PAR Laboratory Assignment Lab 1: Experimental setup and tools

E. Ayguadé, J. R. Herrero, P. Martínez-Ferrer,
J. Morillo, J. Tubella and G. Utrera Spring
2021-22

Maria Montalvo Falcón
Victor Pla Sanchís
Group: 42
Course: 2021-2022 Q2

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Maria Montalvo Falcón - par4214
Victor Pla Sanchís - par4219
2021-2022 Q2
Group: 42

Index

Maria Montalvo Falcón - par4214
Victor Pla Sanchís - par4219
2021-2022 Q2
Group: 42

FIRST SESSION

1. Introduction

In this subject, the main purpose is to work the parallelism of applications. We are going to understand why this goal is so important in order to achieve efficiency.
In the first practice lesson we will get used to the environment and tools of the course.

2. Node architecture and memory

In order to understand the environment of the course, we have to understand the architecture of the boada server.

Boada is organised as a cluster of nodes that goes from boada-1 to boada-8. The first node, boada-1, is a shared node between the users of the system. As a shared node, it has some limitations as for example: the number of CPUs one can use at the same time, in our system they are, 2. In order to execute one program in this node, one can directly execute the command: ./run-xxxx.sh and see the result on the console.

On the other hand, we have the rest of the nodes from boada-2 to boada-8. These nodes aren't shared, so each program could be executed from 1 to 12 CPUs. In this case, to organise the programs that are going to be executed, the system has a queue of priority with the oldest time as the priorest. In this case, to execute a program the user has to execute the command sbatch [-p partition] ./submit-xxxx.sh and then execute the cat command with the generated file as an argument.

One could see the summarised information of boada-1 to boada-4 on Table 1.1, obtained after executing the commands: lscpu and lstopo.

	boada-1 **to** boada-4
Number of sockets per node	2
Number of cores per socket	6
Number of threads per core	2
Maximum core frequency	2.40 Ghz
L1-I cache size (per-core)	32 KB
L1-D cache size (per-core)	32 KB
L2 cache size (per-core)	256 KB
Last-level cache size (per-socket)	128 MB
Main memory size (per socket)	12 GB
Main memory size (per node)	23 GB

Table 2.1 Summarised information of boada-1 to boada-4

To finish this first point, and also to understand graphically the information plotted on Table 1.1, we can see the diagram of node boada-1 displayed on Figure 1.1.
We can see in this figure: the memory of the system which is 23 GB, the NUMANNodes that are the corresponding sockets of the system with each memory of 12 GB, the number of cores per socket that are 6, and all the capacities of each memory.

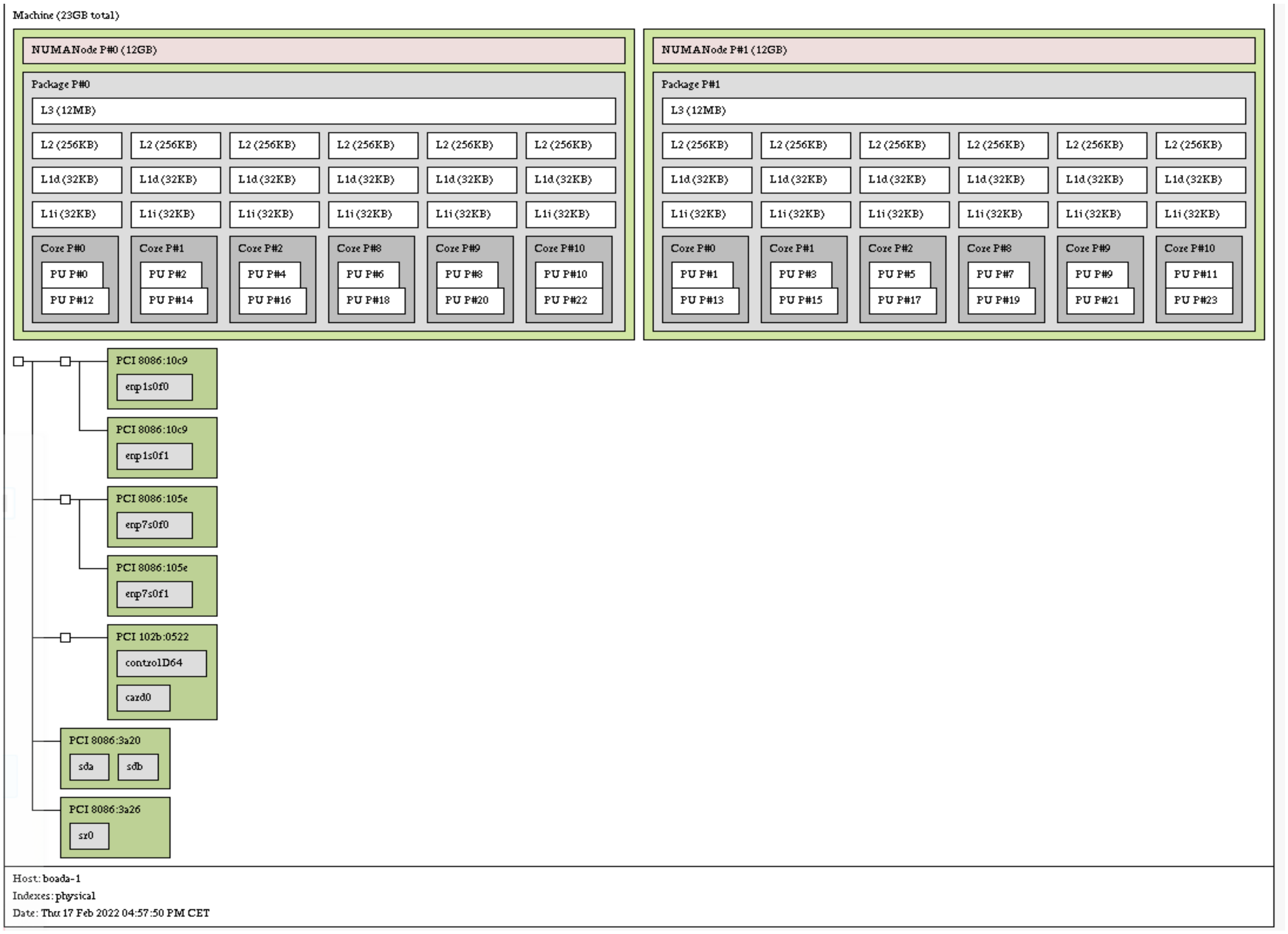


Figure 2.1 : Diagram node boada-1

3. Strong vs. weak scalability

The main difference between Interactive and Queued execution is reflected on the elapsed time and also the % of use of CPU. As we see on the following table, the Interactive execution gets similar values on all different threads executions on all the parameters. That's because boada-1, where Interactive execution is run, just can use 1 thread about all threads available; the reason why this happens is because it is a shared environment. On the other hand, the Queued execution allows running the program in a better way talking about parallelism because it can use from 1 to 12 threads. This is reflected in the differences on the % of CPU, and of course on the time execution of the program. Mathematically, the parallelism difference between Interactive and Queued execution is:

$$Interactive = Texe / P, \text{ where } P = \{1\}$$

$$Queued = Texe / P, \text{ where } P = \{1,2,4,8\}$$

$$Interactive / Queued = Pinteractive / Pqueued = 1 / 8$$

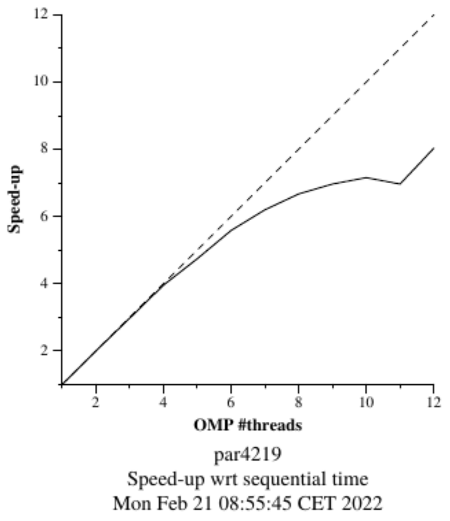
As we see, on the maximum performance of Interactive and Queued executions we have that the factor of speed-up on the Interactive respect Queued execution is about 8 times less.

# threads	Interactive				Queued			
	user	system	elapsed	% of CPU	user	system	elapsed	% of CPU
1	7.89	0.0	0:07.89	99%	7.86	0.0	0:08.13	96%
2	15.96	0.0	0:07:98	199%	7.89	0.0	0:03.98	198%
4	15.93	0.0	0:07.97	199%	7.98	0.0	0:02.01	395%
8	15.95	0.0	0:07.98	199%	9.06	0.0	0:01.24	731%

Table 3.1 : Results Interactive and Queued executions

When we talk about strong scalability, we are referring to the improving performance of a program growing parallelism in the number of threads. That allows us to reduce the time execution of the program if we differ correctly the tasks to parallelise it. On the other hand, in weak scalability we can also improve our performance program by growing parallelism in quality of resolution. That means that we are also improving our program but not reducing the time execution by growing the amount of data that our program is able to process, remaining the execution time always the same value but in a better way of resolution.

As we have said, the purpose of strong scalability is to reduce the time execution as we see on figure 1.3. We can figure that the speed-up increases in a logarithmic form. That's because the performance of a parallelism program is limited in terms of the number of threads, so we'll not be able to reduce the overhead time or to parallelise sequentials parts of the program. On the right plot, we can see how that affects the execution time of the program, so it will never reach overhead time but it behaves as asymptotic.



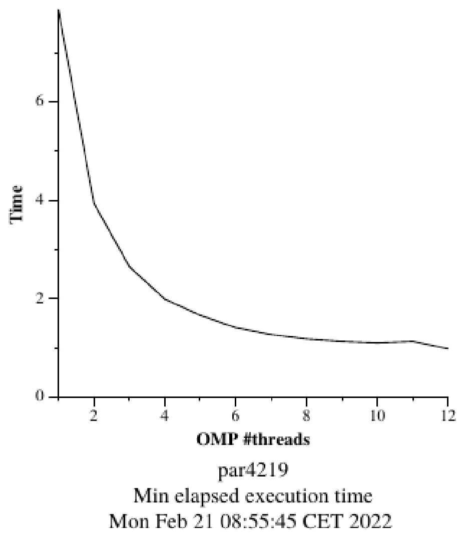


Figure 3.1 : Plots strong scalability

Here below, we can see on figure 3.2 the plot related to weak scaling. As we have previously said, this scalability is produced when one increases the quantity of tasks in the program but remains the execution time by adding processors.
In this case we can see that the parallel efficiency also decreases when we add approximately more than 4-5 threads. We can relate this decrease in efficiency with the overhead of the program. As we have seen, parallelising it is not for free, and we have to weigh up all the pros and cons in order to see if a change is positive or negative to our application.

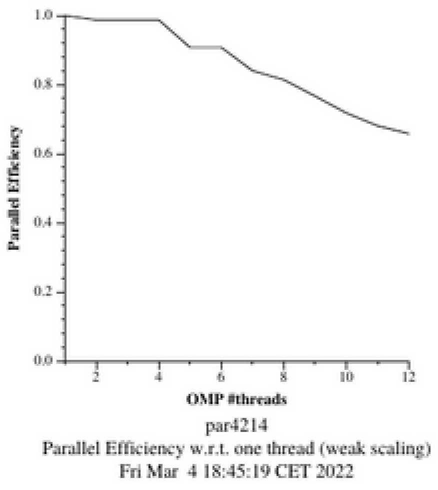


Figure 3.2 : Plot weak scalability

Maria Montalvo Falcón - par4214
Victor Pla Sanchís - par4219
2021-2022 Q2
Group: 42

SECOND SESSION

4. Analysis of task decompositions for *3DFFT*

In this lesson we are going to be introduced to a new tool named *Tareador*, which allows the user to parallelise a program in an interactive mode.

Tareador gives the user the option to divide a task in many little ones inside of a program. The user also has different graphic supports to understand some properties of the task such as: the number of instructions, tasks' dependencies (including data or control ones), the execution diagram with a set of CPUs chosen, or different shapes and colours to identify the tasks and their granularity.

To study the possibilities that bring us the tool *Tareador,* we have worked on the same program with different kinds of task decomposition and number of CPUs.

In order to explain the results, we have the backup of: the table that plots T_1 and T_∞ (see plot 4.1) and the diagrams for each version.

Version	T_1	T_∞	**Parallelism**
seq	6,39 s	6,39 s	1
v1	6,39 S	6,39 S	1
v2	6,39 s	3,61 s	1,77
v3	6,39 s	1,55 s	4,15
v4	6,39 s	0,64 s	9,26
v5	6,39 s	0,35 s	18,26

Table 4.1 : Plots T_1,$T_\infty$and parallelism

First of all, the program given was divided into 5 different tasks: the first one for the initialization, then 3 tasks for the computation of the program and the last one for the reduction of the program (see Figure 4.1).

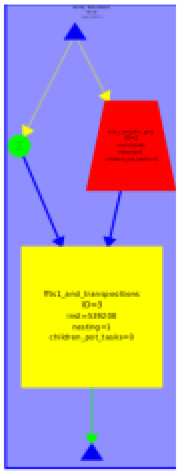


Figure 4.1 : Initial diagram

For the first version we have analysed the biggest task (the yellow one), talking in terms of time execution, and divided this into different smaller ones, for every method that the task was invoking (see figure 4.2). Even though the yellow task was divided, the execution time wasn't reduced because of the dependencies between methods. We can see on table 4.1 that in this case T_1 and T_∞ are the same and parallelism equals 1.

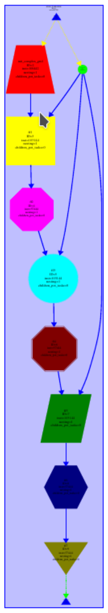
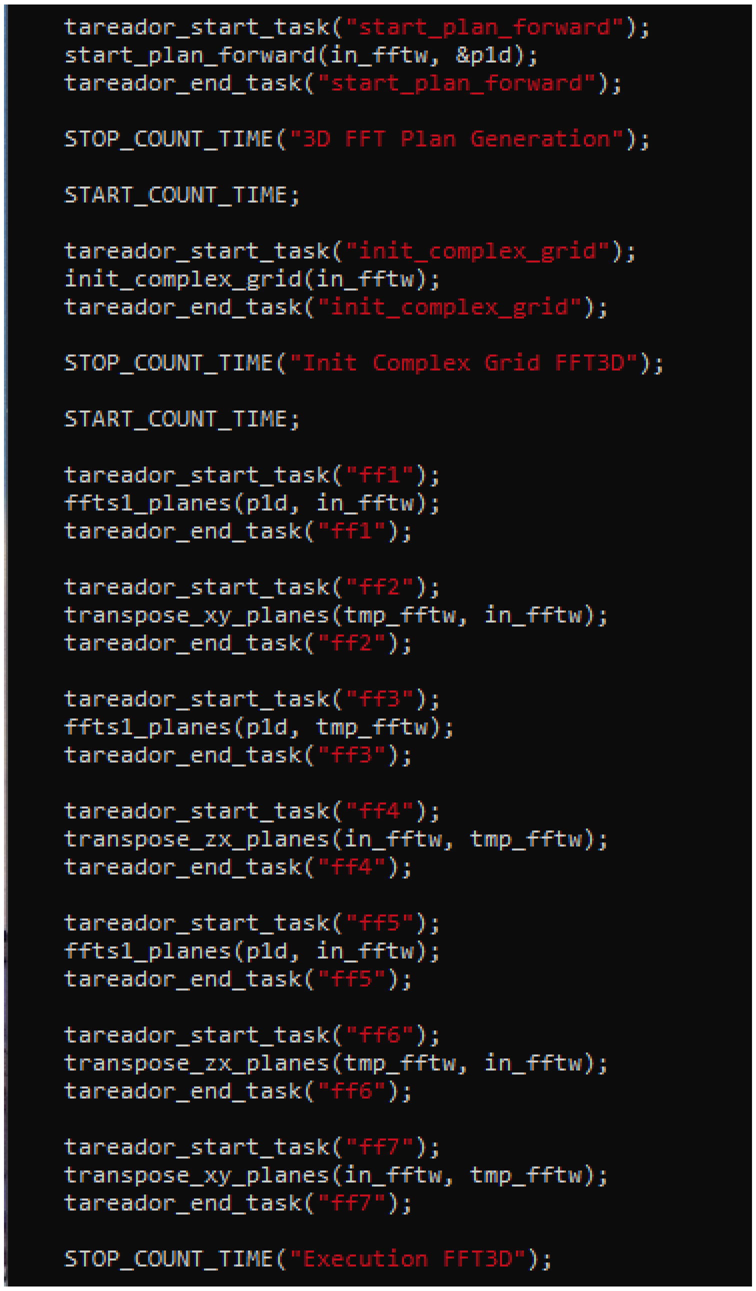


Figure 4.2 : Diagram version 1



```
    tareador_start_task("start_plan_forward");
    start_plan_forward(in_fftw, &p1d);
    tareador_end_task("start_plan_forward");

    STOP_COUNT_TIME("3D FFT Plan Generation");

    START_COUNT_TIME;

    tareador_start_task("init_complex_grid");
    init_complex_grid(in_fftw);
    tareador_end_task("init_complex_grid");

    STOP_COUNT_TIME("Init Complex Grid FFT3D");

    START_COUNT_TIME;

    tareador_start_task("ff1");
    ffts1_planes(p1d, in_fftw);
    tareador_end_task("ff1");

    tareador_start_task("ff2");
    transpose_xy_planes(tmp_fftw, in_fftw);
    tareador_end_task("ff2");

    tareador_start_task("ff3");
    ffts1_planes(p1d, tmp_fftw);
    tareador_end_task("ff3");

    tareador_start_task("ff4");
    transpose_zx_planes(in_fftw, tmp_fftw);
    tareador_end_task("ff4");

    tareador_start_task("ff5");
    ffts1_planes(p1d, in_fftw);
    tareador_end_task("ff5");

    tareador_start_task("ff6");
    transpose_zx_planes(tmp_fftw, in_fftw);
    tareador_end_task("ff6");

    tareador_start_task("ff7");
    transpose_xy_planes(in_fftw, tmp_fftw);
    tareador_end_task("ff7");

    STOP_COUNT_TIME("Execution FFT3D");
```

Figure 4.3: Changes made on v1

In version 2 , we analysed what tasks had a potentially parallelism improvement. These ones were the methods that executed a double loop, so, we created as many tasks as the number of executions of the external loop. As we see that tasks improved the execution time and as a consequence, was close to double the parallelism.

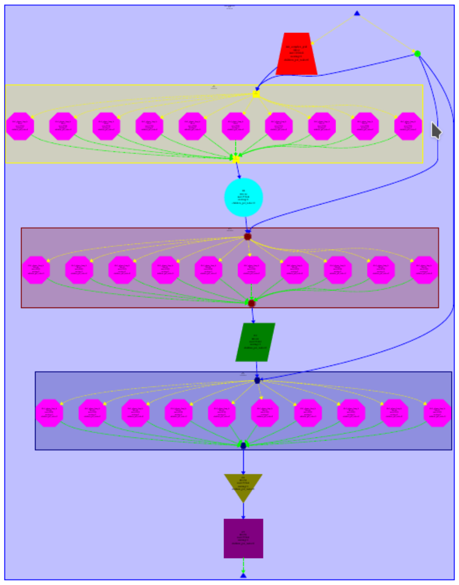
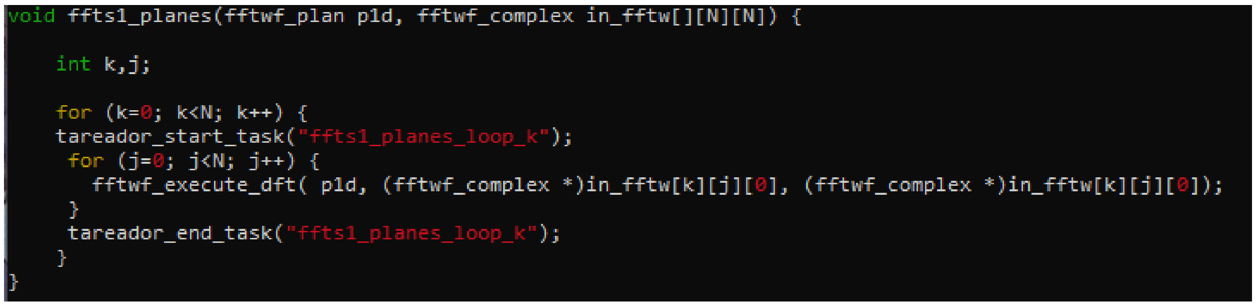


Figure 4.4 : Diagram version 2



```
void ffts1_planes(fftwf_plan p1d, fftwf_complex in_fftw[][N][N]) {

    int k,j;

    for (k=0; k<N; k++) {
    tareador_start_task("ffts1_planes_loop_k");
     for (j=0; j<N; j++) {
       fftwf_execute_dft( p1d, (fftwf_complex *)in_fftw[k][j][0], (fftwf_complex *)in_fftw[k][j][0]);
     }
     tareador_end_task("ffts1_planes_loop_k");
    }
}
```

Figure 4.5 : Changes made on v2

The next version, number 3, we applied the same improvement on version 2 to the methods called: transpose xy planes and transpose zx planes. As a result, the execution time was reduced again by more than a half (in relation to version 2) and the parallelism improved to 4,15. See figure 4.4 to see the diagram of this version.

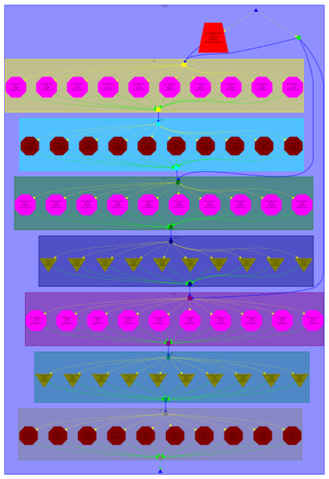
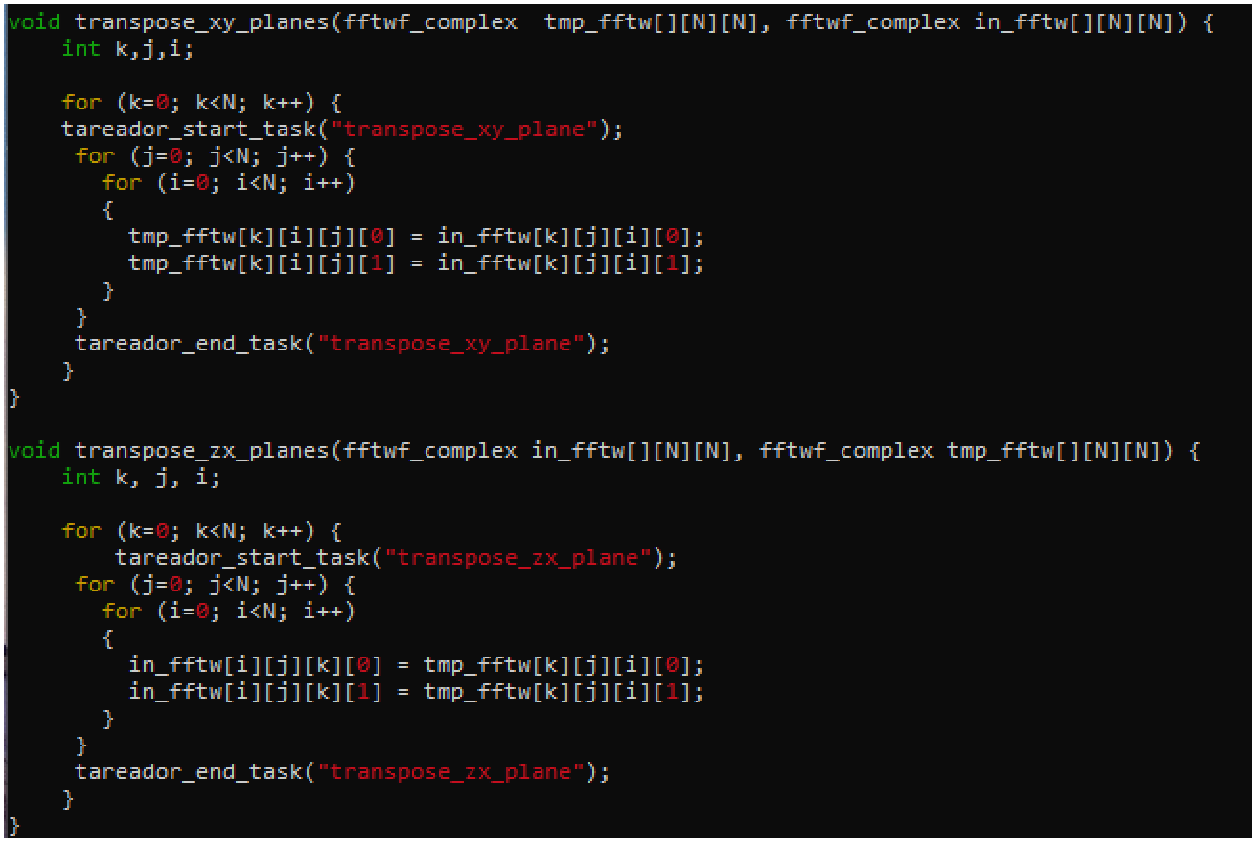


Figure 4.6 : Diagram version 3



```
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++) {
    tareador_start_task("transpose_xy_plane");
     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_plane");
    }
}

void transpose_zx_planes(fftwf_complex in_fftw[][N][N], fftwf_complex tmp_fftw[][N][N]) {
    int k, j, i;

    for (k=0; k<N; k++) {
        tareador_start_task("transpose_zx_plane");
     for (j=0; j<N; j++) {
       for (i=0; i<N; i++)
       {
         in_fftw[i][j][k][0] = tmp_fftw[k][j][i][0];
         in_fftw[i][j][k][1] = tmp_fftw[k][j][i][1];
       }
     }
     tareador_end_task("transpose_zx_plane");
    }
}
```

Figure 4.7 : Changes made on v3

Following the version 4, we are now in a situation where we improved that much for one task but the others are being a bottleneck. For that version we splitted the initial task as we did in the other methods, dividing as many tasks as iterations of the external loop.

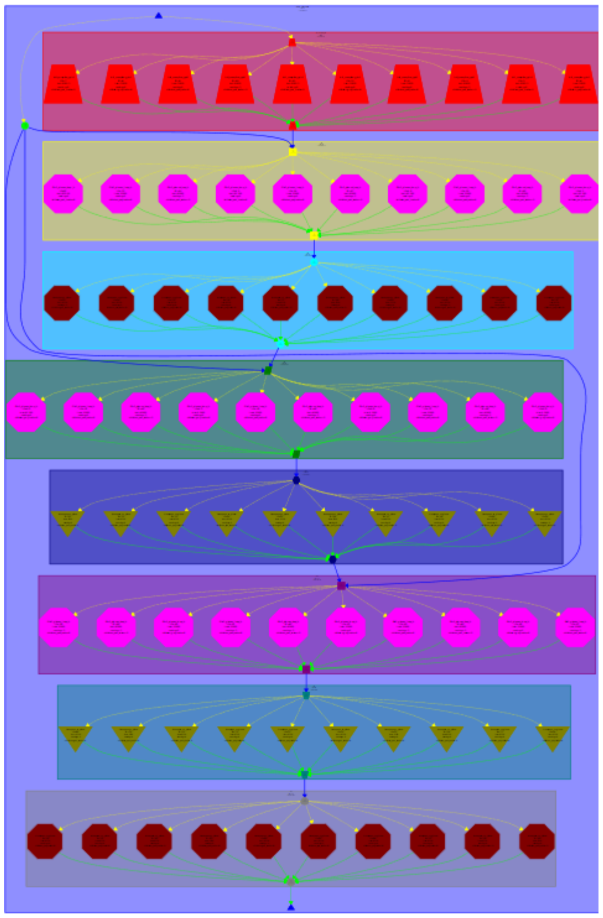
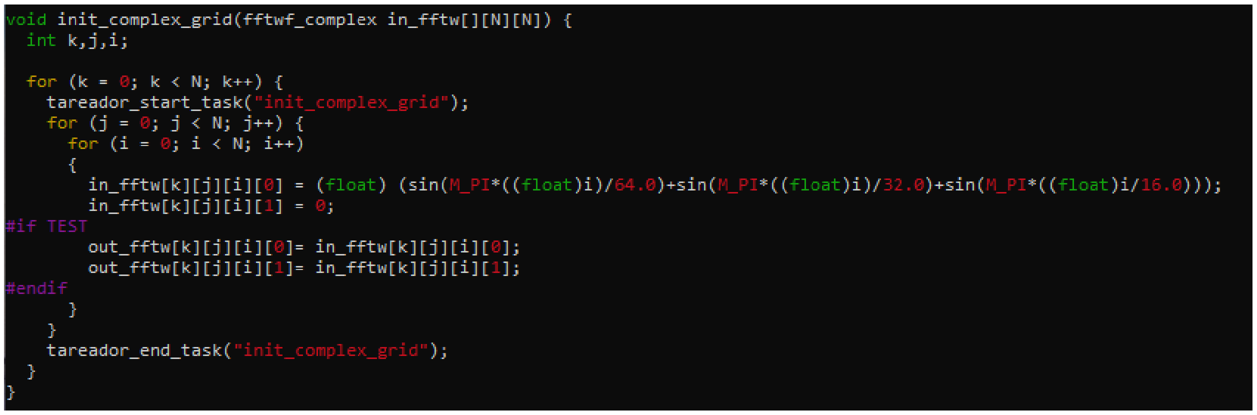


Figure 4.8 : Diagram version 4



```
void init_complex_grid(fftwf_complex in_fftw[][N][N]) {
  int k,j,i;

  for (k = 0; k < N; k++) {
    tareador_start_task("init_complex_grid");
    for (j = 0; j < N; j++) {
      for (i = 0; i < N; i++)
      {
        in_fftw[k][j][i][0] = (float) (sin(M_PI*((float)i)/64.0)+sin(M_PI*((float)i)/32.0)+sin(M_PI*((float)i/16.0)));
        in_fftw[k][j][i][1] = 0;
#if TEST
        out_fftw[k][j][i][0]= in_fftw[k][j][i][0];
        out_fftw[k][j][i][1]= in_fftw[k][j][i][1];
#endif
      }
    }
    tareador_end_task("init_complex_grid");
  }
}
```

Figure 4.9 : Changes made on v4

In addition to that version, we have worked on a table that shows the potential speedup (parallelism) as we increase the number of processors. We can see on Table 4.2 that information summarised. We can reduce the execution time to 0,64s with 16 processors, which means a speedup of almost 10 (we are reducing the time by a factor of 10). As we can see also on Table 4.2, the parallelism reaches its maximum with 16 processors.

# of procs	Execution Time	Speedup
1	6,39	-
2	3,20	1,99
4	1,92	3,33
8	1,28	4,99
16	0,64	9,98
32	0,64	9,98

Table 4.2 : Summarise of Strong Scalability version 4

Finally, on the last version of the exercise, we were allowed to apply any implementation we thought would be an improvement. As we were able to learn during the practice lesson, we should split into different tasks, the biggest ones. We decided to split all the triple loops creating a task for every external and middle loop. With these changes we improved the parallelism by 2 in relation with version 4 (see Table 4.1). Even though the improvement, the number of CPUs required was bigger than 32, that implies a high cost on hardware.

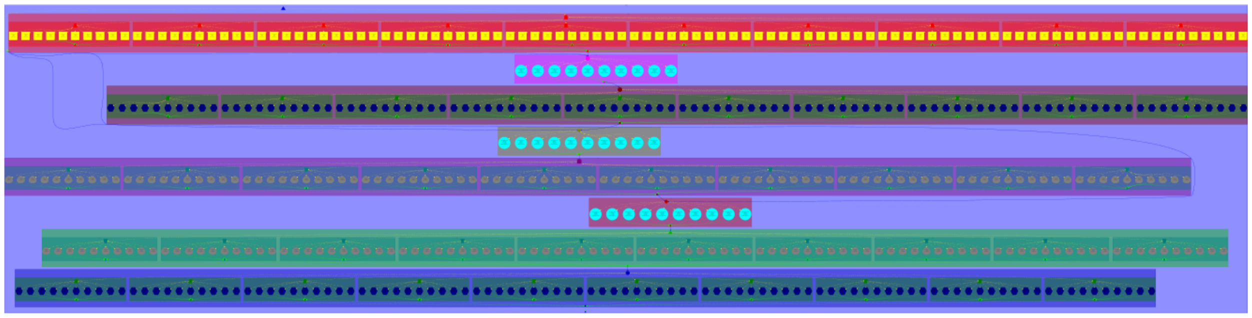
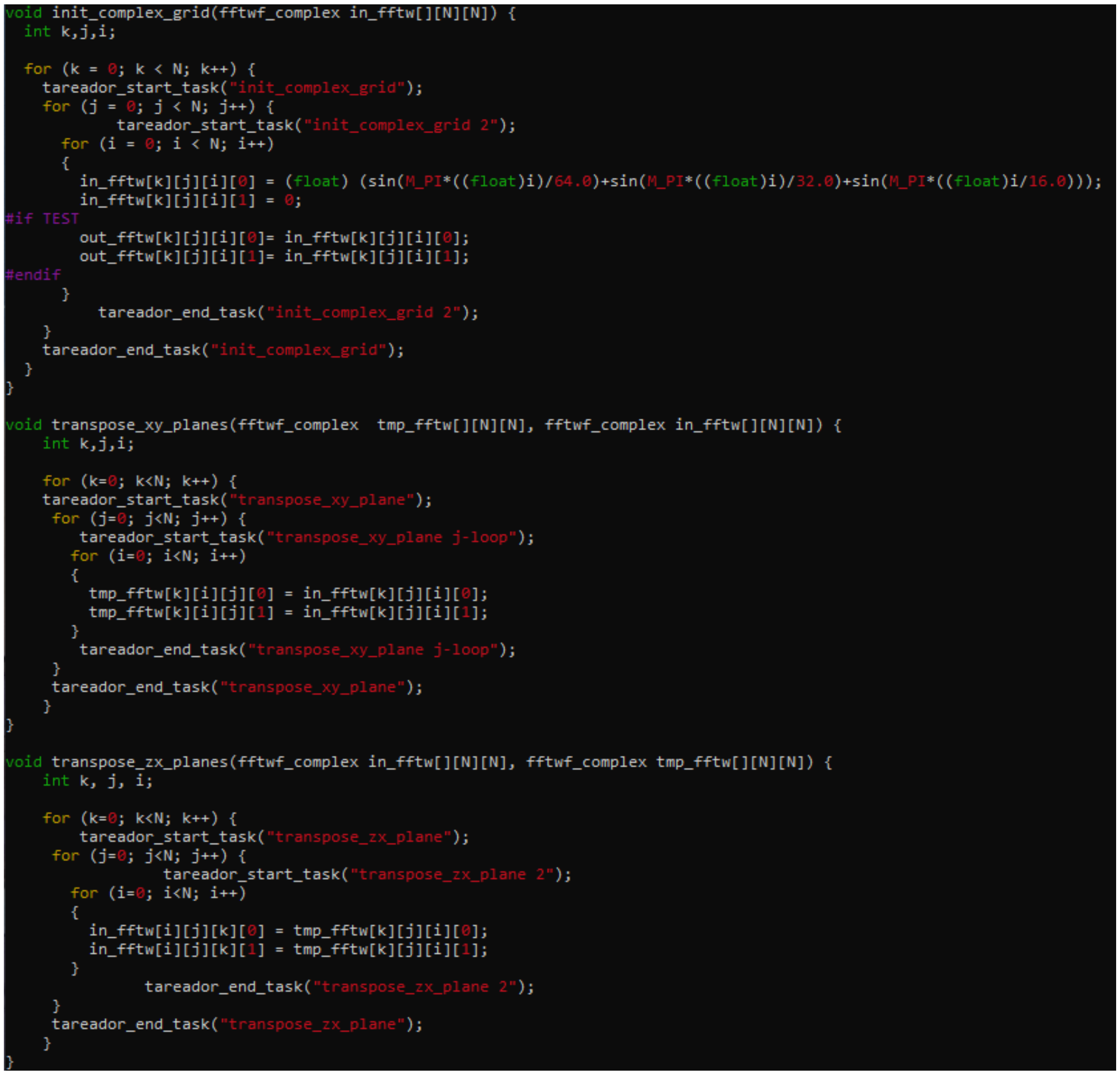


Figure 4.10 : Diagram version 5



```
void init_complex_grid(fftwf_complex in_fftw[][N][N]) {
  int k,j,i;

  for (k = 0; k < N; k++) {
    tareador_start_task("init_complex_grid");
    for (j = 0; j < N; j++) {
            tareador_start_task("init_complex_grid 2");
      for (i = 0; i < N; i++)
      {
        in_fftw[k][j][i][0] = (float) (sin(M_PI*((float)i)/64.0)+sin(M_PI*((float)i)/32.0)+sin(M_PI*((float)i/16.0)));
        in_fftw[k][j][i][1] = 0;
#if TEST
        out_fftw[k][j][i][0]= in_fftw[k][j][i][0];
        out_fftw[k][j][i][1]= in_fftw[k][j][i][1];
#endif
      }
          tareador_end_task("init_complex_grid 2");
    }
    tareador_end_task("init_complex_grid");
  }
}

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++) {
    tareador_start_task("transpose_xy_plane");
     for (j=0; j<N; j++) {
        tareador_start_task("transpose_xy_plane j-loop");
       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_plane j-loop");
     }
     tareador_end_task("transpose_xy_plane");
    }
}

void transpose_zx_planes(fftwf_complex in_fftw[][N][N], fftwf_complex tmp_fftw[][N][N]) {
    int k, j, i;

    for (k=0; k<N; k++) {
        tareador_start_task("transpose_zx_plane");
     for (j=0; j<N; j++) {
                 tareador_start_task("transpose_zx_plane 2");
       for (i=0; i<N; i++)
       {
         in_fftw[i][j][k][0] = tmp_fftw[k][j][i][0];
         in_fftw[i][j][k][1] = tmp_fftw[k][j][i][1];
       }
               tareador_end_task("transpose_zx_plane 2");
     }
     tareador_end_task("transpose_zx_plane");
    }
}
```

Figure 4.11 : Changes made on v5

Here below, we can see Table 4.3 summarising the scalability on version 5. As we have commented, there exists a big improvement on parallelism but as a consequence, it depends on a huge number of CPUs.

# of procs	Execution time	Speedup
1	6,39	-
2	3,20	1,99
4	1,76	3,63
8	1,04	6,14
16	0,54	11,83
32	0,44	14,52

Table 4.3 : Summarise of Strong Scalability version 5

Maria Montalvo Falcón - par4214
Victor Pla Sanchís - par4219
2021-2022 Q2
Group: 42

THIRD SESSION

5. Understanding the parallel execution of *3DFFT*

In this final section of the first laboratory, we are going to be introduced into a new tool named Paraver. With this new tool, we will have the possibility to analyse a parallel application in OpenMP in a quality and quantitative way, and visualise the changes and characteristics of our code.

We have a wide range of tables and histograms that could be generated, such as:

1) **Implicits tasks:**
   a) *New window*: shows us the thread state.

      We can distinguish different meanings in colours: running (blue), synchronisation (red) and scheduling fork (yellow).

   b) *Parallel construct:* shows us the activation of the parallel region with colour red.
   c) *Worksharing construct:* shows us when the single region is activated
   d) *Implicit tasks in parallel constructs:* shows us the task line number
   e) *Tasks duration:* shows us the larger ranges of time in tonalities of blue, whether the samallers are in colour green.
2) **Explicit tasks**
   a) *Explicit tasks executed duration*
   b) *Explicit tasks function created and executed*

With these different types of support, we are able to analyse the code and explain the results.

Here below, we can see Table 5.1 with the summarised information of each version.

Version	φ	**ideal** S_8	T_1	T_8	**real** S_8
initial version in 3dfft omp.c	0,67	2,41	2,46s	1,41s	1,74
new version with improved φ	0,91	4,91	2,45s	0,88s	2,78
final version with reduced parallelisation overheads	0,87	4,18	2,48s	0,66s	3,75

Table 5.1. Summarised information extracted from Paraver data

$$\varphi = \frac{T_{par}}{T_1}$$

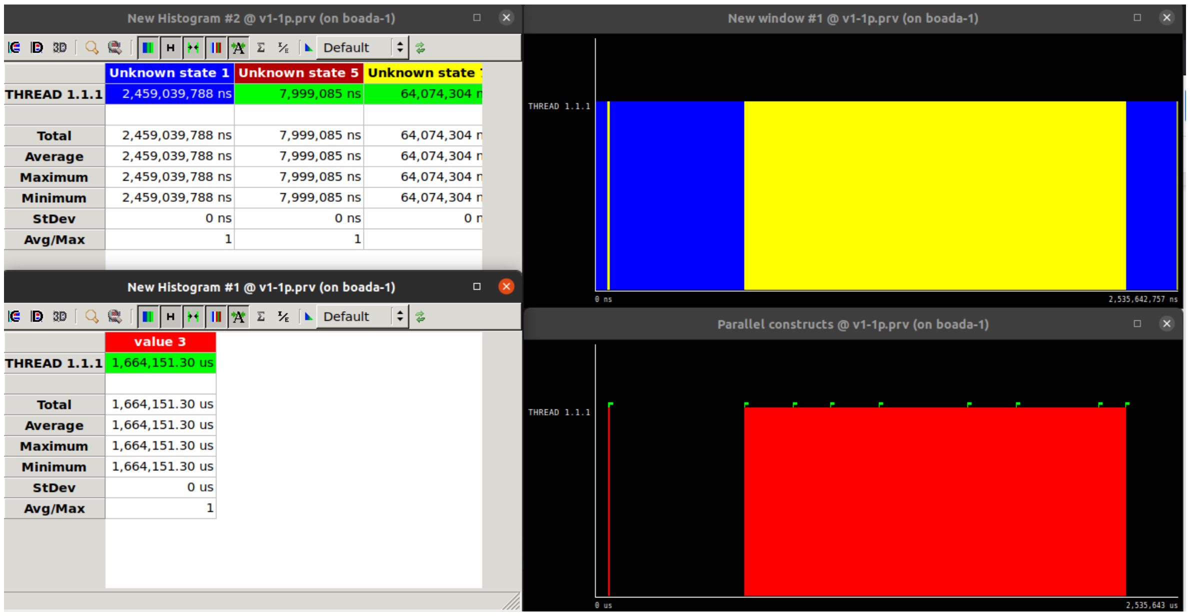
$$Real\ S_8 = \frac{T_1}{T_8}$$

$$Ideal\ S_8 = \frac{1}{(1-\varphi) + \frac{\varphi}{8}}$$

$$T_1 = T_{seq} + T_{par}$$

Maria Montalvo Falcón - par4214
Victor Pla Sanchís - par4219
2021-2022 Q2
Group: 42

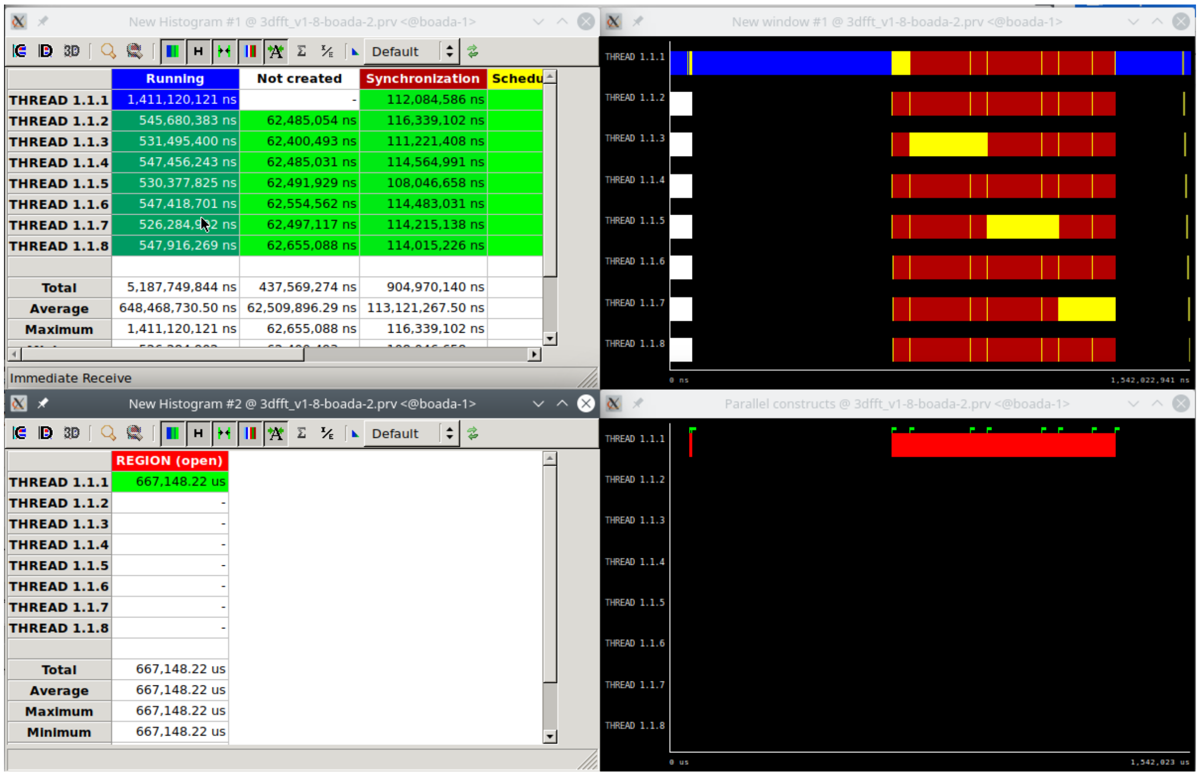
The initial program was written parallelising the methods: *transpose_xy_planes*, *transpose_zx_planes* and *ffts1_planes*. We see that the program parallelism increases by little the time execution of the sequential program, at all, the speedup is lower than 2.


New Histogram #2 @ v1-1p.prv (on boada-1)
New window #1 @ v1-1p.prv (on boada-1)
THREAD 1.1.1
0 ns
2,535,642,757 ns
New Histogram #1 @ v1-1p.prv (on boada-1)
Parallel constructs @ v1-1p.prv (on boada-1)
THREAD 1.1.1
0 us
2,535,643 us
Default

	Unknown state 1	Unknown state 5	Unknown state
THREAD 1.1.1	2,459,039,788 ns	7,999,085 ns	64,074,304 n
Total	2,459,039,788 ns	7,999,085 ns	64,074,304 n
Average	2,459,039,788 ns	7,999,085 ns	64,074,304 n
Maximum	2,459,039,788 ns	7,999,085 ns	64,074,304 n
Minimum	2,459,039,788 ns	7,999,085 ns	64,074,304 n
StDev	0 ns	0 ns	0 n
Avg/Max	1	1	

	value 3
THREAD 1.1.1	1,664,151.30 us
Total	1,664,151.30 us
Average	1,664,151.30 us
Maximum	1,664,151.30 us
Minimum	1,664,151.30 us
StDev	0 us
Avg/Max	1

Figure 5.1. Graphics and histogram of initial version in 3dfft omp.c with one processor

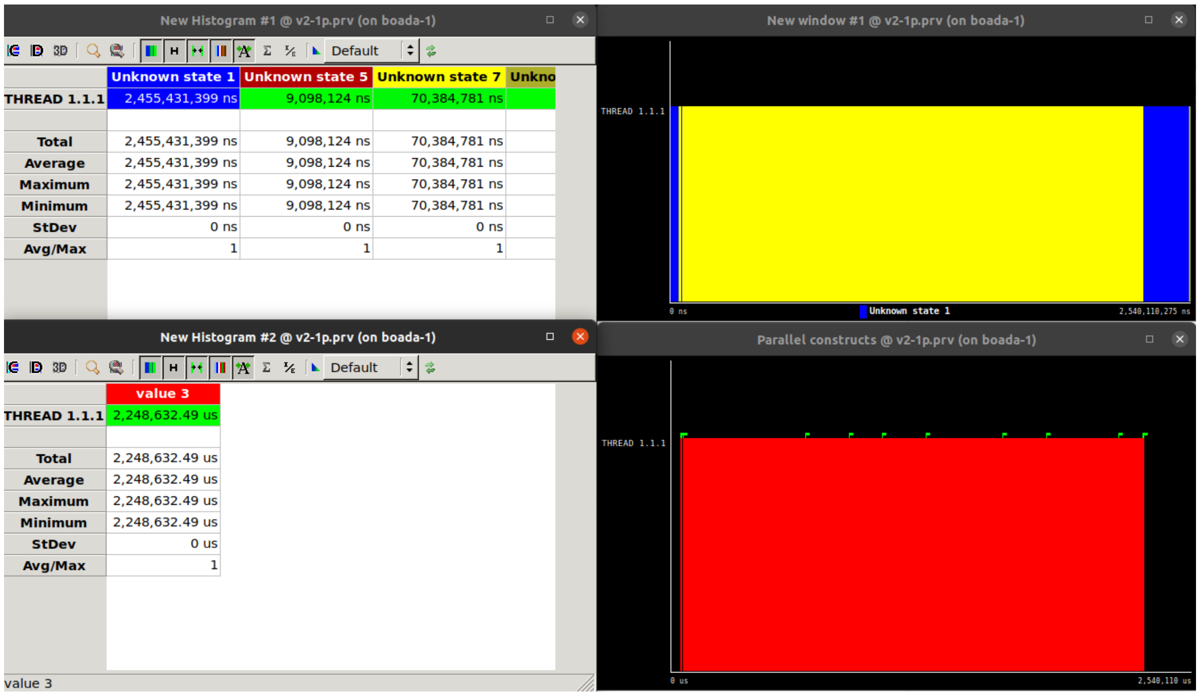

New Histogram #1 @ 3dfft_v1-8-boada-2.prv <@boada-1>
New window #1 @ 3dfft_v1-8-boada-2.prv <@boada-1>
THREAD 1.1.1
THREAD 1.1.2
THREAD 1.1.3
THREAD 1.1.4
THREAD 1.1.5
THREAD 1.1.6
THREAD 1.1.7
THREAD 1.1.8
0 ns
1,542,822,941 ns
Immediate Receive
New Histogram #2 @ 3dfft_v1-8-boada-2.prv <@boada-1>
Parallel constructs @ 3dfft_v1-8-boada-2.prv <@boada-1>
0 us
1,542,823 us
Default

	Running	Not created	Synchronization	Schedu
THREAD 1.1.1	1,411,120,121 ns	-	112,084,586 ns	
THREAD 1.1.2	545,680,383 ns	62,485,054 ns	116,339,102 ns	
THREAD 1.1.3	531,495,400 ns	62,400,493 ns	111,221,408 ns	
THREAD 1.1.4	547,456,243 ns	62,485,031 ns	114,564,991 ns	
THREAD 1.1.5	530,377,825 ns	62,491,929 ns	108,046,658 ns	
THREAD 1.1.6	547,418,701 ns	62,554,562 ns	114,483,031 ns	
THREAD 1.1.7	526,284,9[illegible]2 ns	62,497,117 ns	114,215,138 ns	
THREAD 1.1.8	547,916,269 ns	62,655,088 ns	114,015,226 ns	
Total	5,187,749,844 ns	437,569,274 ns	904,970,140 ns	
Average	648,468,730.50 ns	62,509,896.29 ns	113,121,267.50 ns	
Maximum	1,411,120,121 ns	62,655,088 ns	116,339,102 ns	

	REGION (open)
THREAD 1.1.1	667,148.22 us
THREAD 1.1.2	-
THREAD 1.1.3	-
THREAD 1.1.4	-
THREAD 1.1.5	-
THREAD 1.1.6	-
THREAD 1.1.7	-
THREAD 1.1.8	-
Total	667,148.22 us
Average	667,148.22 us
Maximum	667,148.22 us
Minimum	667,148.22 us

Figure 5.2. Graphics and histogram of initial version in 3dfft omp.c with eight processors

In a way to improve that parallelism, the second version of this program was parallelising the method *init_complex_grid* in the same way, the last one that was not parallelized yet. As we see on the figures 5.1-5.4, the parallelizable time increases, as well ***φ***, and the time execution gets better. Finally we improved the real speed-up by 1.


New window #1 @ v2-1p.prv (on boada-1)
THREAD 1.1.1
0 ns
Unknown state 1
2,540,110,275 ns
Parallel constructs @ v2-1p.prv (on boada-1)
THREAD 1.1.1
0 us
2,540,110 us

New Histogram #1 @ v2-1p.prv (on boada-1)

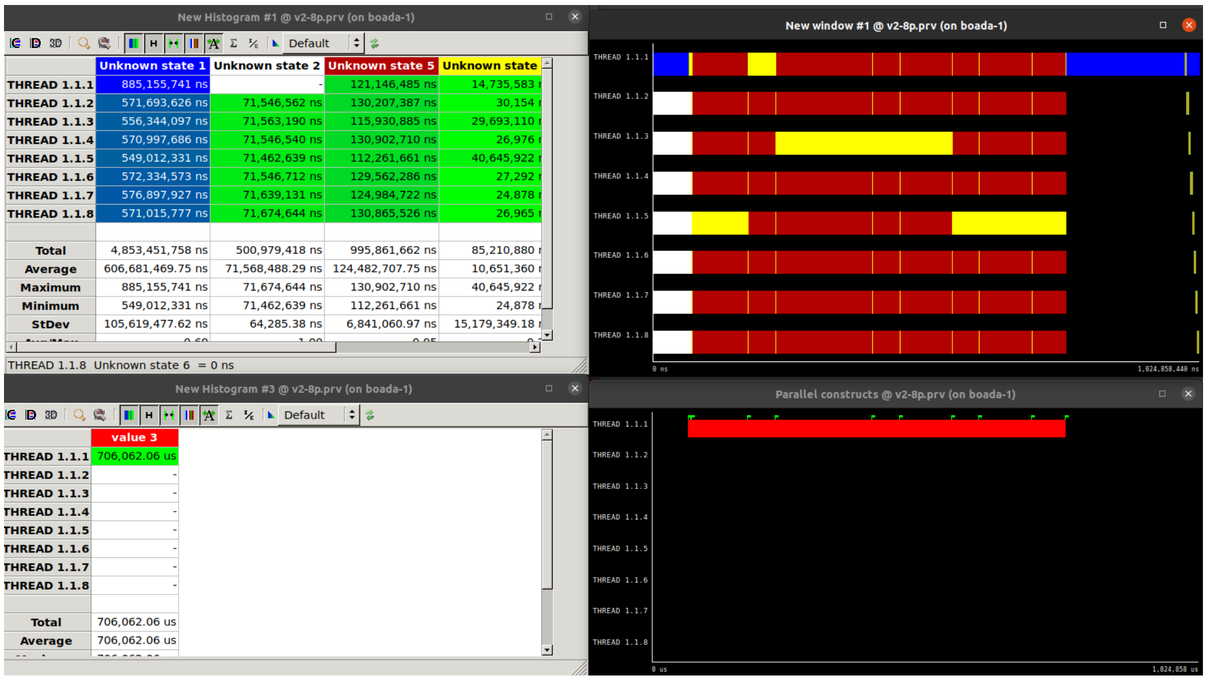
	Unknown state 1	Unknown state 5	Unknown state 7	Unkno
THREAD 1.1.1	2,455,431,399 ns	9,098,124 ns	70,384,781 ns	
Total	2,455,431,399 ns	9,098,124 ns	70,384,781 ns	
Average	2,455,431,399 ns	9,098,124 ns	70,384,781 ns	
Maximum	2,455,431,399 ns	9,098,124 ns	70,384,781 ns	
Minimum	2,455,431,399 ns	9,098,124 ns	70,384,781 ns	
StDev	0 ns	0 ns	0 ns	
Avg/Max	1	1	1	

New Histogram #2 @ v2-1p.prv (on boada-1)

	value 3
THREAD 1.1.1	2,248,632.49 us
Total	2,248,632.49 us
Average	2,248,632.49 us
Maximum	2,248,632.49 us
Minimum	2,248,632.49 us
StDev	0 us
Avg/Max	1

value 3

Figure 5.3 Graphics and histogram of version with improved φ with one processor



New Histogram #1 @ v2-8p.prv (on boada-1)

	Unknown state 1	Unknown state 2	Unknown state 5	Unknown state
THREAD 1.1.1	885,155,741 ns	-	121,146,485 ns	14,735,583
THREAD 1.1.2	571,693,626 ns	71,546,562 ns	130,207,387 ns	30,154
THREAD 1.1.3	556,344,097 ns	71,563,190 ns	115,930,885 ns	29,693,110
THREAD 1.1.4	570,997,686 ns	71,546,540 ns	130,902,710 ns	26,976
THREAD 1.1.5	549,012,331 ns	71,462,639 ns	112,261,661 ns	40,645,922
THREAD 1.1.6	572,334,573 ns	71,546,712 ns	129,562,286 ns	27,292
THREAD 1.1.7	576,897,927 ns	71,639,131 ns	124,984,722 ns	24,878
THREAD 1.1.8	571,015,777 ns	71,674,644 ns	130,865,526 ns	26,965
Total	4,853,451,758 ns	500,979,418 ns	995,861,662 ns	85,210,880
Average	606,681,469.75 ns	71,568,488.29 ns	124,482,707.75 ns	10,651,360
Maximum	885,155,741 ns	71,674,644 ns	130,902,710 ns	40,645,922
Minimum	549,012,331 ns	71,462,639 ns	112,261,661 ns	24,878
StDev	105,619,477.62 ns	64,285.38 ns	6,841,060.97 ns	15,179,349.18

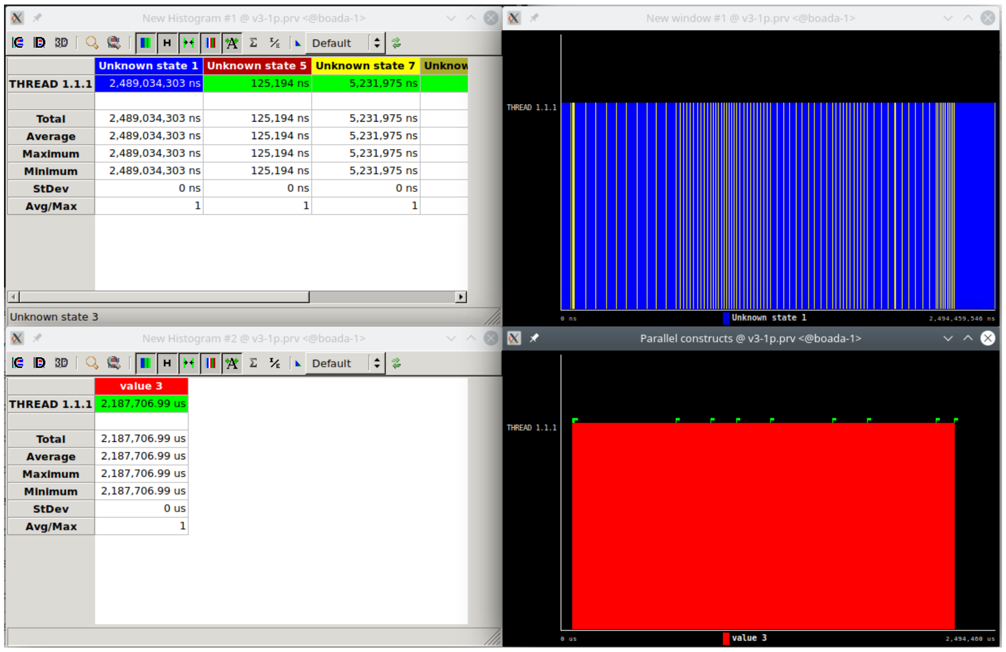
THREAD 1.1.8 Unknown state 6 = 0 ns

New Histogram #3 @ v2-8p.prv (on boada-1)

	value 3
THREAD 1.1.1	706,062.06 us
THREAD 1.1.2	-
THREAD 1.1.3	-
THREAD 1.1.4	-
THREAD 1.1.5	-
THREAD 1.1.6	-
THREAD 1.1.7	-
THREAD 1.1.8	-
Total	706,062.06 us
Average	706,062.06 us

Figure 5.4 Graphics and histogram of version with improved φ with eight processors

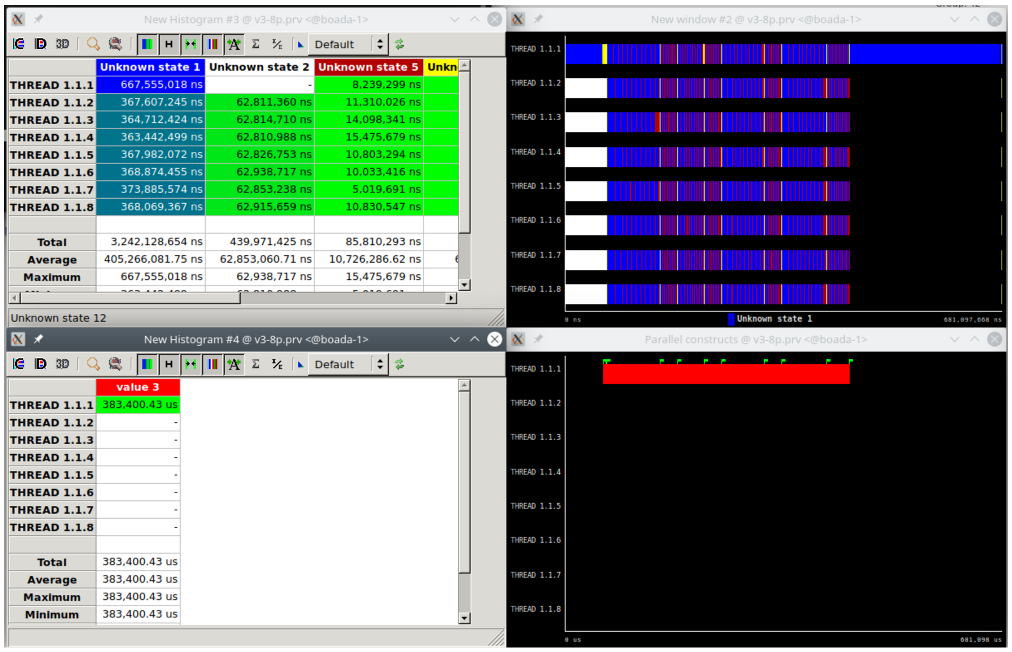
The last version was dedicated to decrease the overhead on the second version of the program. The parallelizable time doesn't change a lot between last versions, as well ***φ*** doesn't change. It makes sense as we are not changing the code to improve the ratio of parallel time, but to decrease the time of creation of the tasks, reducing the program, etc. As a consequence, the speed-up increases because we reduced the overhead time, that's because the pareaver calls were now outside of the loop.


New Histogram #1 @ v3-1p.prv <@boada-1>
Default
New window #1 @ v3-1p.prv <@boada-1>
THREAD 1.1.1
Unknown state 1
Unknown state 3
New Histogram #2 @ v3-1p.prv <@boada-1>
Parallel constructs @ v3-1p.prv <@boada-1>
value 3

	Unknown state 1	Unknown state 5	Unknown state 7	Unknow
THREAD 1.1.1	2,489,034,303 ns	125,194 ns	5,231,975 ns	
Total	2,489,034,303 ns	125,194 ns	5,231,975 ns	
Average	2,489,034,303 ns	125,194 ns	5,231,975 ns	
Maximum	2,489,034,303 ns	125,194 ns	5,231,975 ns	
Minimum	2,489,034,303 ns	125,194 ns	5,231,975 ns	
StDev	0 ns	0 ns	0 ns	
Avg/Max	1	1	1	

	value 3
THREAD 1.1.1	2,187,706.99 us
Total	2,187,706.99 us
Average	2,187,706.99 us
Maximum	2,187,706.99 us
Minimum	2,187,706.99 us
StDev	0 us
Avg/Max	1

Figure 5.5 Graphics and histogram of final version with one processor


New Histogram #3 @ v3-8p.prv <@boada-1>
Default
New window #2 @ v3-8p.prv <@boada-1>
THREAD 1.1.1
THREAD 1.1.2
THREAD 1.1.3
THREAD 1.1.4
THREAD 1.1.5
THREAD 1.1.6
THREAD 1.1.7
THREAD 1.1.8
Unknown state 1
Unknown state 12
New Histogram #4 @ v3-8p.prv <@boada-1>
Parallel constructs @ v3-8p.prv <@boada-1>

	Unknown state 1	Unknown state 2	Unknown state 5	Unkn
THREAD 1.1.1	667,555,018 ns	-	8,239,299 ns	
THREAD 1.1.2	367,607,245 ns	62,811,360 ns	11,310,026 ns	
THREAD 1.1.3	364,712,424 ns	62,814,710 ns	14,098,341 ns	
THREAD 1.1.4	363,442,499 ns	62,810,988 ns	15,475,679 ns	
THREAD 1.1.5	367,982,072 ns	62,826,753 ns	10,803,294 ns	
THREAD 1.1.6	368,874,455 ns	62,938,717 ns	10,033,416 ns	
THREAD 1.1.7	373,885,574 ns	62,853,238 ns	5,019,691 ns	
THREAD 1.1.8	368,069,367 ns	62,915,659 ns	10,830,547 ns	
Total	3,242,128,654 ns	439,971,425 ns	85,810,293 ns	
Average	405,266,081.75 ns	62,853,060.71 ns	10,726,286.62 ns	6
Maximum	667,555,018 ns	62,938,717 ns	15,475,679 ns	

	value 3
THREAD 1.1.1	383,400.43 us
THREAD 1.1.2	-
THREAD 1.1.3	-
THREAD 1.1.4	-
THREAD 1.1.5	-
THREAD 1.1.6	-
THREAD 1.1.7	-
THREAD 1.1.8	-
Total	383,400.43 us
Average	383,400.43 us
Maximum	383,400.43 us
Minimum	383,400.43 us

Figure 5.6 Graphics and histogram of final version with eight processors

Maria Montalvo Falcón - par4214
Victor Pla Sanchís - par4219
2021-2022 Q2
Group: 42

6. Conclusions

Along these three first sessions of laboratory one, we have seen how parallelism could be an advantage or become a disadvantage if you don’t weigh it up the pros and cons of a program. We have seen that every problem is unique and we have to divide and analyse every point in order to make a good improvement with parallelism.

In the first session, we have studied how the environment is distributed and the limitations it has. We have seen that in boada-1 only one thread can do real work whether in boada-4 to boada-8 we can use till 12 threads. Therefore, in this session we’ve studied how to analyse a problem with strong scalability (increasing the number or procs to reduce time execution and remaining the size of the problem) and weak scalability (increasing the size of the problem with number of procs in order to remain the time equal) and how in a extreme situation parallelism also has its limitations. All these things put together implies that architecture could develop a great paper in performance of a program.

For the second session, we have analysed with the tool Tareador, how important is the division of the tasks. In this case, taking into account the size, we have to be concerned about remaining the proportion between tasks and not to leave the majority of the program to only one task (in this case the parallelism would not be useful). In this session we have seen that the division into tasks could lower the time execution, but it also has its limitations. We have to be careful with dependencies between tasks, or data dependencies, etc. With Tareador, it is possible to graphically see all this information.

Finally we have seen in the last session with the help of Paraver, how important it is to study our program and its barriers to optimise it. It is so important to analyse the problem before starting parallelising because due to this, we could achieve such different results. As we have seen, one of the most dominant constraints is the time that could be parallelised in the program. Beyond this, we have also to take into account the overhead created when the program is parallelised, because sometimes, could be improved, but sometimes could not be worth parallelising the program in some way.

Summing up, parallel programming is a great tool that engineers have to work with, but mostly understand; without understanding the results could take us out of the best solution.