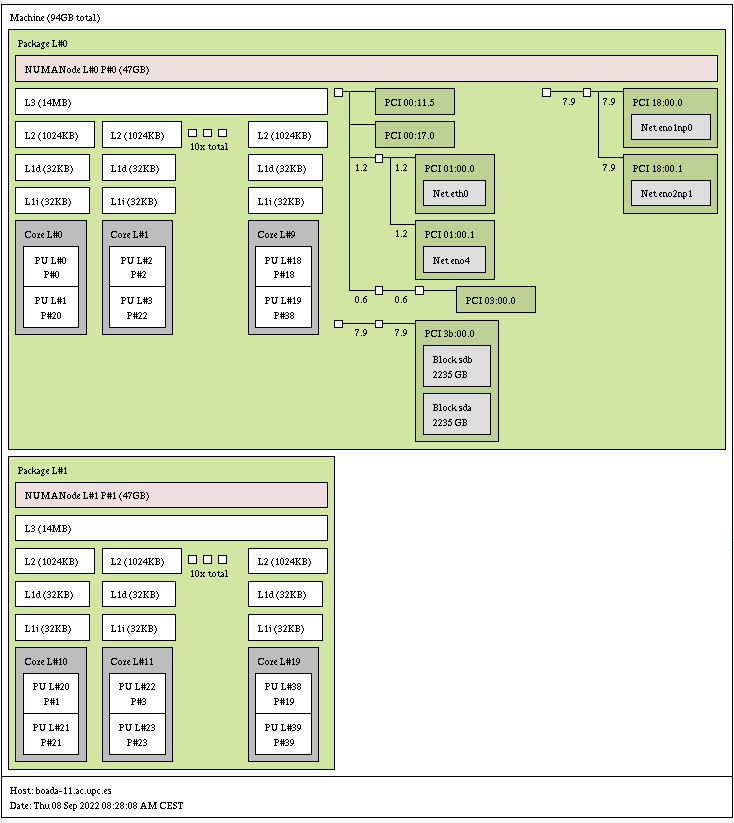
LABORATORY 1

Index

1. Experimental setup
   1. Node architecture
   2. Compilation and execution of OpenMP programs
2. Systematically analysing task decompositions with Tareador

1. Experimental setup

1.1 Node architecture

Boada server has 2 main operational parts, the boada 6-8 nodes which are used to run basic programs in “Interactive” mode with up to 2 parallel threads. The other cluster of nodes are the 11-14, which can run programs in “Execution” mode with a queue system and up to 20 parallel threads.

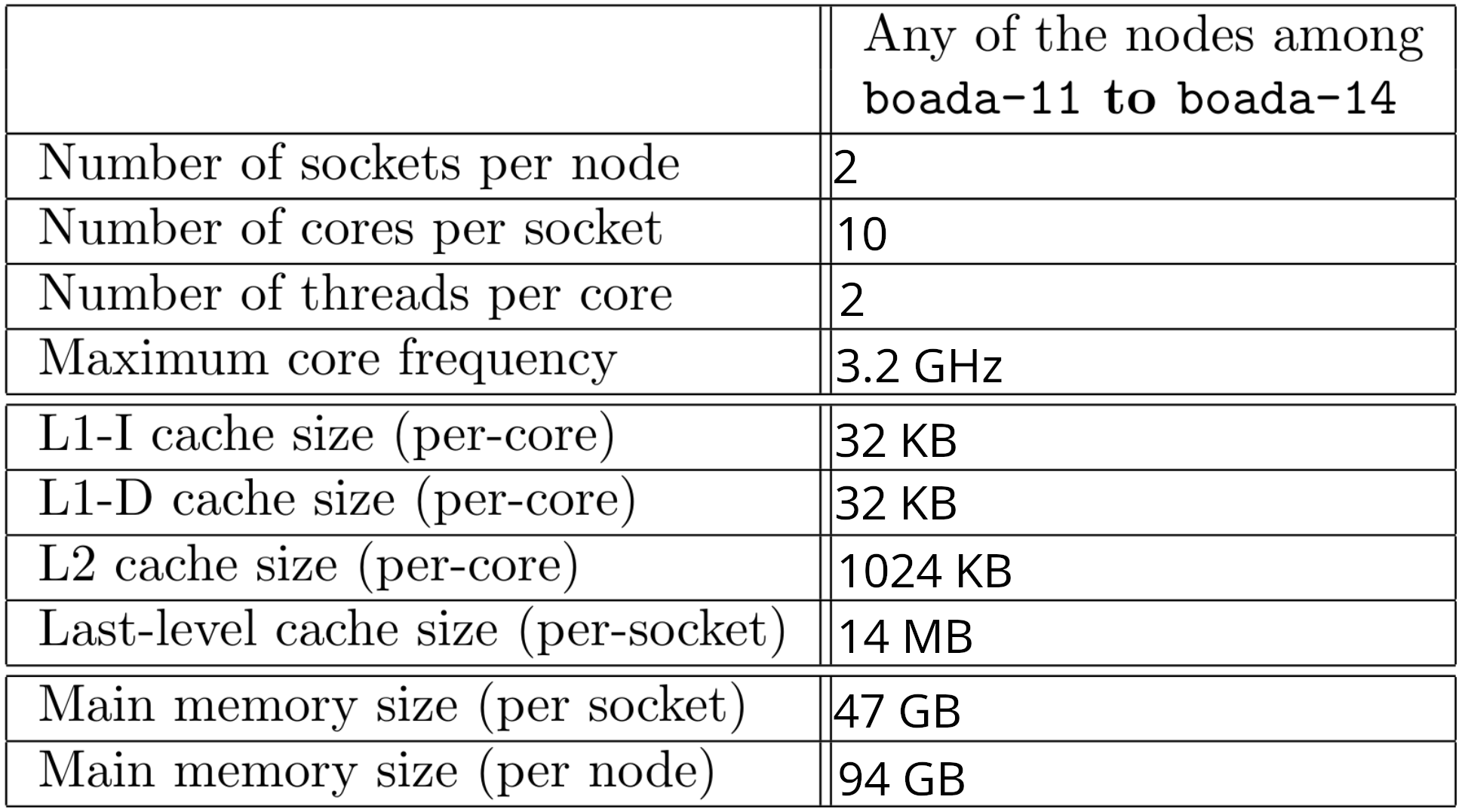


…

CPU max MHz: 3200.0000

CPU min MHz: 1000.0000

…

1.2 Compilation and execution of OpenMP programs

1.3 Strong vs weak scalability for PI

| # threads | Timing information (Interactive) | | | | Timing information (Execution) | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| user | system | elapsed | % of CPU | user | system | elapsed | % of CPU |
| 1 | 2.36s | 0.00s | 2.37s | 99% | 0.68s | 0.00s | 0.71s | 97% |
| 2 | 2.37s | 0.00s | 1.19s | 199% | 0.68s | 0.00s | 0.36s | 190% |
| 4 | 2.37s | 0.00s | 1.19s | 199% | 0.70s | 0.00s | 0.19s | 367% |
| 8 | 2.41s | 0.03s | 1.22s | 199% | 0.75s | 0.00s | 0.11s | 661% |
| 16 | 2.47s | 0.11s | 1.29s | 198% | 0.80s | 0.00s | 0.70s | 1147% |
| 20 | 2.47s | 0.11s | 1.30s | 199% | 0.83s | 0.00s | 0.60s | 1331% |



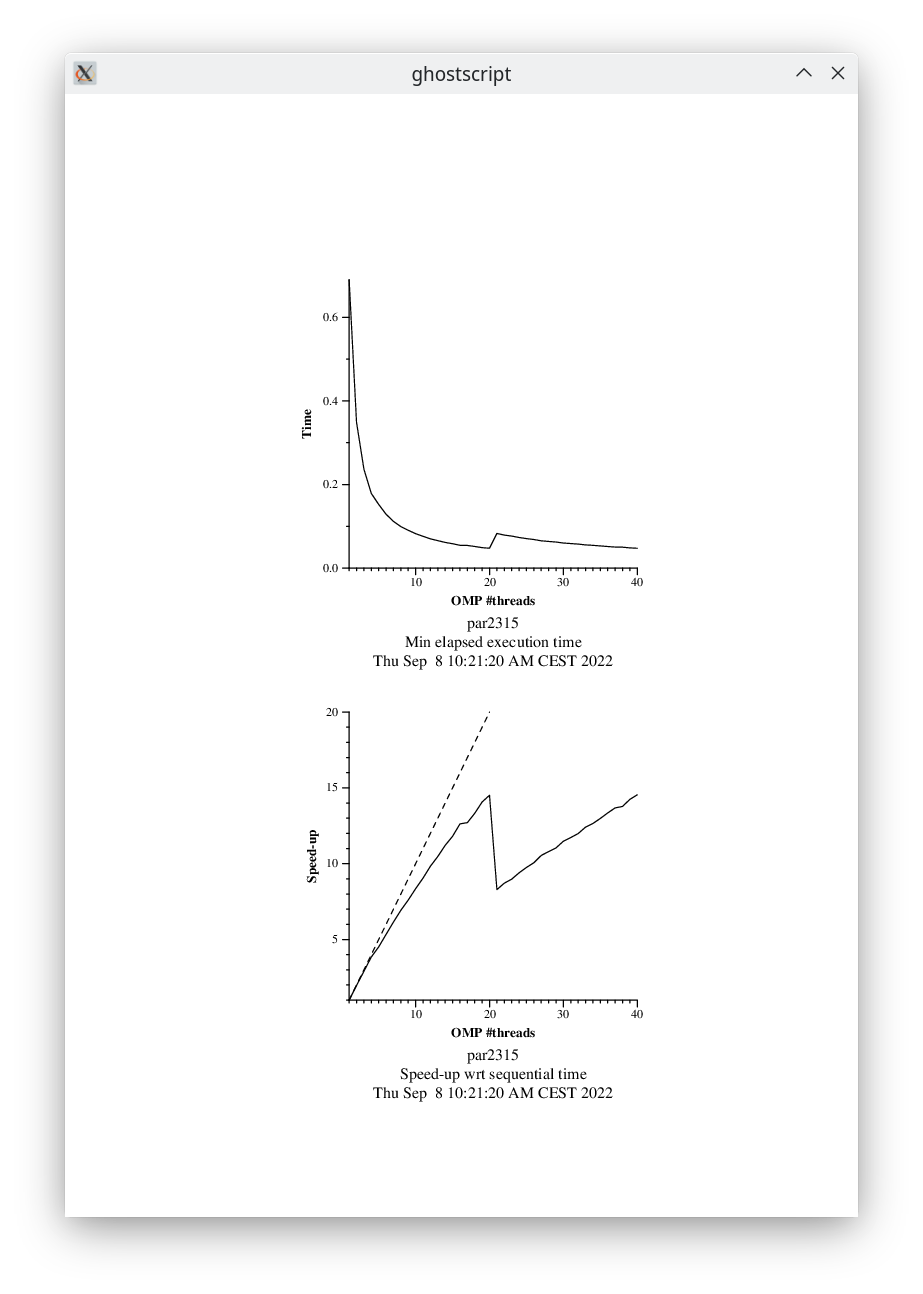
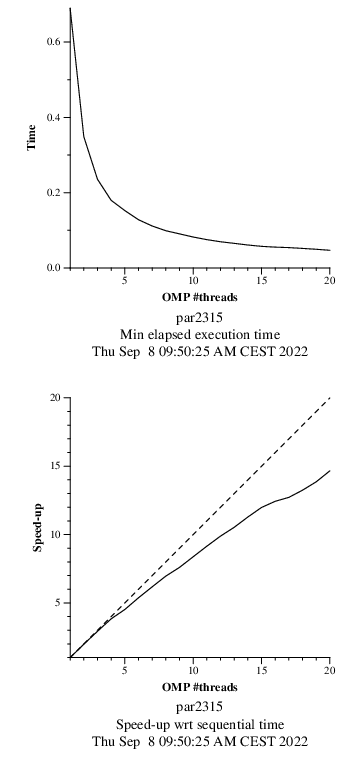
From the table we can deduce that the Interactive nodes had a maximum of 2 threads per program, because in spite of being the same code in both of them the Interactive one does not reduce the execution time past the threshold of 2 threads.

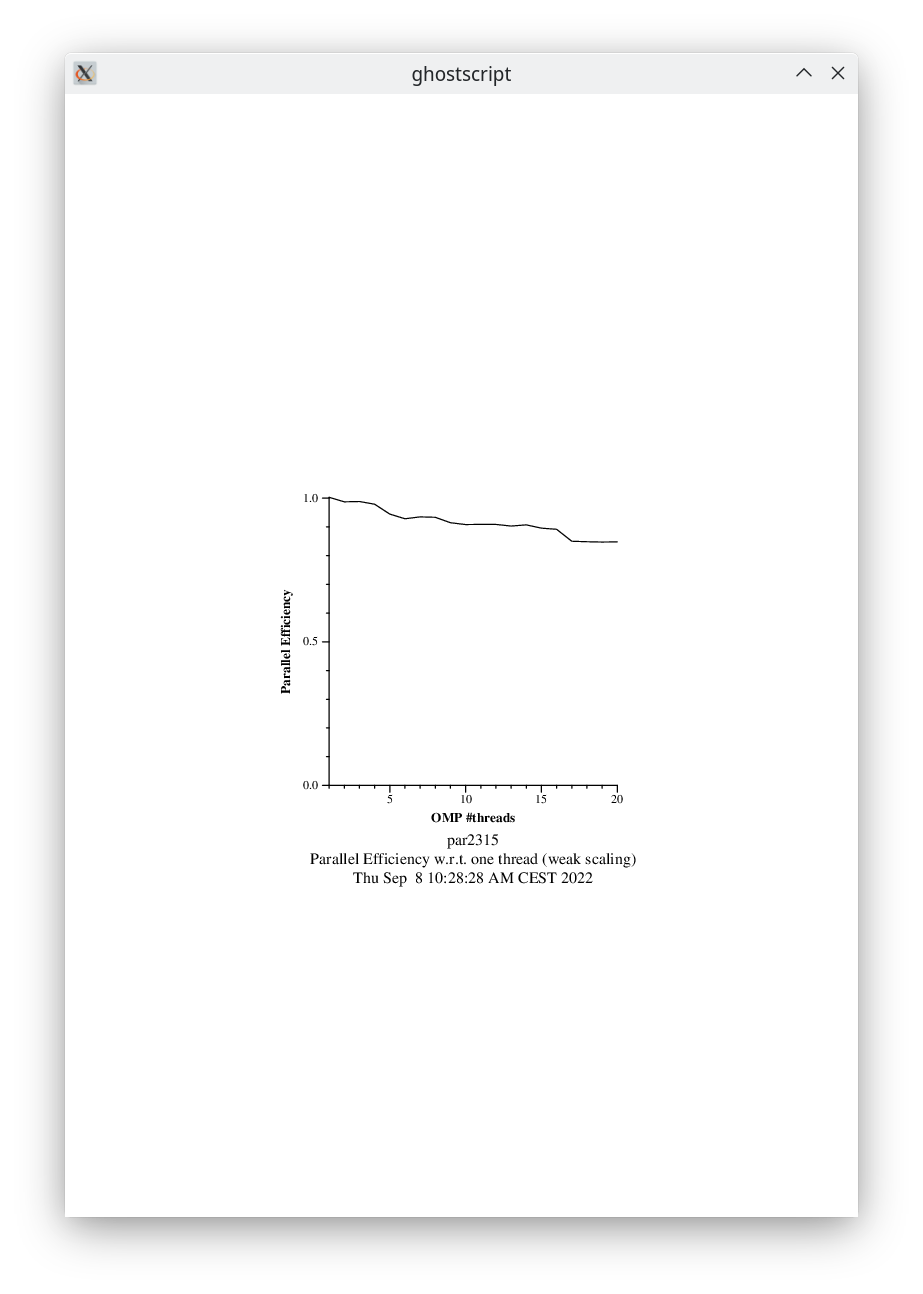
Another deduction that can be made is that the Interactive nodes are way less powerful than the Execution ones, because in the 1 thread test the elapsed time in the Execution node is approximately 3 times less than the other elapsed time.

The last conclusion we observe is that in the Execution nodes the increment rate of the CPU usage reduces when the number of threads increases. We deduce that this is caused by the non-parallelizable code.

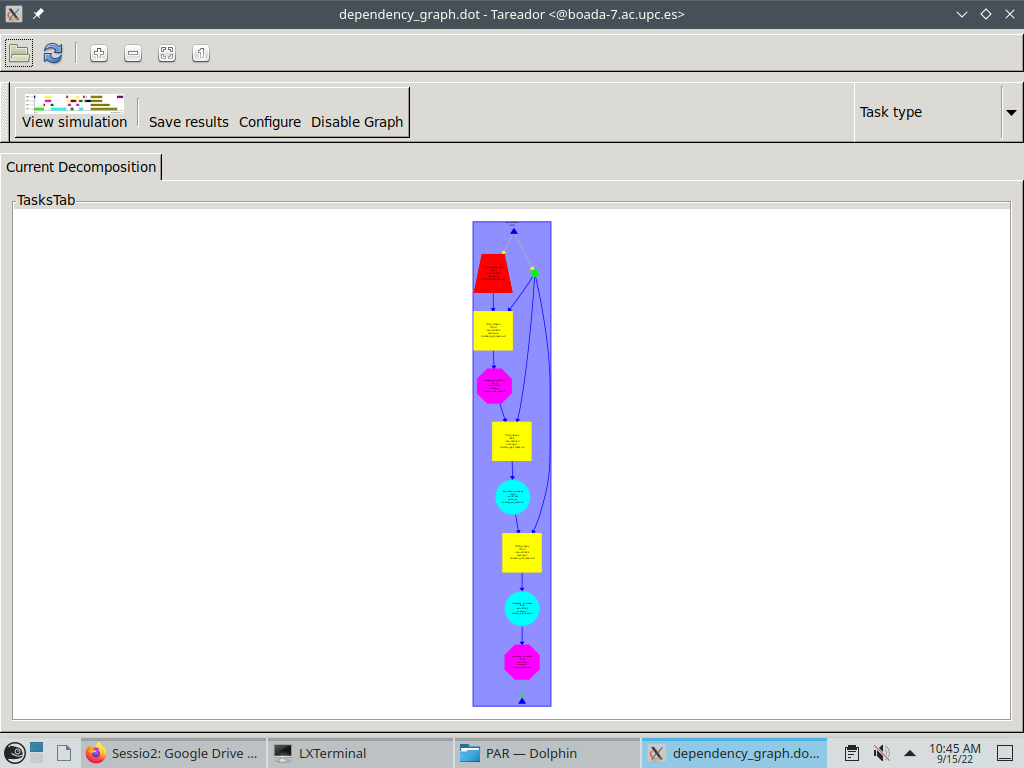
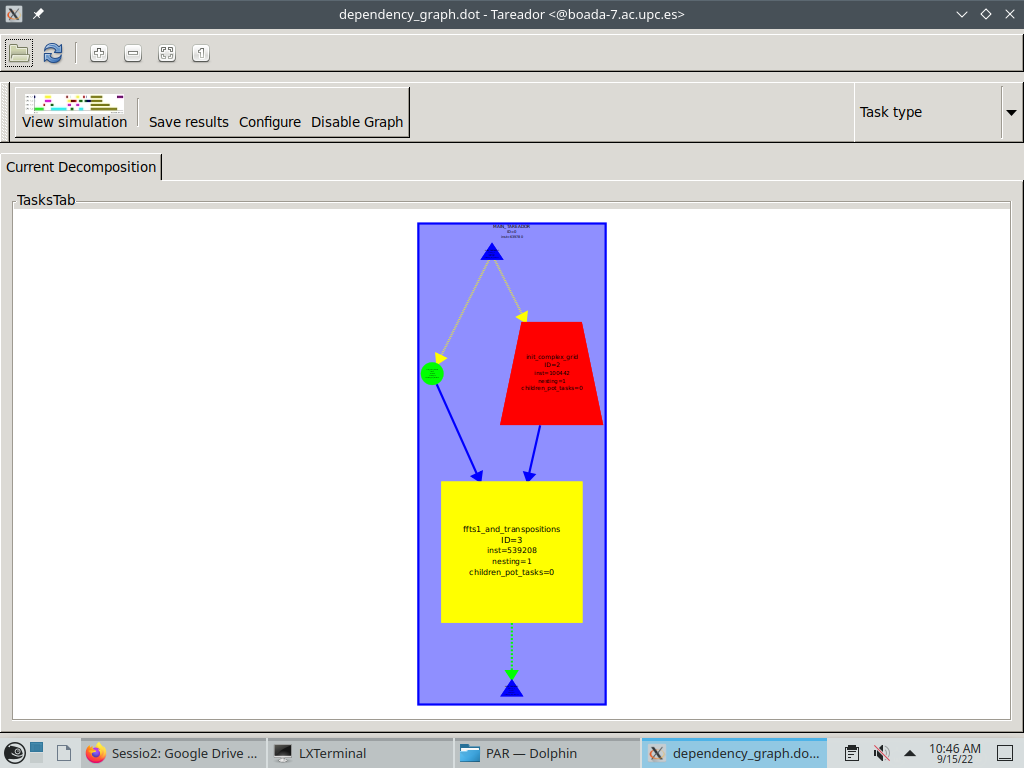
The scalability of a program is the speedup between the sequential and the parallel execution times. There are two types:

1. Strong scalability: we use parallelism to reduce the execution time of a program changing the number of threads to solve a problem of a fixed size (same number of iterations). In the minimum elapsed execution time graph of 20 cores we can see that the execution time of a problem rapidly decreases when we increase the number of threads.
2. Weak scalability: in this case the size of the problem changes proportionally to the number of threads. In the parallel efficiency graph we can see that the line graph is almost horizontal, which means that the parallel efficiency doesn’t improve as we increase the number of threads, unlike in the strong scalability case. That’s mainly because of two reasons: there will always be non-parallelizable code we can’t change, and the time needed for the synchronization of the threads also increases.

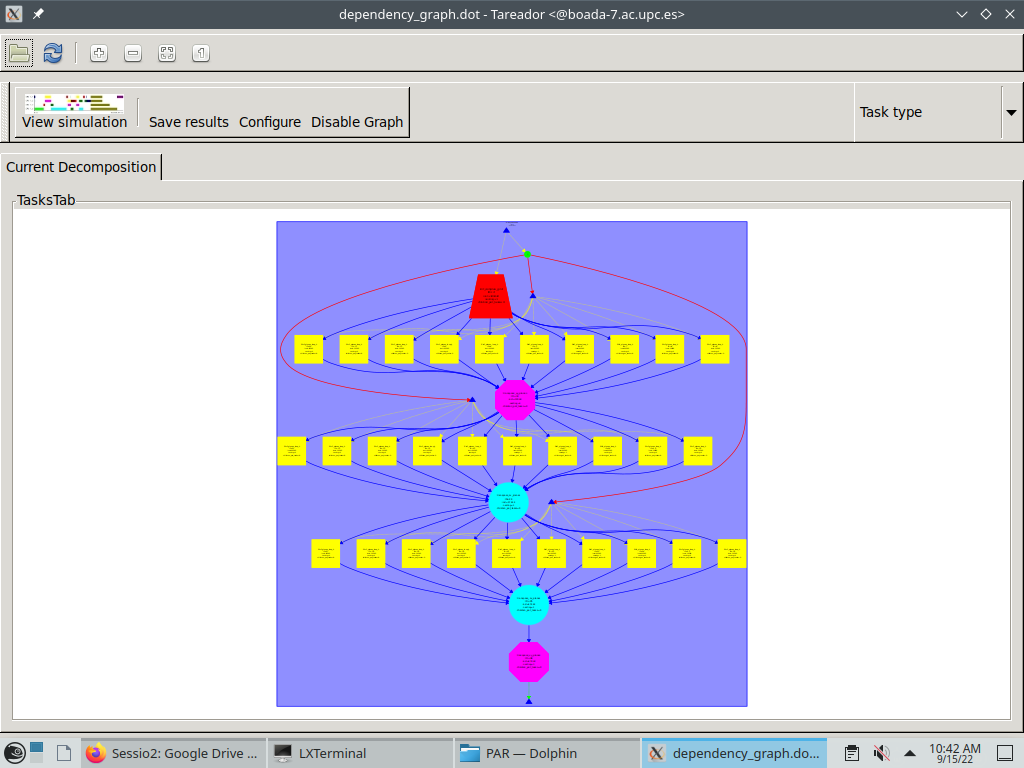


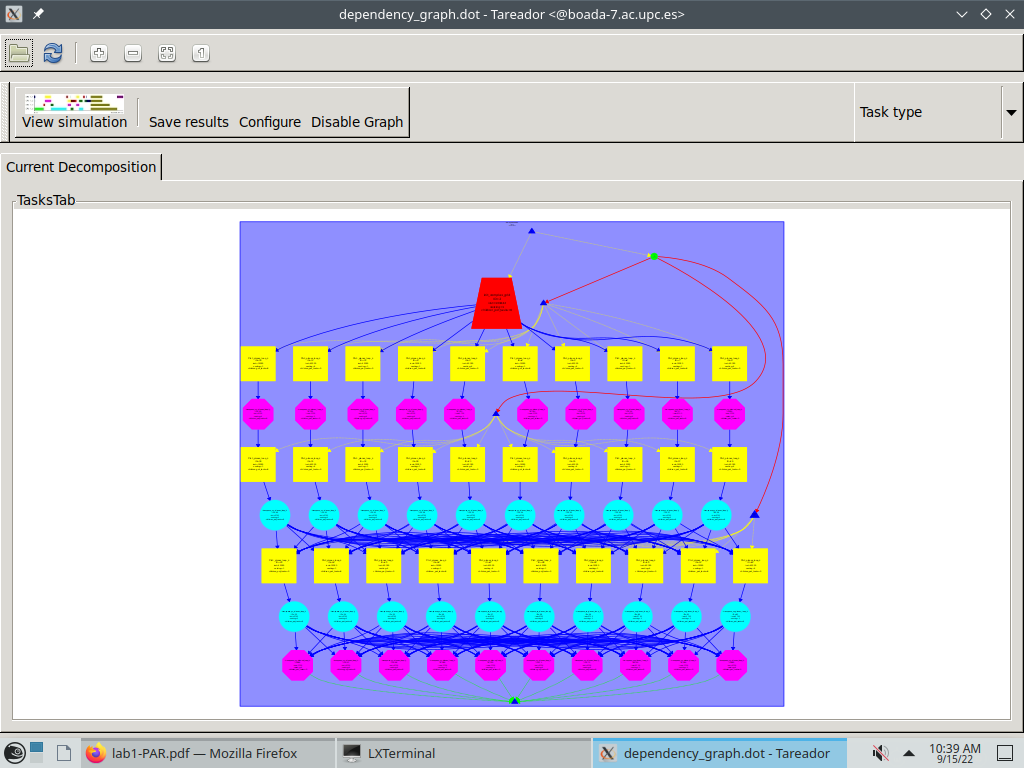


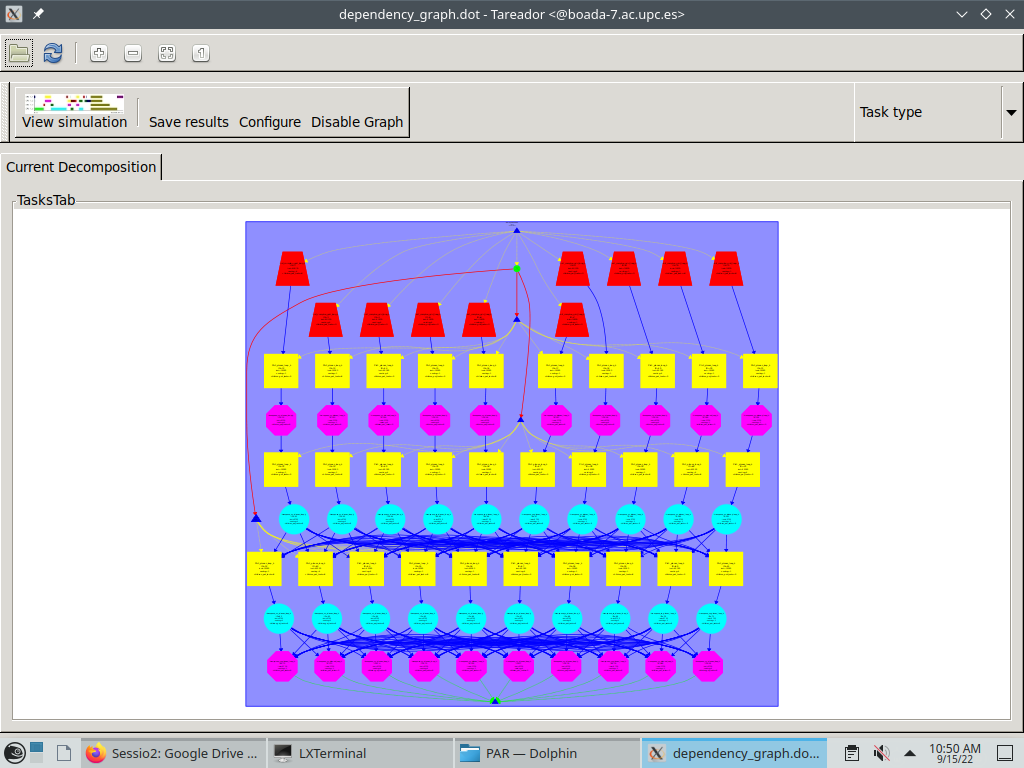
3.1 Exploring new task decompositions for 3DFFT

Task Dependence Graph of the original code (3dfft\_tarV0.c on the left) and version 1 (3dfft\_tarV1.c on the right).

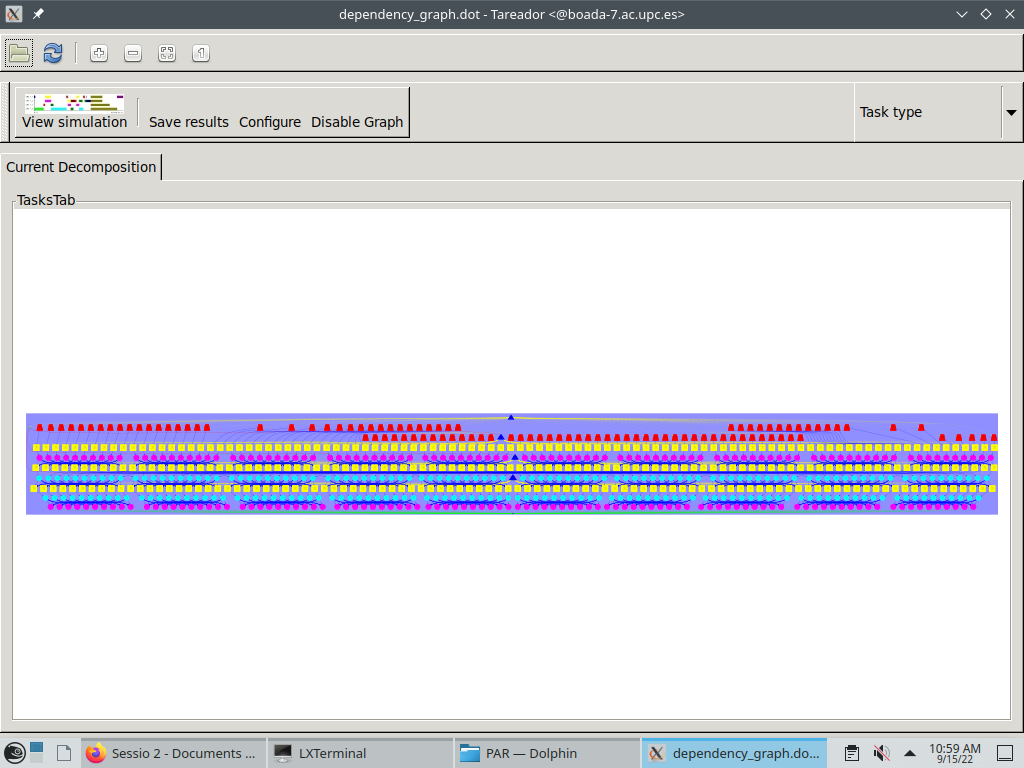
Task Dependence Graph of version 2 (3dfft\_tarV2.c).



Task Dependence Graph of version 3 (3dfft\_tarV3.c on the left) and version 4 (3dfft\_tarV4.c on the right).



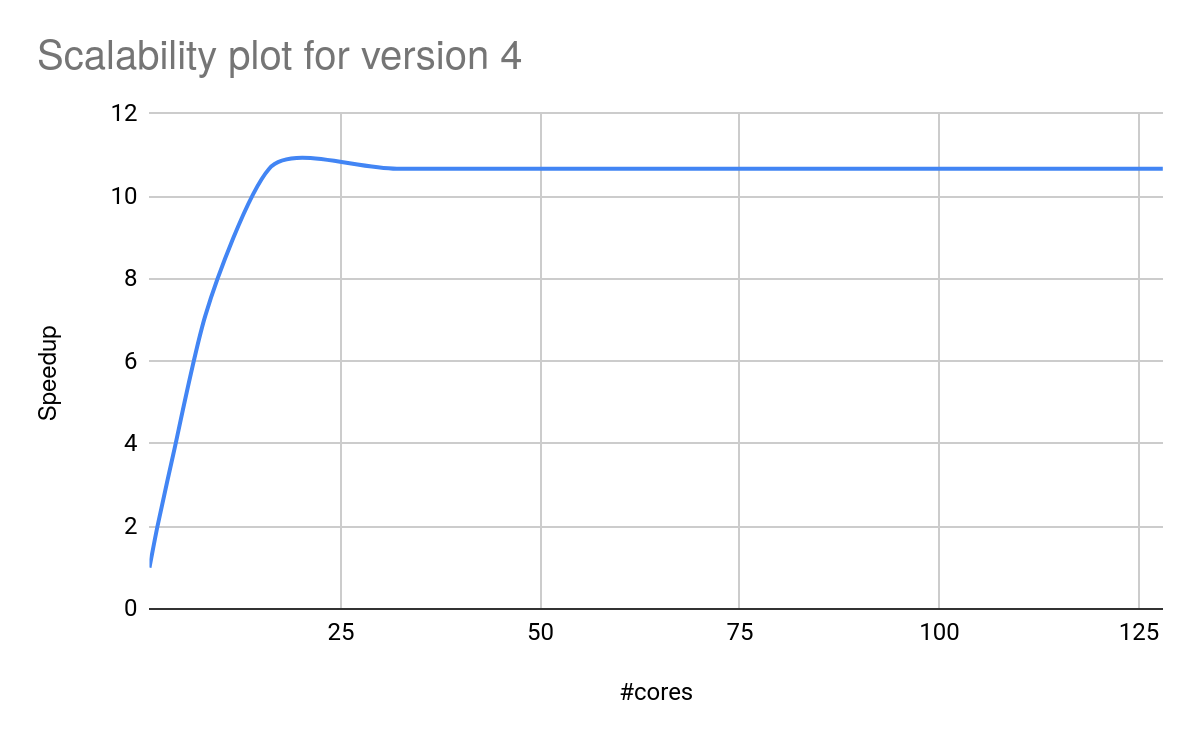
Task Dependence Graph of version 5 (3dfft\_tarV5.c).

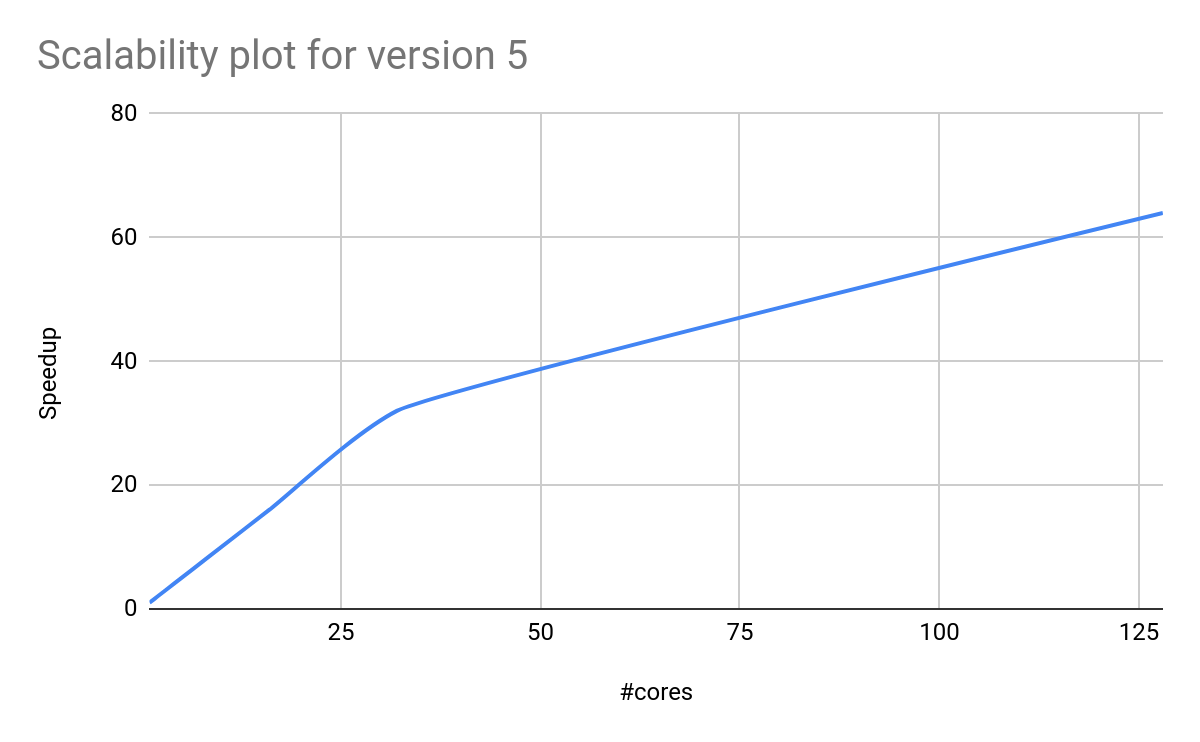
3.2 Execution table for all versions

| Version | T1 | T∞ | Parallelism |
| --- | --- | --- | --- |
| seq | 0.64 | 0.64 | 1 |
| v1 | 0.64 | 0.64 | 1 |
| v2 | 0.64 | 0.36 | 1.78 |
| v3 | 0.64 | 0.15 | 4.27 |
| v4 | 0.64 | 0.06 | 10.67 |
| v5 | 0.64 | 0.01 | 64 |

3.3 Time table and scalability graphs for versions 4 and 5

| Threads Num | V4 | V4 Speed-up | V5 | V5 Speed-up |
| --- | --- | --- | --- | --- |
| 1 | 0.64 | 1 | 0.64 | 1 |
| 2 | 0.32 | 2 | 0.32 | 2 |
| 4 | 0.17 | 3.76 | 0.16 | 4 |
| 8 | 0.09 | 7.11 | 0.08 | 8 |
| 16 | 0.06 | 10.67 | 0.04 | 16 |
| 32 | 0.06 | 10.67 | 0.02 | 32 |
| 128 | 0.06 | 10.67 | 0.01 | 64 |





We can see that version 4 has a weaker scalability than version 5. That’s because in version 4 we told the program that a task is equal to two loops, while in version 5 the task has only one loop, which makes the task faster. In the following sections we can see the difference in the respective codes.

3.4 Relevant code in version 4

…

for (k=0; k<N; k++) {

tareador\_start\_task("transpose\_xy\_planes\_loop\_k");

for (j=0; j<N; j++) {

for (i=0; i<N; i++) {

tmp\_fftw[k][i][j][0] = in\_fftw[k][j][i][0];

tmp\_fftw[k][i][j][1] = in\_fftw[k][j][i][1];

}

}

tareador\_end\_task("transpose\_xy\_planes\_loop\_k");

}

…

3.5 Relevant code in version 5

void transpose\_xy\_planes(fftwf\_complex tmp\_fftw[][N][N], fftwf\_complex in\_fftw[][N][N]) {

int k,j,i;

for (k=0; k<N; k++) {

for (j=0; j<N; j++) {

tareador\_start\_task("transpose\_xy\_planes\_loop\_j");

for (i=0; i<N; i++)

{

tmp\_fftw[k][i][j][0] = in\_fftw[k][j][i][0];

tmp\_fftw[k][i][j][1] = in\_fftw[k][j][i][1];

}

tareador\_end\_task("transpose\_xy\_planes\_loop\_j");

}

}

}

4.1