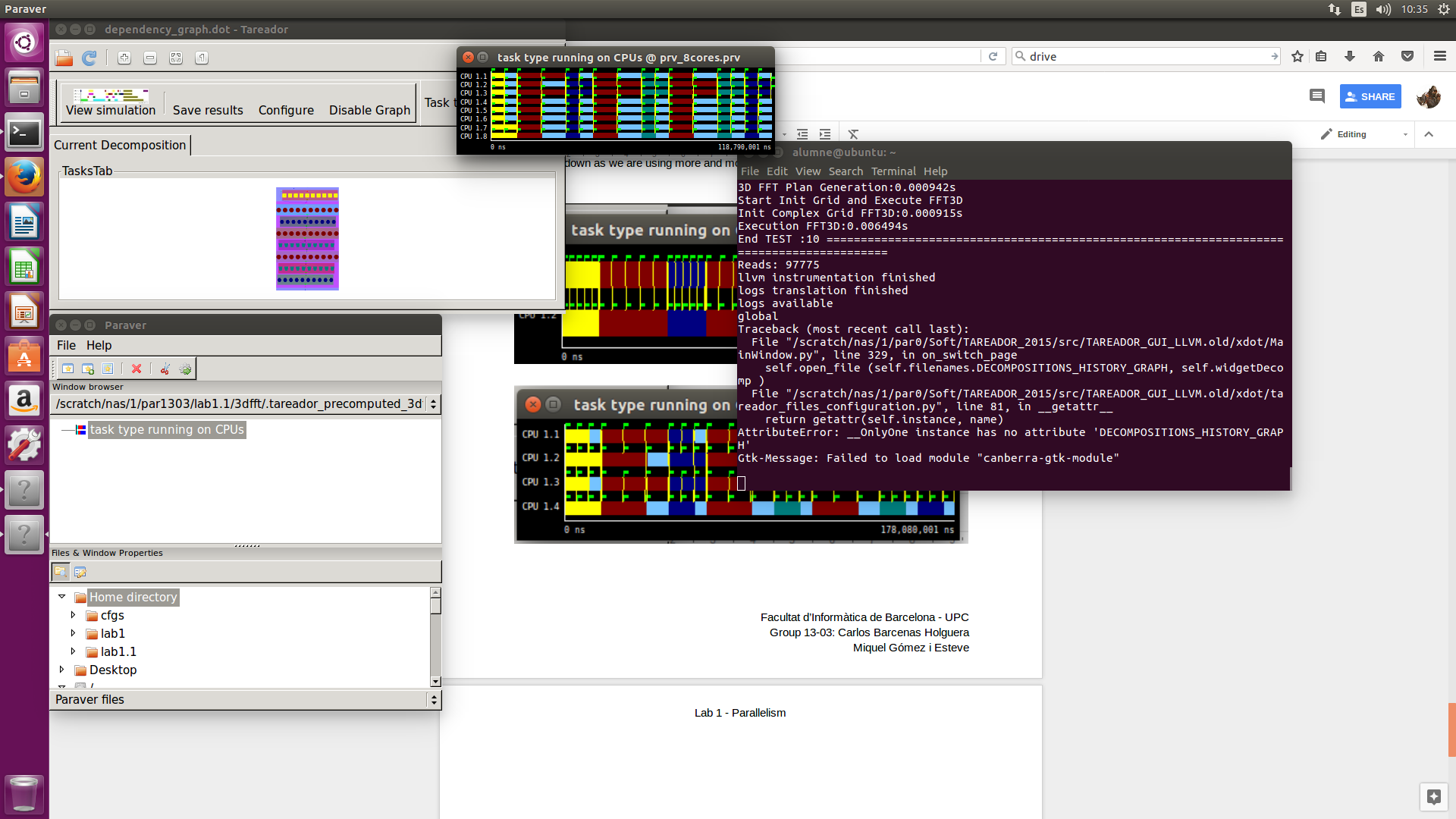
We have defined 4 versions for the 3dfft\_seq program:

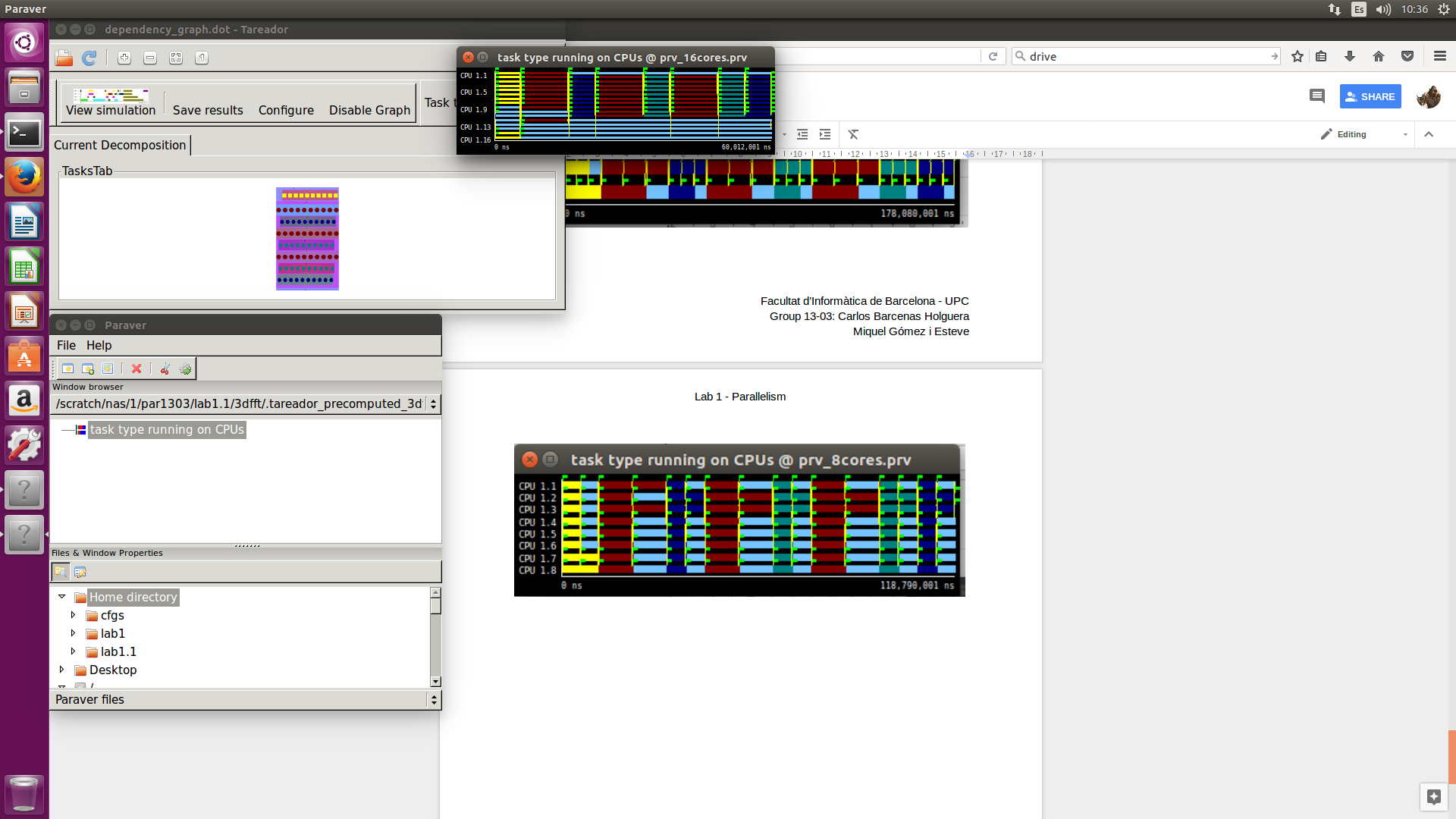
|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **T1** | **T∞** | **Parallelism** |
| seq | 593.772.001 | 593.705.001 | 1 |
| v1 | 593.772.001 | 593.705.001 | 1 |
| v2 | 593.772.001 | 315.437.001 | 1,882379046 |
| v3 | 593.772.001 | 108.937.001 | 5,450599847 |
| v4 | 593.772.001 | 60.012.001 | 9,894221008 |

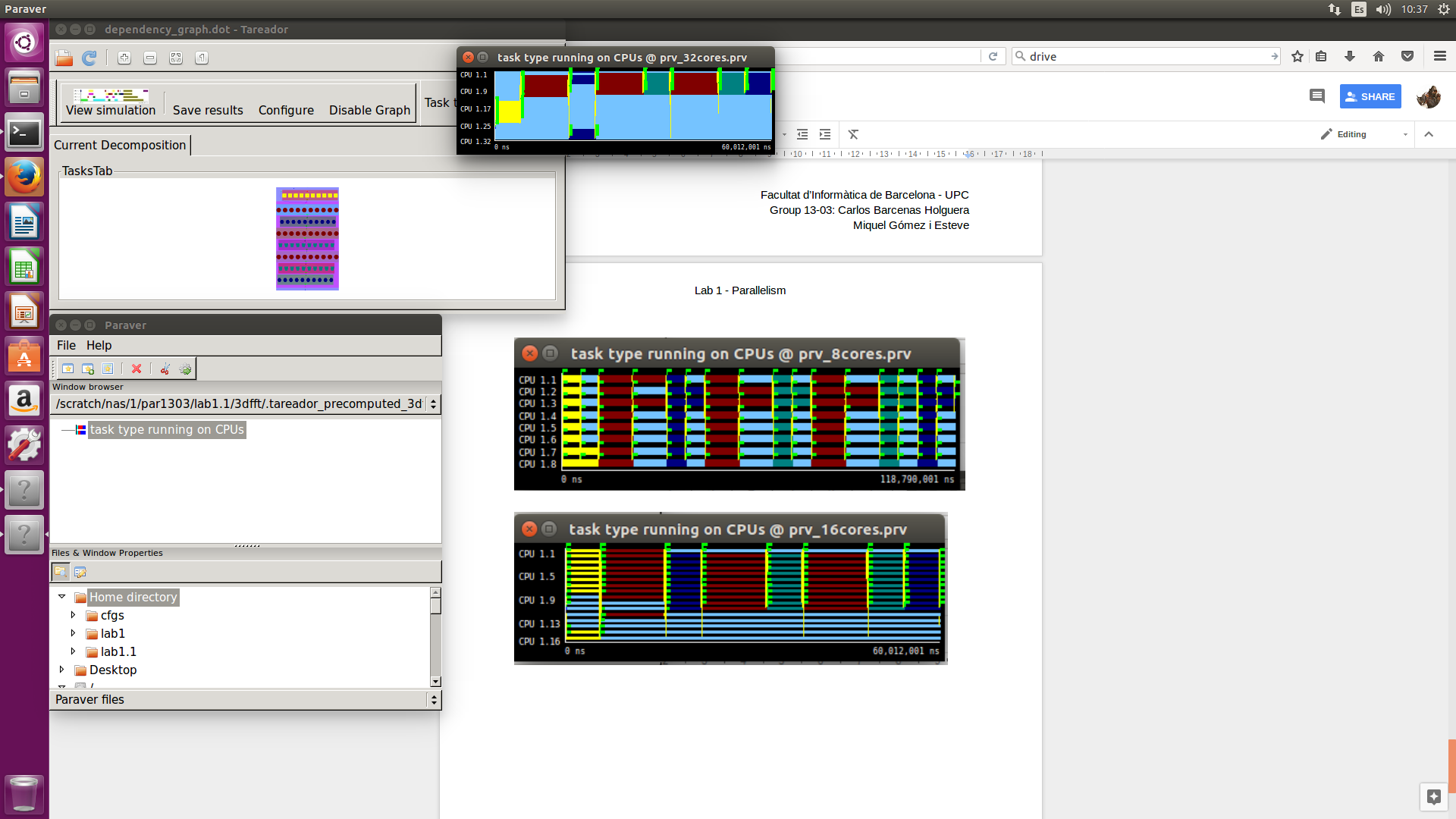
As we can see, the metric with one processor remains the same, because the changes to the code with the fine-grained tasks don’t affect when we are executing the code with only one processor. On the other hand, when we use a ∞ number of processors the execution time goes down as we are using more and more fine-grained tasks.



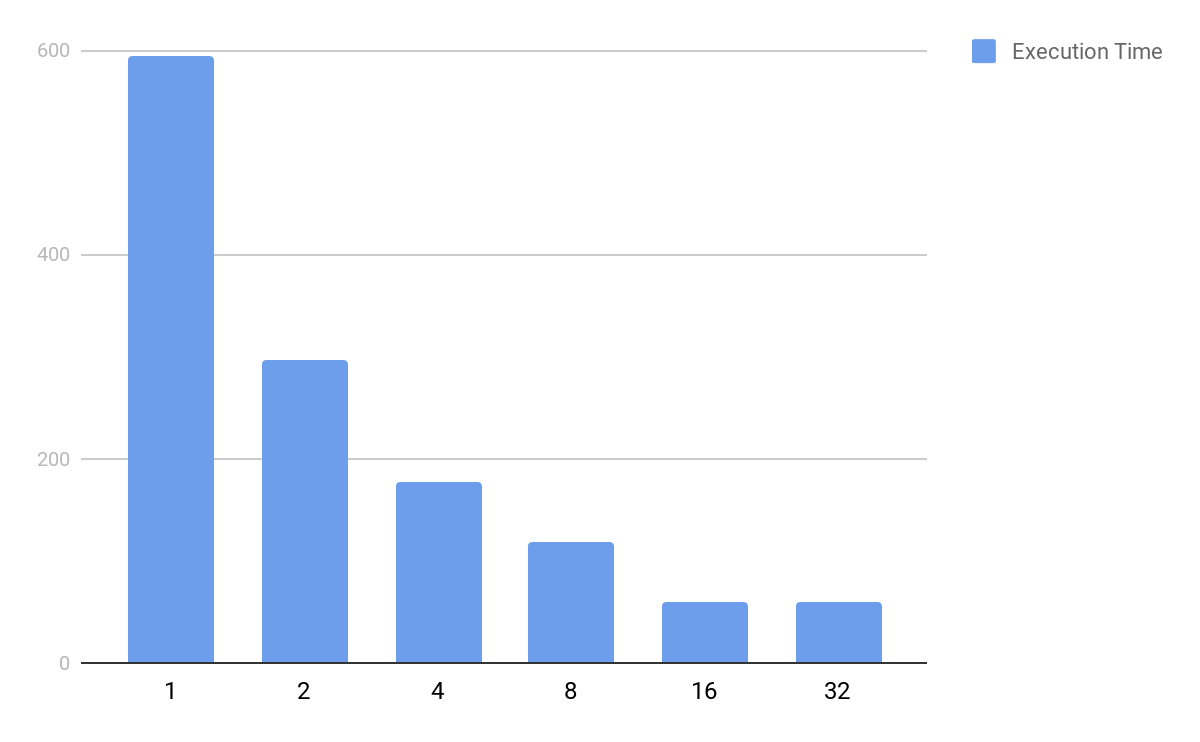


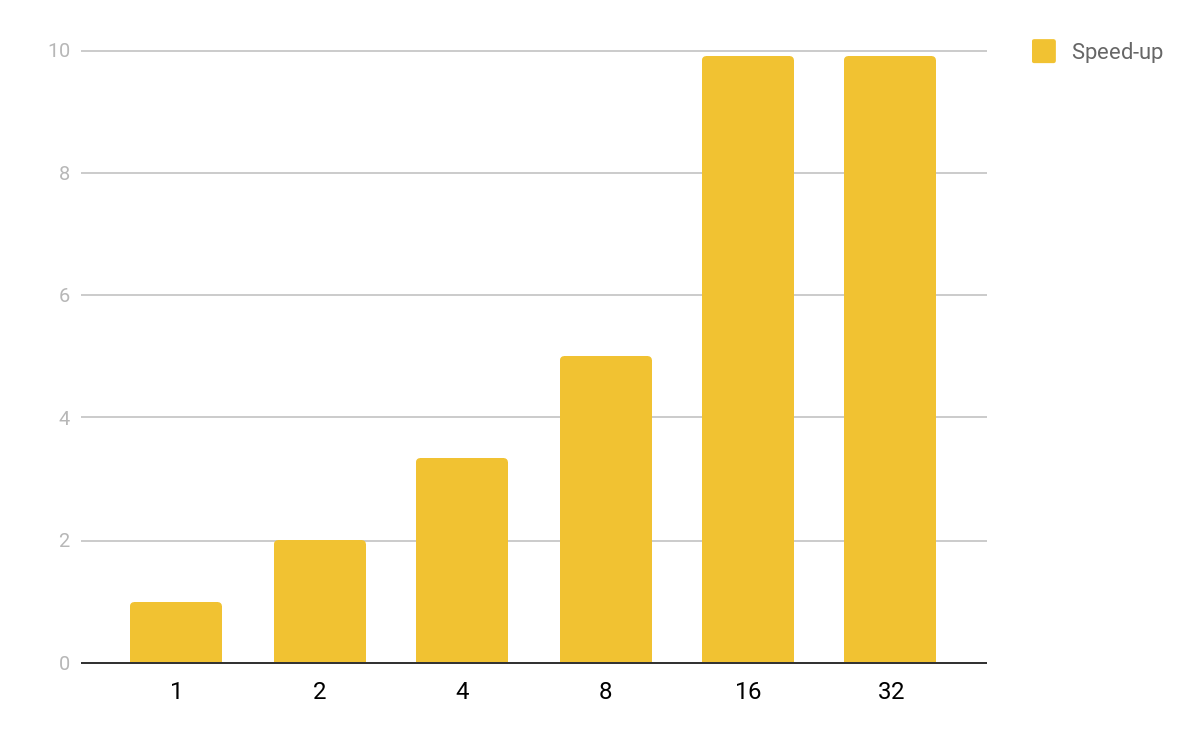






We executed the program 3dfft\_v4 with different amounts of threads, and the next graphics are the results we obtained on their execution time and the speed-up.





We can clearly see the speed-up growth (or the execution time degrowth) while we are adding more threads, however, once we reach 16 threads, we see no difference with the execution with 32 threads (and in some executions it’s even worse, meaning that it has a larger execution time). The reason is, that because of node dependencies it takes a lot of valuable time to create tasks as well as synchronizations and other necessary work.

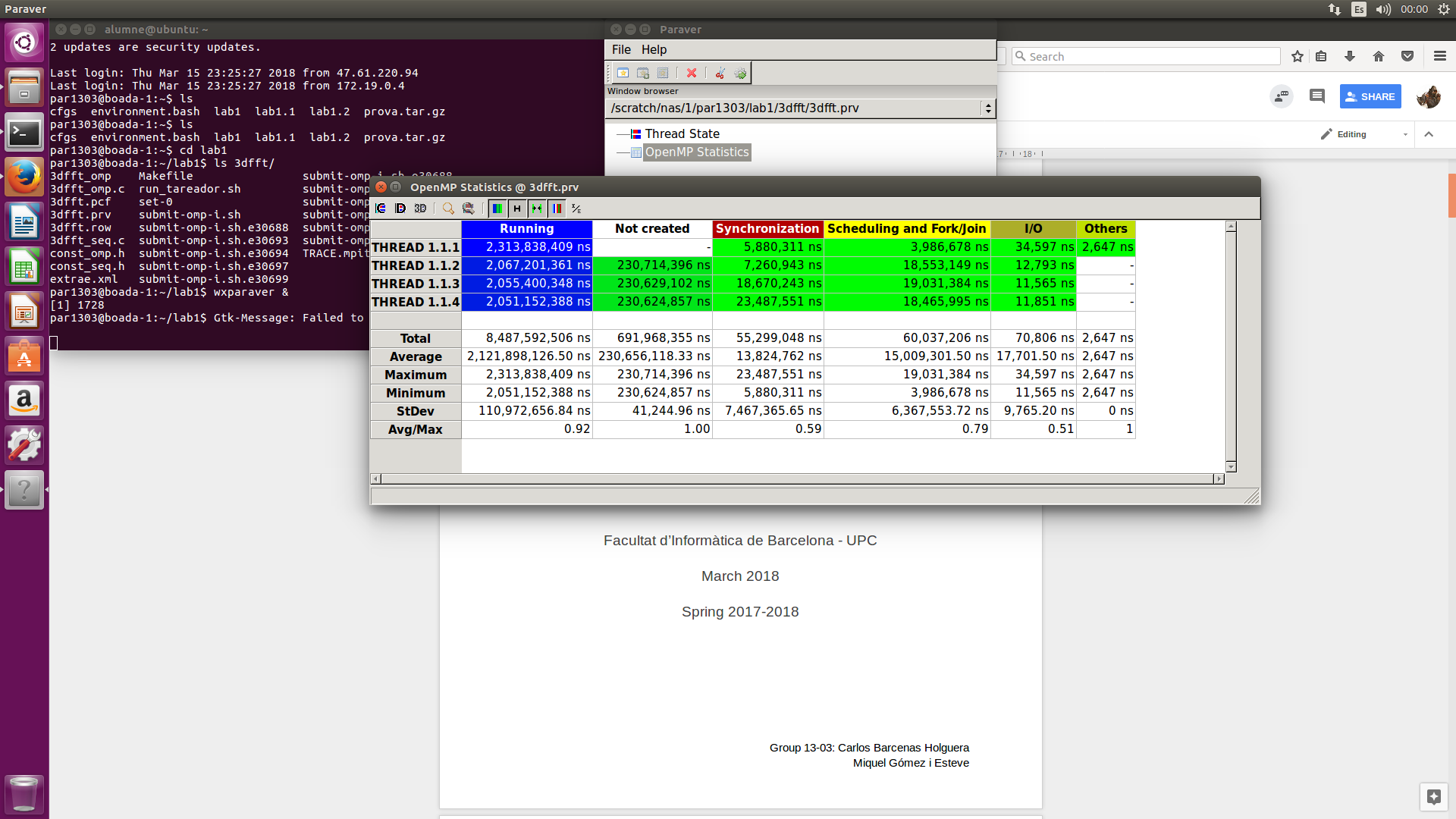
#### Tracing the execution of parallel programs

To compute φ, we must consider the amount of time the program has lasted (total) and the amount of time the program has been with two or more cores working (parallel).

With this info we know that the total time is 2,499,071,038 and the parallel time is 2,296,837,809.

So after that, we can compute the parallelization with this two values:

φ = Tpar/Ttotal = 0.919



With this image we can see some clear data. More parallelized part, bigger synchronization we need, and that’s logic because if we divide our program in more parts we will need more time and resources to synchronize.