PAR Laboratory Assignment
LAB 1: Experimental Setup and Tools

GROUP 23 - PAR1405

Raimon Mercé Gotsens

Meir Carlos Mouyal Amselem

March 7th, 2019
2nd Semester

Node architecture and memory

Describing the architecture of the boada server

Architectural characteristics of the **boada** server according to the different node types available:

	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	2395MHz	2600MHz	1700MHz
L1-I cache size (per-core)	32KB	32KB	32KB
L1-D cache size (per-core)	32KB	32KB	32KB
L2 cache size (per-core)	256KB	256KB	256KB
Last-level cache size (per-socket)	12MB	15MB	20MB
Main memory size (per socket)	12GB	31GB	16GB
Main memory size (per node)	23GB	63GB	31GB

Architectural diagram for the node of **boada-X**:

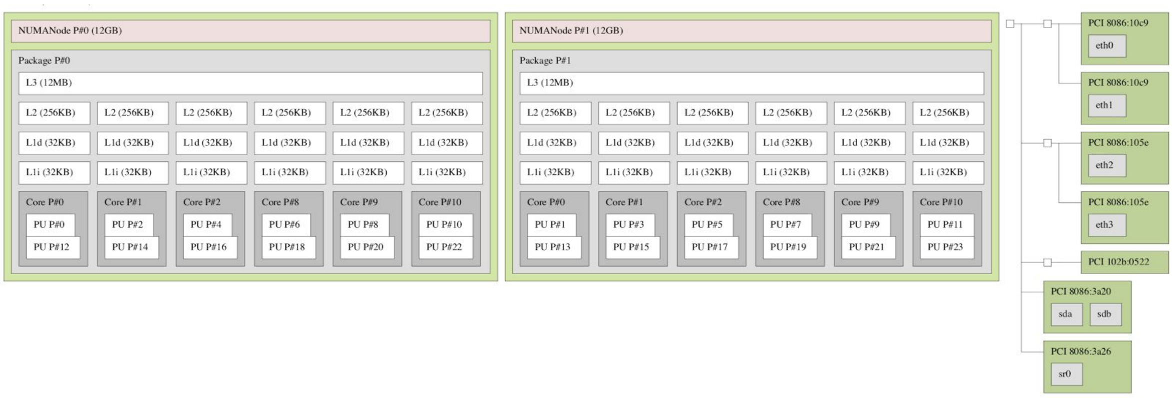


Figure 1 - lstopo obtained from boada-1

Strong vs. weak scalability

Definitions

Strong scalability: When the number of threads (processors) is increased while the problem size remains constant. It's used to reduce the execution time of the program.

Weak scalability: When the number of threads (processors) is increased proportionally to the problem size. It's used to know the limitations of parallelism in a large problem size.

Plots of pi_omp.c

Strong scalability

Boada 1-4

In Figure 2 we can see how increasing the number of threads decreases the execution time of the program logarithmically, therefore increasing its speed-up, as it can be seen in Figure 3, which doesn't linearly because the overhead increases every time a new thread is added.

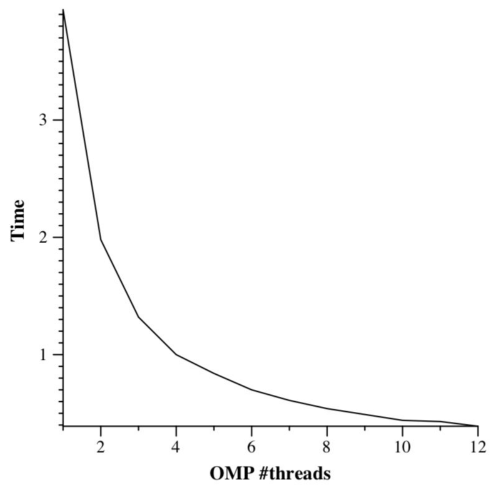


Figure 2 - Plot of strong scalability showing the minimum elapsed execution time for boada-3

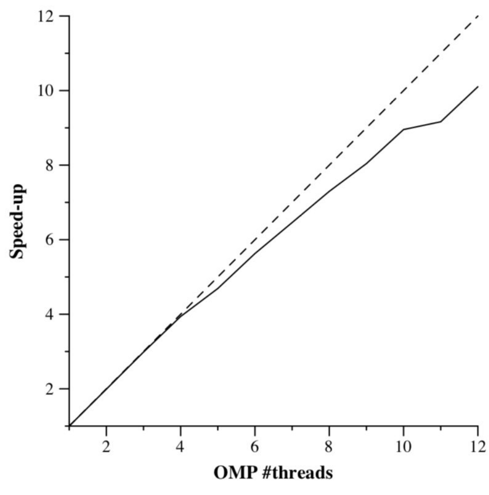


Figure 3 - Plot of strong scalability showing the speedup for boada-3

Boada 5

Figure 3 is almost as Figure 2, even it has a bit of a deformation, but the interesting figure to analyze is Figure 4. Figure 4 seems to have a speed-up much better than the one shown in Figure 2, since it increases linearly. But this only happens until the 9th thread is added, in which the overhead seems to change dramatically.

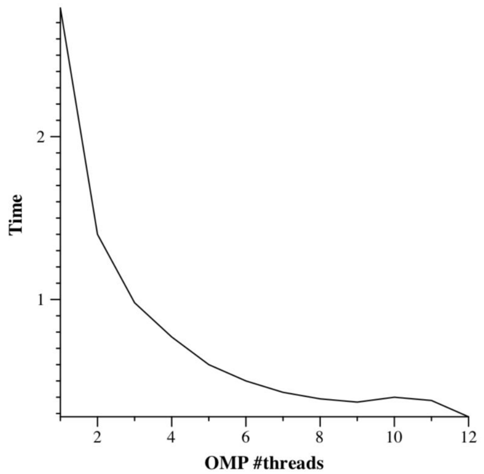


Figure 3 - Plot of strong scalability showing the minimum elapsed execution time for boada-5

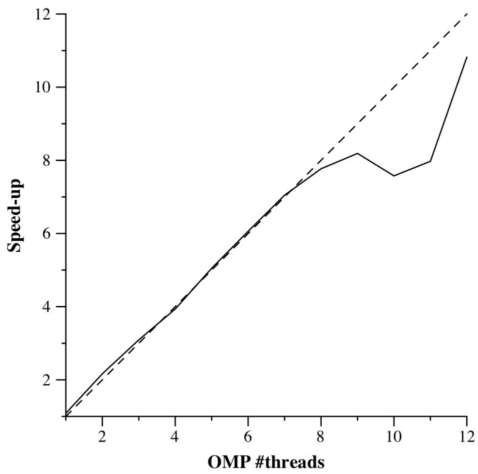


Figure 4 - Plot of strong scalability showing the speedup for boada-5

Boada 6-8

Figure 5 is almost identical as Figure 2, but in Figure 6 we can perfectly notice how the speed-up become almost linear, becoming the best architecture for parallelizing pi_omp.c in strong scaling, having an almost linear increase of its speed-up.

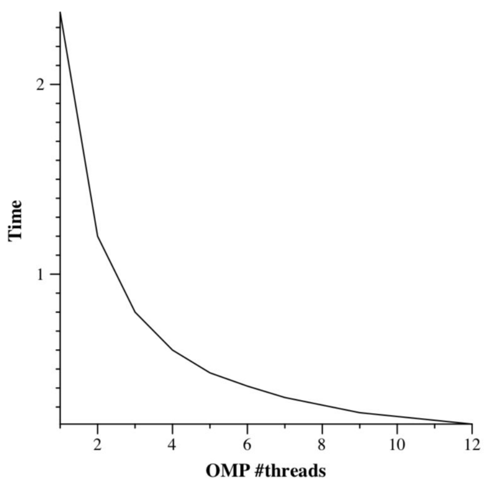


Figure 5 - Plot of strong scalability showing the minimum elapsed execution time for boada-6

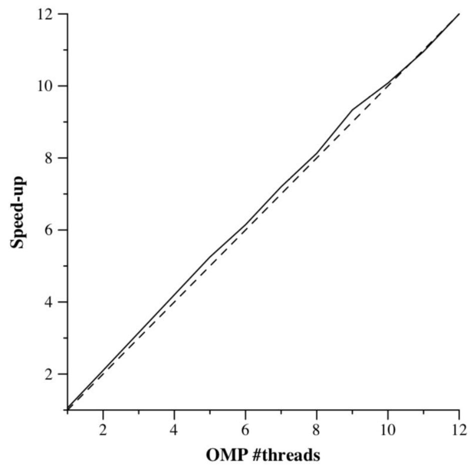


Figure 6 - Plot of strong scalability showing the speedup for boada-6

Weak scalability

Boada 1-4

In Figure 7, where we can observe how the speed-up decreases when more threads are added to the execution program. This is due to that the problem size increases proportionally to the amount of threads, and therefore execution time also increases due to the overhead mentioned in the previous section.

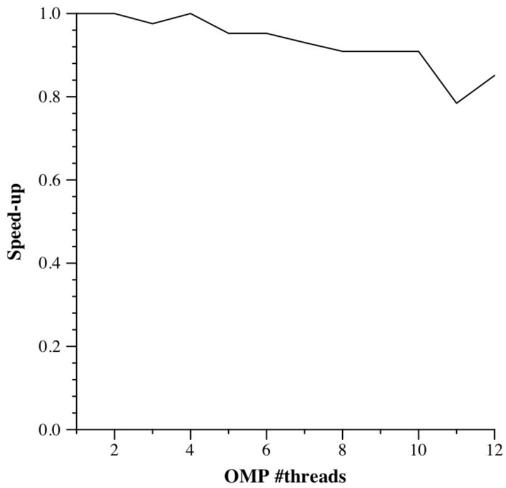


Figure 7 - Plot of weak scalability showing the speed-up for boada-4

Boada 5

For some reason through Figure 8, we can deduce that boada-5 behaves in a strange way by having peaks in its speed-up, until the 10th thread, when the speed-up seems to decrease dramatically. But as shown until the 10th thread, we can assume that boada-5 is the best architecture of weak scaling.

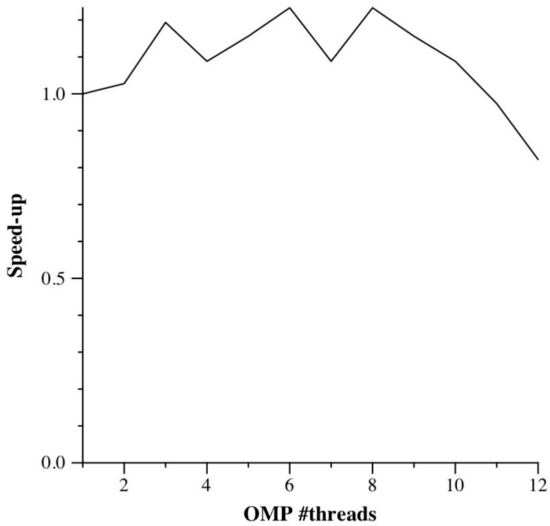


Figure 8 - Plot of weak scalability showing the speed-up for boada-5

Boada 6-8

By analyzing Figure 9, we can infer that this interval of boada's does a really good attempt at maintaining its speed-up constant while the problem size increases proportionally with the amount of threads.

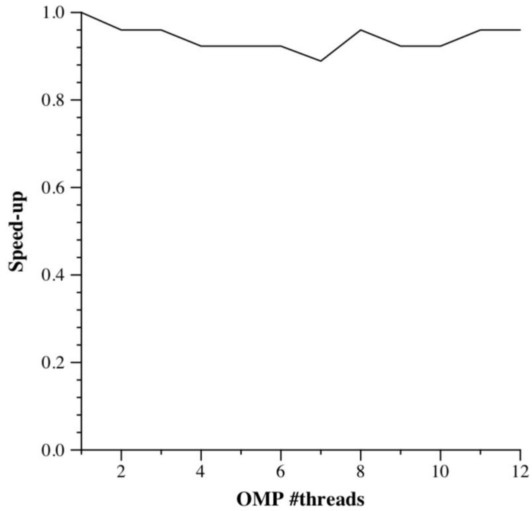


Figure 9 - Plot of weak scalability showing the speed-up for boada-8

Analysis of task decompositions for 3DFFT

By following the instruction of the manual of Lab1, we obtained the results shown in the following table:

- With the initial and different versions generated for 3dfft_tar.c, briefly commenting the evolution of the metrics.

Version	T_1	T_∞	Parallelism
seq	630 ms	630 ms	1.000
v1	630 ms	639 ms	0.986
v2	639 ms	361 ms	1.771
v3	638 ms	154 ms	4.145
v4	639 ms	64 ms	9.993
v5	639 ms	37 ms	17.229

What the results show is that for every version we tested, we managed to increase its T_∞ and its parallelism rate, which we almost managed to double from v4 to v5. With this results we learned the importance of task decomposition and the impact of changing a single line of code from one line to another, giving much better results.

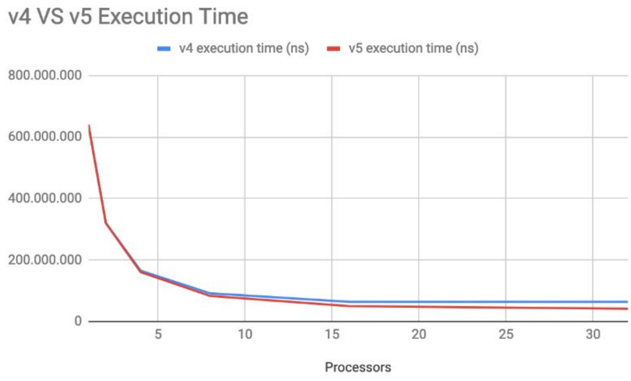
Now we will proceed to explain what was done in each version of the code in order to understand its evolution:

- In v1 we generated a task for each of the functions that the main code called.
- For v2 we realized that the function ffts1_planes(pd1, tmp_fftw) was being called 3 times, so as an improvement we decided to decompose its tasks inside of the function and not outside as it was done in v1.
- For v3 we followed the same logic as in v2 and applied the same changes for the functions transpose_xy_planes and transpose_zx_planes.
- For v4 we finally repeated the process for one last time for the only function left, init_complex_grid, without the previous change explained.
- Nevertheless, v5 was different, for this version we didn't had instructions to follow, and therefore we had to reason our own way of improving the parallelism of the code, so we decided to add another task decomposition in the second for loop.

For the simplicity of the document and to help the reader understand why v5 is able to scale to a higher number of processors compared to v4, we will be attaching the code, graphs and traces of all versions in the .tar delivered.

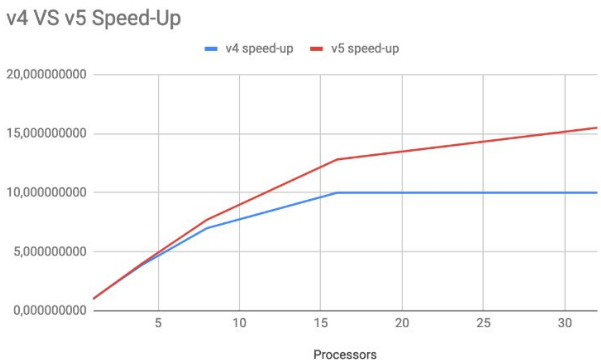
Plots

Processors	v4 execution time (ns)	v5 execution time (ns)
1	639.780.001	639.780.001
2	320.310.001	319.946.001
4	165.389.001	160.350.001
8	91.496.001	83.000.001
16	64.018.001	49.946.001
32	64.018.001	41.287.001



Plot of execution time of v4 and v5 using different processors

Processors	v4 speed-up	v5 speed-up
1	1,000000000	1,000000000
2	1,997377537	1,999649938
4	3,868334636	3,989897081
8	6,992436762	7,708192690
16	9,993751617	12,809433950
32	9,993751617	15,495918460



Plot of speed-up of v4 and v5 using different processors

Understanding the parallel execution of 3DFFT

Version	ϕ	S_∞	T_1 (ms)	T_8 (ms)	S_8
initial version in **3dfft_omp.c**	0,6497	1.82	2.356,80	1.706,70	1.3809
new version with improved ϕ	0,8876	1,83	2.340,69	1.429,78	1.6370
final version with reduced parallelisation overheads	0,8897	1,79	2.497,75	619,99	4.0286

In this part of the document, we will referee the differents versions of the programs as V1, V2 and V3.

V1 = Original version
V2 = Improved ϕ version
V3 = Reducing parallelisation overheads version

First, we need to know the % of the programs that can be parallelized (Φ). In order to do that, we will execute each version using 1 thread, so it behaves as a sequential program (T_1).

Using Parever and loading the trace we want, we can open a New Single Timeline window and see the duration of our program at the bottom right (Figure 10).

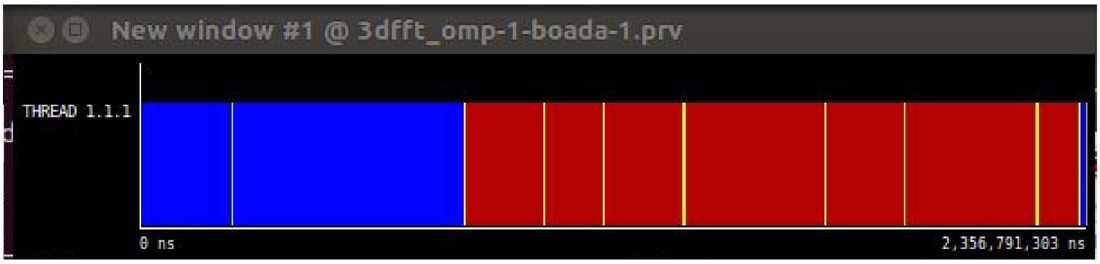


Figure 10 - Single Timeline window of V1 with 1 thread

First of all we loaded the configuration Parallel construction (Figure 11), then we loaded the State Profile and on Window Properties we changed the Control View selecting Parallel Constructs, and finally we chose Time as Statistic (Figure 12), so we would be able to see the amount of time that is parallelized on the main program (Figure 13).

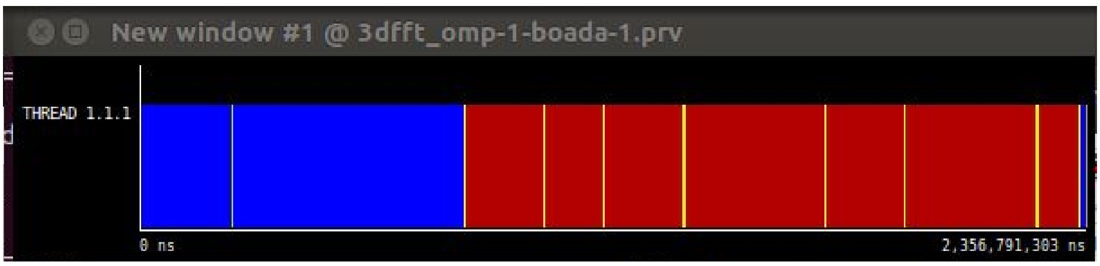


Figure 10 - Single Timeline window of V1 with 1 thread

Figure 11 - Parallel construct of V1 with 1 thread

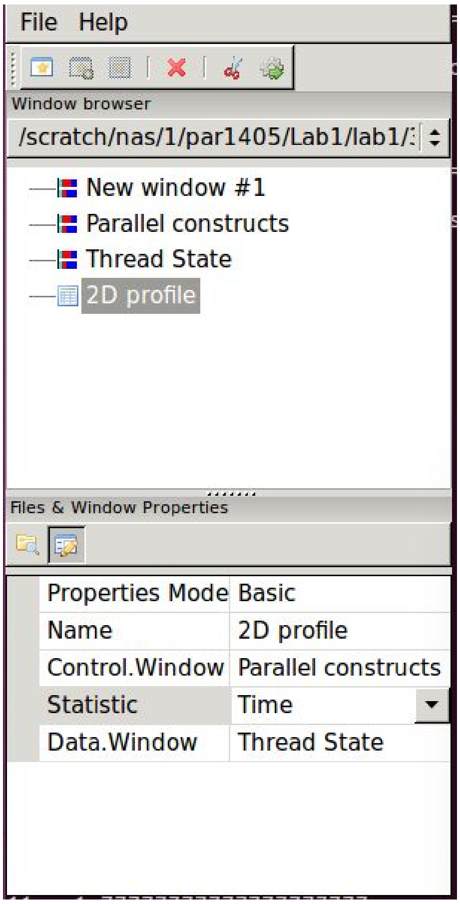


Figure 12 - Main window of paraver of V1 with 1 thread with the correct configuration

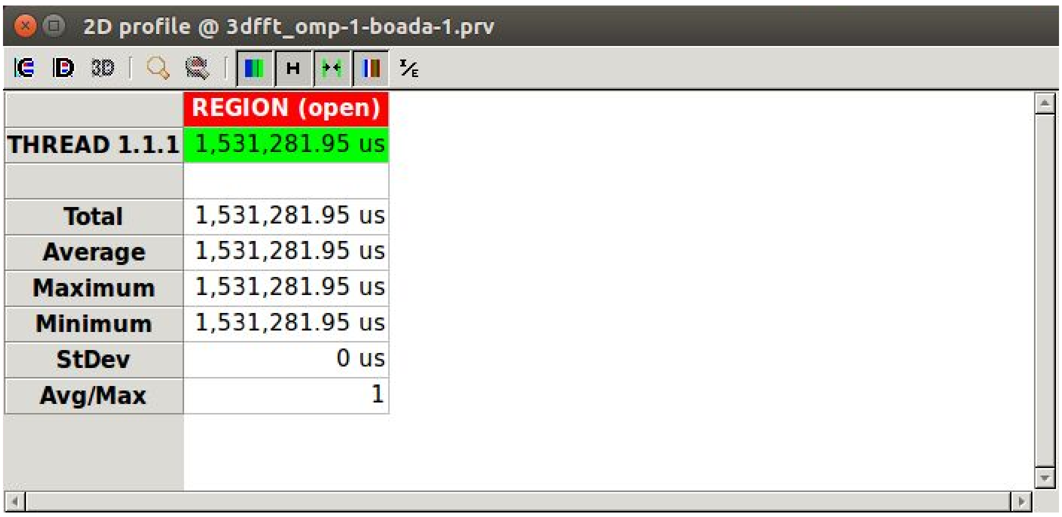


Figure 13 - State Profile of Parallel construct of V1 with 1 thread

With Tpar / (Tpar + Tseq) we can obtain **Φ**.
We can load the trace of *3dft_omp-8-boada-1.prv* and open a New Single Timeline window and see the cost of T_8 (Figure 14).

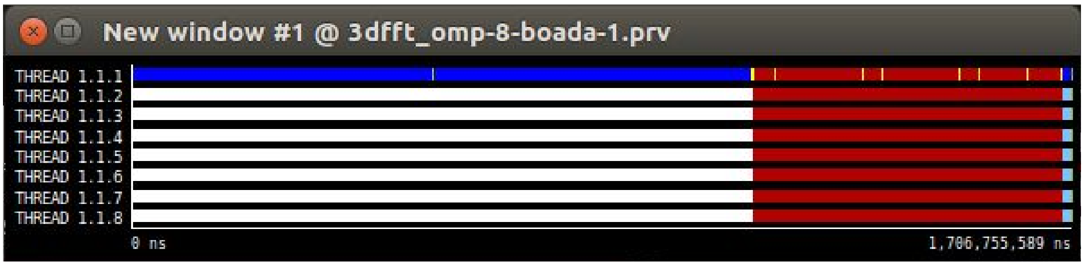


Figure 14 - Single Timeline window of V1 with 8 threads

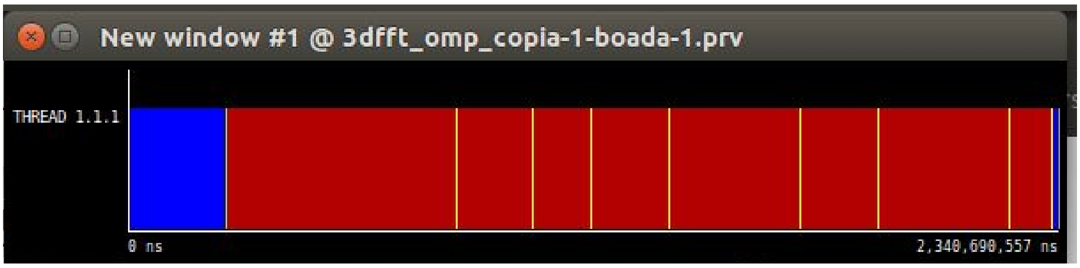
We can obtain S_8 using T_1 / T_8 and executing ***/submit-strong-omp.sh*** and find the S_∞ (Figure 15)

```
Resultat de l'experiment (tambe es troben a  ./elapsed-boada-1.txt  i
./speedup-boada-1.txt  )
#threads        Elapsed min
1    2.14
2    1.78
3    1.35
4    1.25
5    1.18
6    1.17
7    1.16
8    1.15
9    1.16
10    1.16
11    1.19
12    1.17

#threads        Speedup
1    .96728971962616822429
2    1.16292134831460674157
3    1.53333333333333333333
4    1.65600000000000000000
5    1.75423728813559322033
6    1.76923076923076923076
7    1.78448275862068965517
8    1.80000000000000000000
9    1.78448275862068965517
10    1.78448275862068965517
11    1.73949579831932773109
12    1.76923076923076923076
```

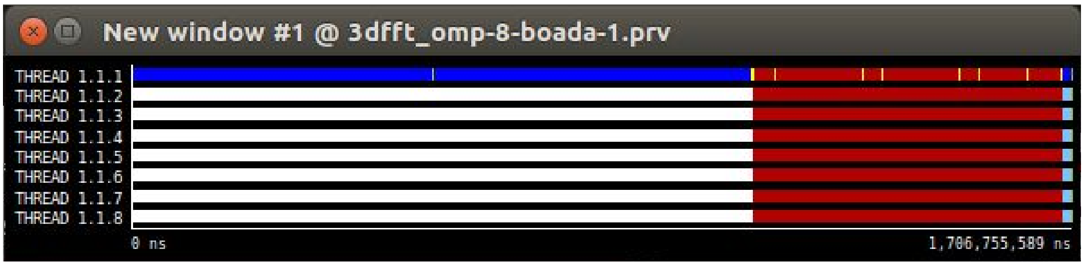
Figure 15 - Execution of strong scalability script on V1

Now we can calculate the rest of the table doing the same for the other two versions of the program. Here some screenshots made during the process.

V2 screenshots

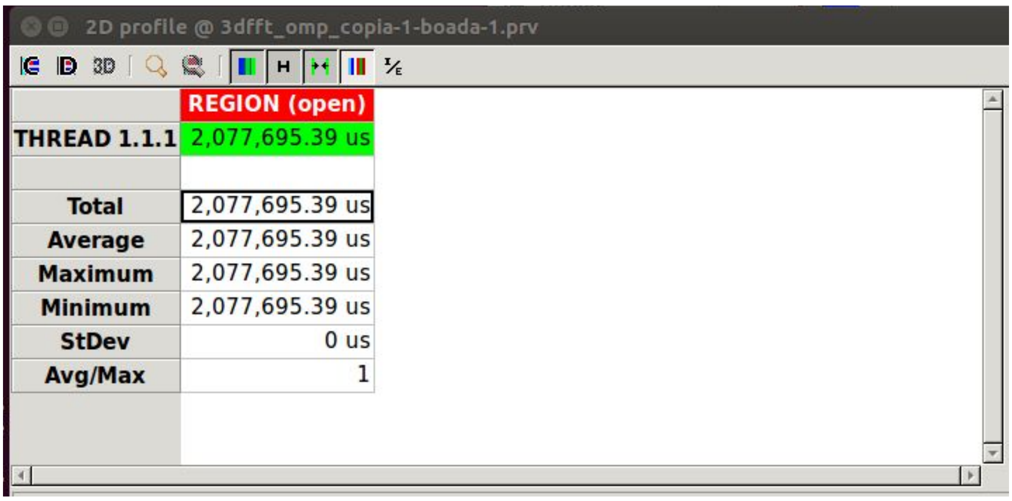


Single Timeline window of V2 with 1 thread



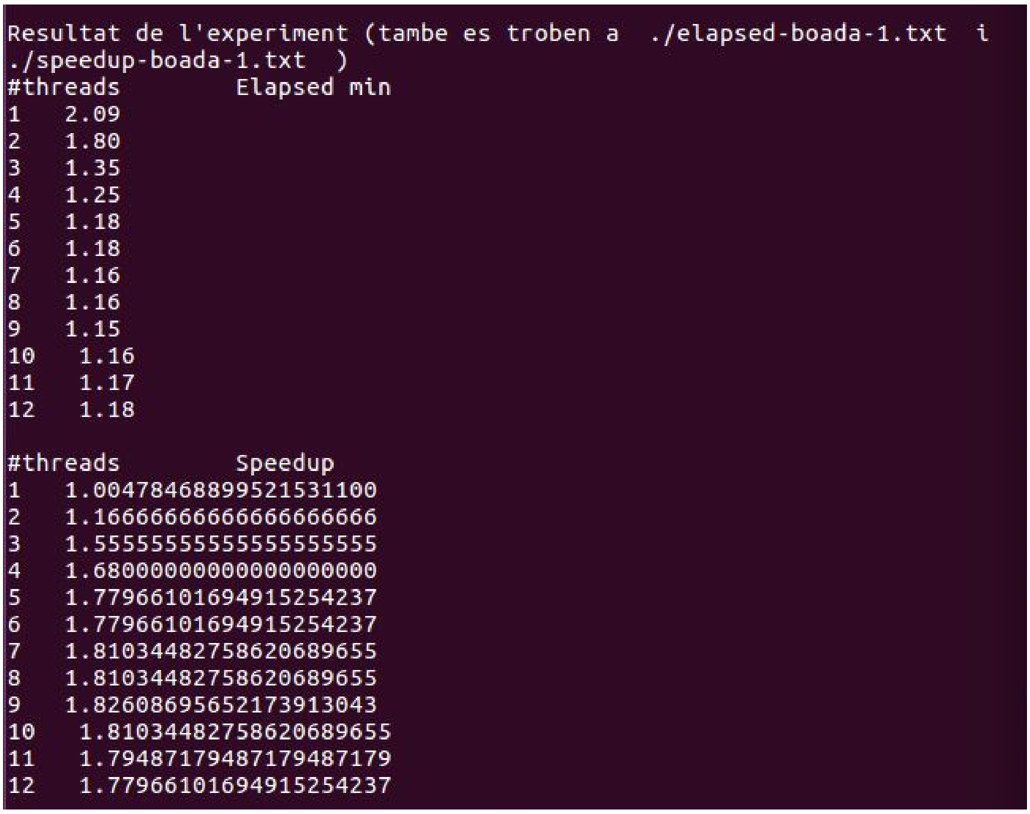
Single Timeline window of V2 with 8 thread

Parallel construct of V2 with 1 thread



	REGION (open)
THREAD 1.1.1	2,077,695.39 us
Total	2,077,695.39 us
Average	2,077,695.39 us
Maximum	2,077,695.39 us
Minimum	2,077,695.39 us
StDev	0 us
Avg/Max	1

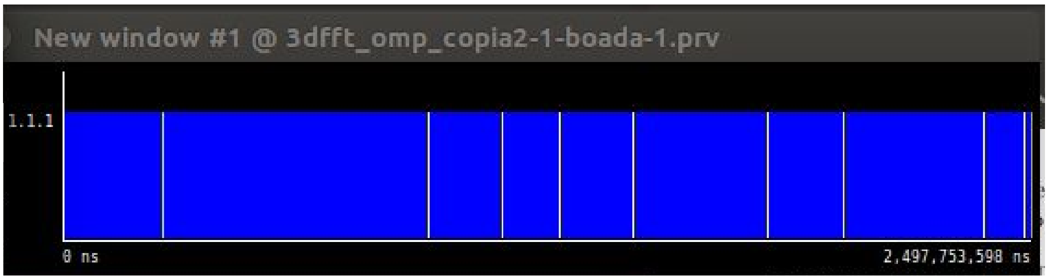
State Profile of Parallel construct of V2 with 1 thread



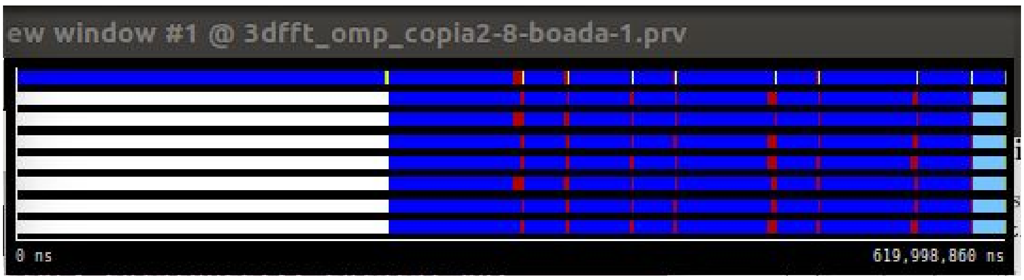
```
Resultat de l'experiment (tambe es troben a  ./elapsed-boada-1.txt  i
./speedup-boada-1.txt  )
#threads        Elapsed min
1    2.09
2    1.80
3    1.35
4    1.25
5    1.18
6    1.18
7    1.16
8    1.16
9    1.15
10    1.16
11    1.17
12    1.18

#threads        Speedup
1    1.00478468899521531100
2    1.16666666666666666666
3    1.55555555555555555555
4    1.68000000000000000000
5    1.77966101694915254237
6    1.77966101694915254237
7    1.81034482758620689655
8    1.81034482758620689655
9    1.82608695652173913043
10    1.81034482758620689655
11    1.79487179487179487179
12    1.77966101694915254237
```

Figure 16 - Execution of strong scalability script on V2

V3 screenshots

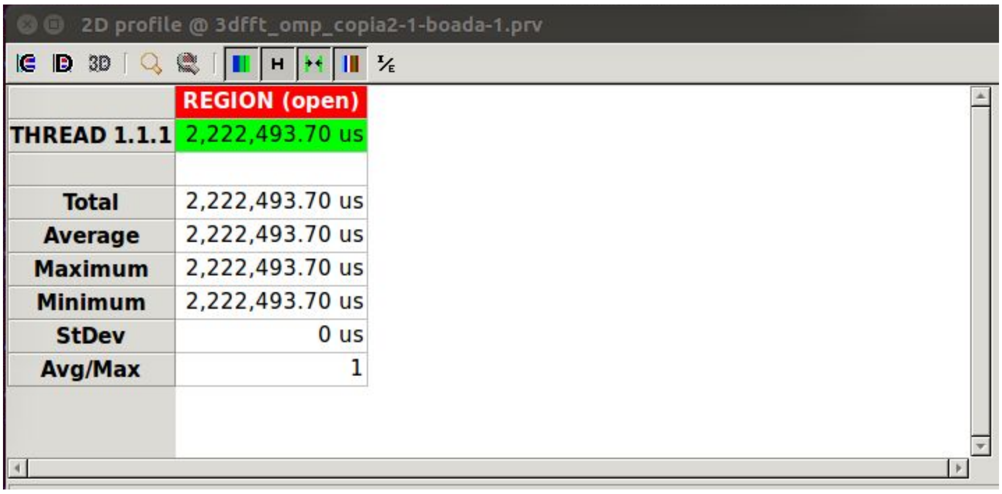


Single Timeline window of V3 with 1 thread



Single Timeline window of V3 with 8 thread

Parallel construct of V3 with 1 thread



	REGION (open)
THREAD 1.1.1	2,222,493.70 us
Total	2,222,493.70 us
Average	2,222,493.70 us
Maximum	2,222,493.70 us
Minimum	2,222,493.70 us
StDev	0 us
Avg/Max	1

State Profile of Parallel construct of V3 with 1 thread

```
Resultat de l'experiment (tambe es troben a  ./elapsed-boada-1.txt  i
./speedup-boada-1.txt  )
#threads         Elapsed min
1    2.09
2    1.79
3    1.35
4    1.26
5    1.19
6    1.18
7    1.16
8    1.16
9    1.16
10    1.16
11    1.16
12    1.19

#threads         Speedup
1    1.00478468899521531100
2    1.17318435754189944134
3    1.55555555555555555555
4    1.66666666666666666666
5    1.76470588235294117647
6    1.77966101694915254237
7    1.81034482758620689655
8    1.81034482758620689655
9    1.81034482758620689655
10    1.81034482758620689655
11    1.81034482758620689655
12    1.76470588235294117647
```

Figure 17 - Execution of strong scalability script on V3

From both Figure 18 and FIgure 19 we have used information of the execution of the strong scalability script (Figure 15, 16 and 17).

Figure 18 shows the Execution time in seconds off the three versions using from 1 to 12 threads. As we can see, all three programs have a very similar cost. From 1 to 3, the cost is reduced considerably, but then it gets stuck.

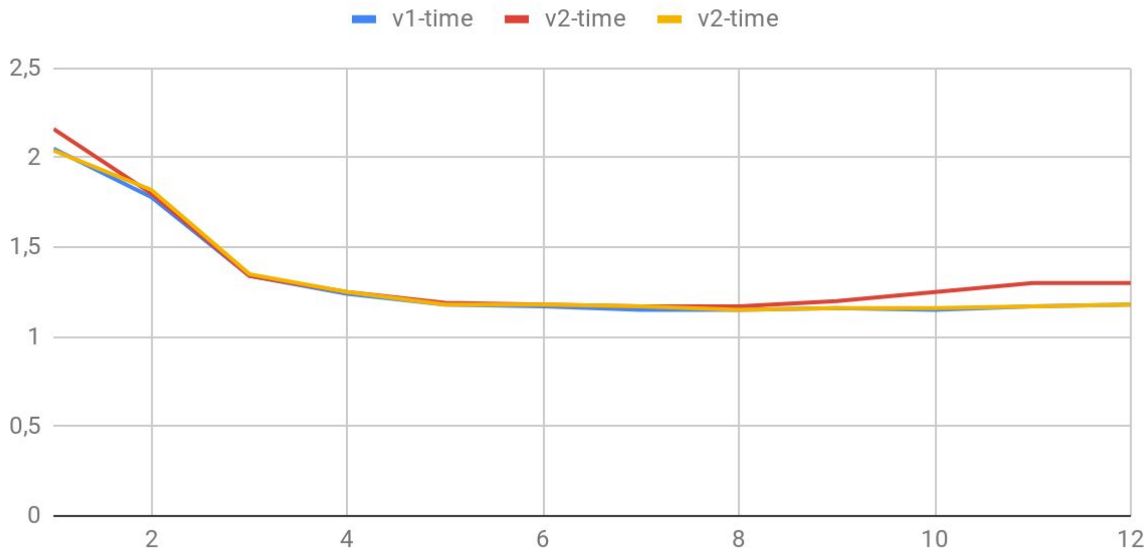


Figure 18 - Execution time/ Threds

Figure 19 shows the the Speed-up off the three versions using also from 1 to 12 threads. Like the previous one, all three programs have a very Speed-up, it grows faster from 1 thread to 5 threads, but then it gets stuck, and as we can see, V2 goes even slower using 11 and 12 threads.

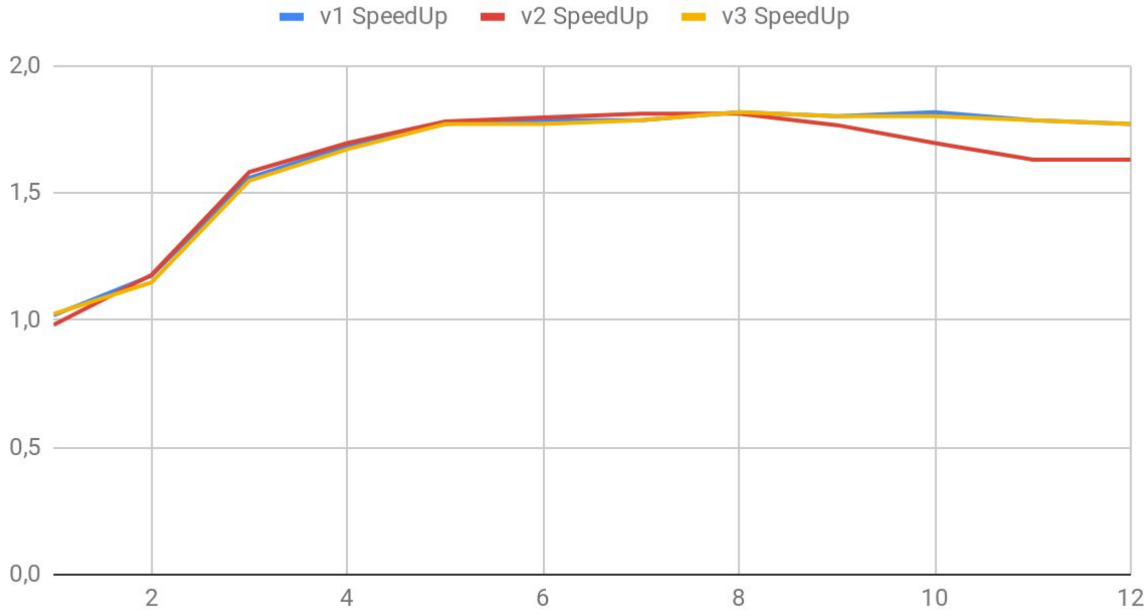


Figure 19 - Speed-up/ Threds