Universitat Politècnica de Catalunya

Parallelism

Lab 1: Experimental setup and tools

Carlota Catot Bragós
Ferran Martínez Felipe
PAR1110

Quadrimestre Primavera 2017-2018

Contents

1 Introduction

One of the first problems that every person encounters when tries to learn a new skill is the fact that he has to learn a lot of new concepts about the subject that he is going to be introduced.

This is the scope of the first of the sessions of PAR laboratory, introduce the students to the Boada architecture, some of the most important paralellism concepts, the *Tareador* and *Paraver* software and so on.

It is for that reason that in the following chapters we are going to explain all of the concepts needed to do the next laboratory sessions, step by step.

2 Node architecture and memory

Talking about parallelism is talk about the architecture of the cluster that it is going to be used to run the parallel programs. This is the reason that the first thing that we need to do is obtaining the information about each of the different Boada nodes.

Boada is a cluster divided in different nodes, each of them with different architecture and different function. For the scope of PAR subject, the nodes available goes from boada-1 to boada-8. In addition, the students only have access to boada-1, and to run programs on the other nodes the cluster have a queue system that allows to choose the architecture in which the program will be run (but not the core, the user just chooses the architecture in which the program will be run).

The easiest way to obtain the information of the hardware used in each node is using the linux commands **lscpu and lstopo**. This commands can be easily applied in the boada-1 node, because is the one used by the user, but in the other nodes a .sh scripts must be created (in our case by modifying the submit-*.sh scripts provided by the PAR teachers).

After creating the scripts and applying them to each of the nodes, we obtained the following hardware information:

	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	2395.0	2600	1700.0
L1-I cache size (per-core)	32K	32K	32K
L1-D cache size (per-core)	32K	32K	32K
L2 cache size (per-core)	256K	256K	256K
Last-level cache size (per-socket)	12288K	15360K	20480K
Main memory size (per socket)	12 GB	31GB	16GB
Main memory size (per node)	23 GB	63 GB	16 GB

The information above is really useful to start understanding about what parallelism is, but in order to make easier the first approach to the boada cluster one of the things that we can do is print the architectural diagram of each of the nodes, we can see it in figure 1 (this figure only shows the architecture of the boada-1).

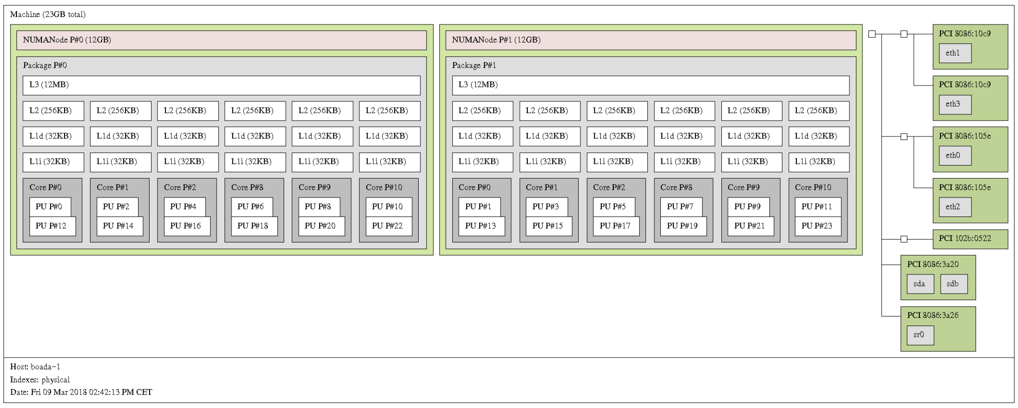


Figure 1: Diagram created by using **lstopo –of fig map.fig**

3 Timing sequential and parallel executions

Althought the fact that paralellism can be useful in different contexts, the main advantatge of this technique is speed-up applications. However, this increase of performance can be achieved in several ways, two of the most importants of them are **strong** and **weak** scalability.

In the following sections we are going to analize this two approaches by applying them to the same program (one which calculate an approach of number pi, called pi_omp.c)

3.1 Strong Scalability

Strong scalability is a technique that consists in increase the number of processors for a problem of a fixed size. That means that in an ideal world where a program could be paralellized into infinite processors this technique would achieve an infinite speed-up. Nevertheless the real world doesn't work in this way, and increasing the number of processors available to a parallel program doesn't increase linearly the speed-up in a ratio 1:1.

Now we are going to compare the execution of pi_omp.c in the 3 different architectures that are available in Boada.

3.1.1 Boada 1-4

The agrupation of the nodes 1 to 4 (and the agrupation of the nodes 6 to 8) is reasonable if we compare the architecture of each of the nodes available in boada. This two groups of nodes have the same architecture, and is reasonable to extrapolate the results obtained in one of the nodes to all the group.

First of all, the first measure of performance that we can use is to compare the execution time versus the number of threads used. If we look at the plot of time vs threads (see figure 2) we can see that the execution time decays logarithmically as we increase the number of threads. That fact basically means that exists a certain number of threads as of the execution time doesn't decrease anymore. It is therefore easy to think that exists an overhead in increasing the number of threads used to execute the program that denies the execution time to decrease infinitely.

After analizing the first plot, the next thing that we can do is analyze the speed-up of the program against the number of threads used (see figure **??**). In this case we can observe the same as commented in the previous plot: The speed-up obtained by parallelism doesn't increase linearly with the number of threads used, and reaches its limit at 11 threads, moment in which the speed-up starts to decrease.

That behaviour is probably obtained because the overhead produced by dividing the program in parallel tasks, which appears to increase linearly with the number of threads (because if that was not the case, the line of the real speed-up obtained should be an straight line).

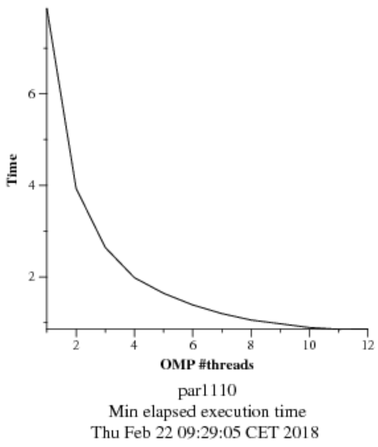


Figure 2: Time vs. Threads

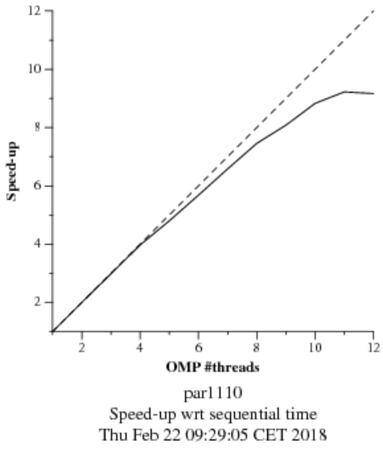


Figure 3: Speed-Up

3.1.2 Boada 5

The first thing to comment about the node prepared to use **cuda** is that all the explained for the previous nodes can be applied to this node (and to the next ones). However, we have to comment that this architecture appears to be better to obtain parallelism, at the moment that if we look the plot of speed-up against threads (figure 5) we can see that for the number of threads plotted the speed-up line never decreases.

About the execution time (figure 5) is easy to see that the behaviour of the execution time against threads is the same than the observed in the previous node (decays logarithmically at nearly the same rate).

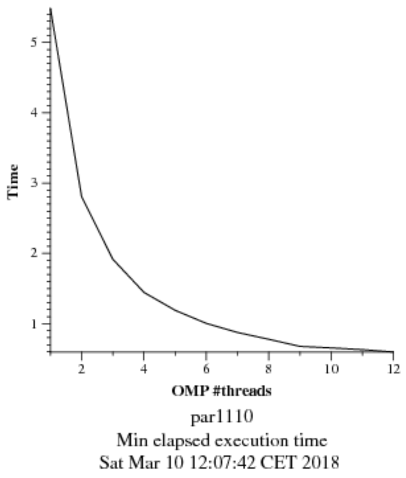


Figure 4: Time vs. Threads

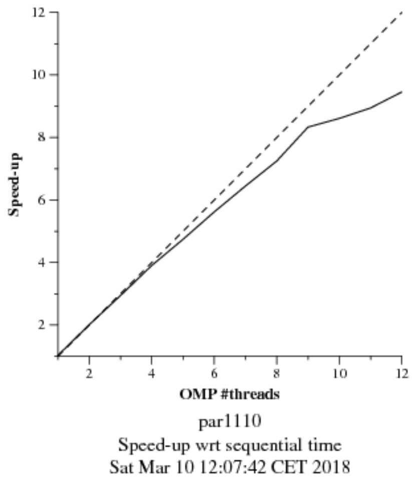


Figure 5: Speed-Up

3.1.3 Boada 6-8

It this point all of the things to comment about strong scalability has been said before. The only point that we must comment is that in this node the speed-up (figure 7) seems to increase linearly (but not at a rate of 1:1) with the number of threads. This is hard to believe, because the overheads commented in the previous architectures exist here too. This is why the only conclusion that we can obtain with this plot is that the decay of the speed-up must be produced with a higher number of threads, outside the scope of the plot.

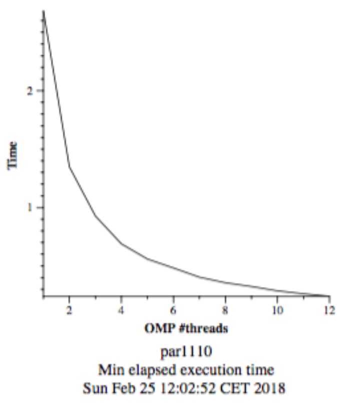


Figure 6: Time vs. Threads

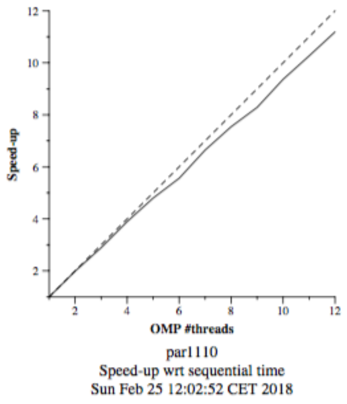


Figure 7: Speed-Up

In conclusion, for Strong Scalability we can see that for the Time vs. Threads plots (figure 2 for Boada 1-4, figure 4 for Boada 5 and figure 6 for Boada 6) we can not apreciate any relevant difference for the different arquitectures of Boada, but for the Speed-Up plot (figure 3 for Boada 1-4, figure 5 for Boada 5 and figure 7 for Boada 6) we can see that there is a little change between the differents arquitectures and the speed-up is better for the Boada 5 than Boada 1-4 and Boada 6 is better than Boada 5.

3.2 Weak Scalability

Weak scalability is a technique that consists in increase the number of processors at the same time that we increase the size of the problem. This technique is used when strong scalability is not possible, to increase the number of computations done in an instant.

In the following sections we are going to comment the results of applying this technique for the 3 different architectures that are available in Boada.

3.2.1 Boada 1-4

As we can see in the plot of speed-up against threads weak scalability (figure 8) does not increase the problem speed-up. This technique only allows to increase the size of the program computed, but doesn't decrease the execution time itself.

Nevertheless, we can observe that the speed-up of the program decreases as we can increase the number of threads, produced by the same overhead that we commented in the strong scalability section.

3.2.2 Boada 5

The behaviour of boada 5 is similar to boada 1-4. If we look at the plot of speed-up against threads (figure 9) we can see that the speed-up never increases (in fact, it only decreases a little bit). That behaviour looks like has to be produced to the overhead produced by generate parallelism, which has to be dependant of the number of threads used (which explains why the speed-up decreases a little bit as we increase the number of threads).

However, the critical point in which the overhead produces a big decrease of speed-up appears to be outside the scope of the plot.

3.2.3 Boada 6-8

At this point all the points to comment about weak scalability have been said. The only thing that we can comment is the same that we explained for this node in the strong scalability section: the decrease of the speed-up produced by the overhead has to exist outside the scope of the plot (figure 10), because if we only take into account the information of the plot we can see that the speed-up in mean doesn't decrease.

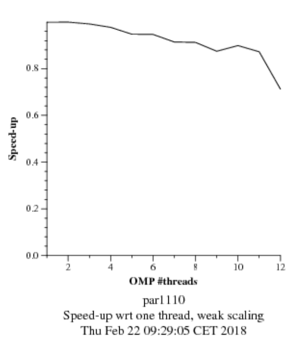


Figure 8: Boada 1-4

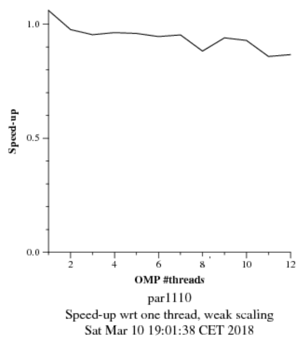


Figure 9: Boada 5

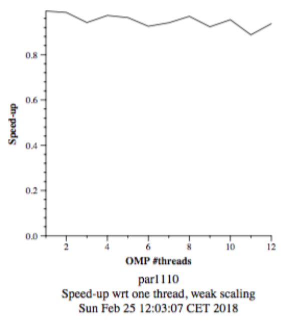


Figure 10: Boada 6-8

In conclusion, we can see that after analyzing the plots 8), 9), 10) we can obtain the same conclusions as the obtained in the strong scalability section. The speed-up obtained by weak scalability starts to fall in boada 1-4 with less number of threads than in boada 5 and boada 5-8. We cannot make the comparision between boada 5 and boada 6-8 with the range of number of threads shown in the plots because is not bigger enought to appreciate the fall of the speed-up.

4 Analysis of task decompositions with *Tareador*

For analyze the task descompositions we are using a tool called *Tareador* witch help us to see diferent things like the dependence graph, the visualisation of data involved in task dependencies and also with *Paraver*, we can view the simulation of the execution with diferents processors, seeing the different traces. To be more familiar with the software presented, we will work with the provided code called **3dfft_seq.c** making little changes in 4 different versions, to discover more parallelism en each one. At the end we will compare them in Table 1 with the data for each version.

4.1 Original Version

In the original code, we see that the *Tareador* starts before all the calls to the diferent functions and ends after all the calls. We can see the code in Figure 11.

For starting the manual calculation, first we need to know the time to execute one instruction. This time can be extracted from figure 12 (the total number of instructions at the top of the image) and from figure 13 (the total time for all the processors, at the bottom left of the imatge). With these 2 data, we can supose that the time to execute one instruction is 593.772.000/593.772 = 1.000. This ratio will be the same for all the versions that we are going to see in the next parts. In the dependence graph (figure 12) we can see that there are 2 different paths. We are going to do the calculations for T_1, T_∞ and the Paralellism. For calculating T_∞ we need to calculate the number of instructions for the critical path. In the dependence graph we can see the size of the tasks proporcionally with the number of instructions they have, this help us to know the critical path.

$$T_1 = totalInstructions * 1.000 = \mathbf{593.772.000}$$

$$T_\infty = (108 + 54.442 + 539.208) * 1.000 = \mathbf{593.758.000}$$

However, we see in the execution with 1 processador from paraver trace (figure 13), that the time for all the instructions is similar as the calculated manually. In addition, we can observe that with the 4 processadors paraver trace (figure 14) the time is approximately the same we calculate for T_∞ which is the critical path. In figure 14 we can check that with 2 processors the code obtaining the maximum parallelism, so adding more processors will have no impact in the parallelism. For calculating the Parallelism we do T_1/T_∞.

$$Parallelism = T_1/T_\infty = 593.772.000/593.758.000 = \mathbf{1,00002358}$$

```
    START_COUNT_TIME;

    tareador_start_task("Seq code");

    ffts1_planes(p1d, in_fftw);
    transpose_xy_planes(tmp_fftw, in_fftw);
    ffts1_planes(p1d, tmp_fftw);
    transpose_zx_planes(in_fftw, tmp_fftw);
    ffts1_planes(p1d, in_fftw);
    transpose_zx_planes(tmp_fftw, in_fftw);
    transpose_xy_planes(in_fftw, tmp_fftw);

    tareador_end_task("Seq code");

    STOP_COUNT_TIME("Execution FFT3D");
```

Figure 11: Part from **3dfft_seq.c** code

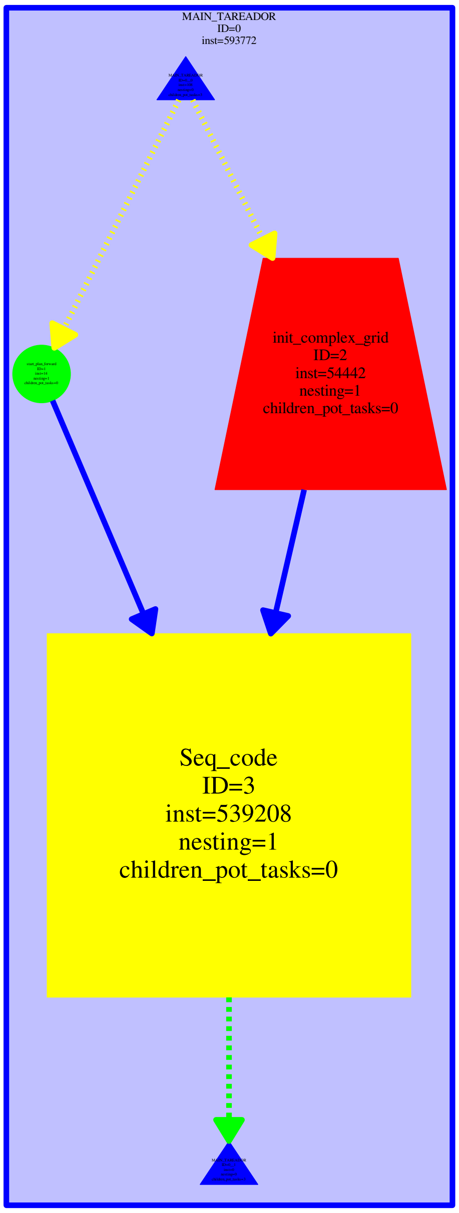


Figure 12: Dependence graph for 3dfft_seq.c

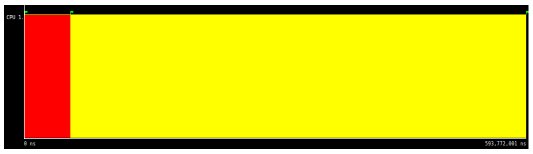


Figure 13: Traces with 1 processador

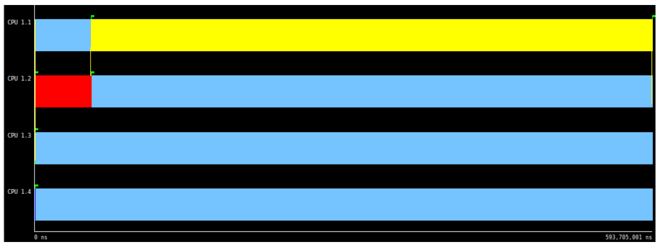


Figure 14: Traces with 4 processadors

4.2 Version 1

In the first version, we have changed the *Tareador* calls. In this case we are generating a *Tareador* task for all the calls to the different functions on the code. In other words, we are generating a task for each function that the main code calls. We can see this in the code below (figure 15)

From the manual calculation, we know the time is 1.000 time units (ns) per instruction. In the dependence graph (figure 16) we can see that there are 4 different paths. We are going to do the calculations for T_1, T_∞ and the Paralellism. For calculating T_∞ we need to calculate the number of instructions for the critical path. In the dependence graph we can see that the size of the tasks is proportional to the number of instructions they have, which can help us to know the worst case (which is the one that have the biggest nodes).

$$T_1 = totalInstructions * 1.000 = \mathbf{593.772.000}$$

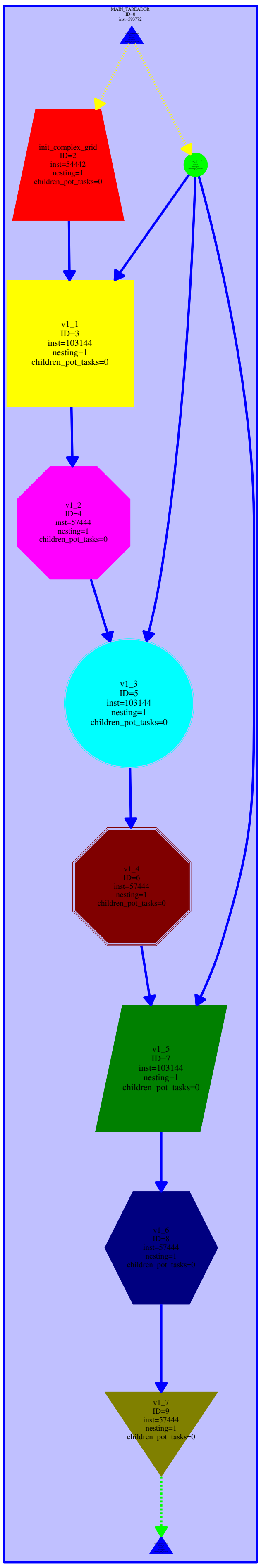
$$T_\infty = (108+54442+103144+57444+103144+57444+103144+57444+57444)*1.000 = \mathbf{593.758.000}$$

However, we can observe from the 1 processador paraver trace (figure 17), that the time for all the instructions is similar as the calculated manually. In addition, this situation happens again with the 4 processadors paraver trace (figure 18), where the time is approximately the calculated for T_∞ (which is the critical path). In figure 18 we can see that with 4 processors this version achieves the maximum parallelism. To calculate the Parallelism we can do T_1/T_∞.

$$Parallelism = T_1/T_\infty = 593.772.000/593.758.000 = \mathbf{1{,}00002357863}$$

```
tareador_start_task("v1_1");
ffts1_planes(p1d, in_fftw);
tareador_end_task("v1_1");

tareador_start_task("v1_2");
transpose_xy_planes(tmp_fftw, in_fftw);
tareador_end_task("v1_2");

tareador_start_task("v1_3");
ffts1_planes(p1d, tmp_fftw);
tareador_end_task("v1_3");

tareador_start_task("v1_4");
transpose_zx_planes(in_fftw, tmp_fftw);
tareador_end_task("v1_4");

tareador_start_task("v1_5");
ffts1_planes(p1d, in_fftw);
tareador_end_task("v1_5");

tareador_start_task("v1_6");
transpose_zx_planes(tmp_fftw, in_fftw);
tareador_end_task("v1_6");

tareador_start_task("v1_7");
transpose_xy_planes(in_fftw, tmp_fftw);
tareador_end_task("v1_7");
```

Figure 15: Part from **3dfft_seq_v1.c** code



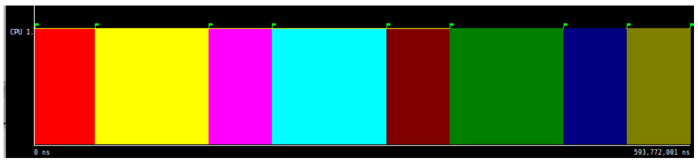


Figure 17: Traces with 1 processador

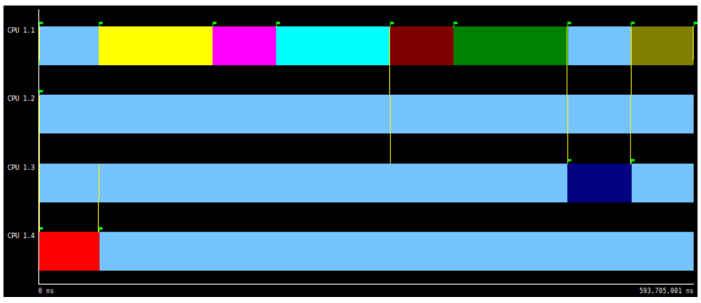


Figure 18: Traces with 4 processadors

Figure 16: Dependence graph for 3dfft_v1.c

4.3 Version 2

In the second version, after analysing the code, we saw that the call **ffts1_planes(p1d, tmp_fftw)** was called 3 times, so we make the tareador tasks inside the function. This change makes that when we execute the code, this part will be done in parallel at the same time than the other ones. We can see this is the code below (figure 20)

From the manual calculation, we know that the time is 1.000 time units (ns) per instruction. In the dependence graph (figure 19) we can see that there are many different paths, but the major part of them are similar. We have done the calculations for T_1, T_∞ and the Paralellism. For calculating T_∞ we need to calculate the number of instructions for the critical path (that is the one that repeats a many times). In the dependence graph we can see that the size of the tasks is proporcional to the number of instructions they have. This fact can help us to know the worst case which is the one that have the biggest nodes.

$$T_1 = totalInstructions * 1.000 = \mathbf{593.772.000}$$

$$T_\infty = (55 + 54442 + 10305 + 57444 + 10305 + 57444 + 10305 + 57444 + 57444) * 1.000 = \mathbf{315.188.000}$$

However, we ca see from the 1 processador paraver trace (figure 21), that the time for all the instructions is similar as the calculated manually. The same result happens with the 10 processadors paraver trace (figure 22) where the time is approximately the same as calculated for T_∞ (which is the critical path) and also we see that with 10 processors this version achieves the maximum parallelism. For calculating the Parallelism we do T_1/T_∞.

$$Parallelism = T_1/T_\infty = 593.772.000/315.188.000 = \mathbf{1{,}88386613704}$$

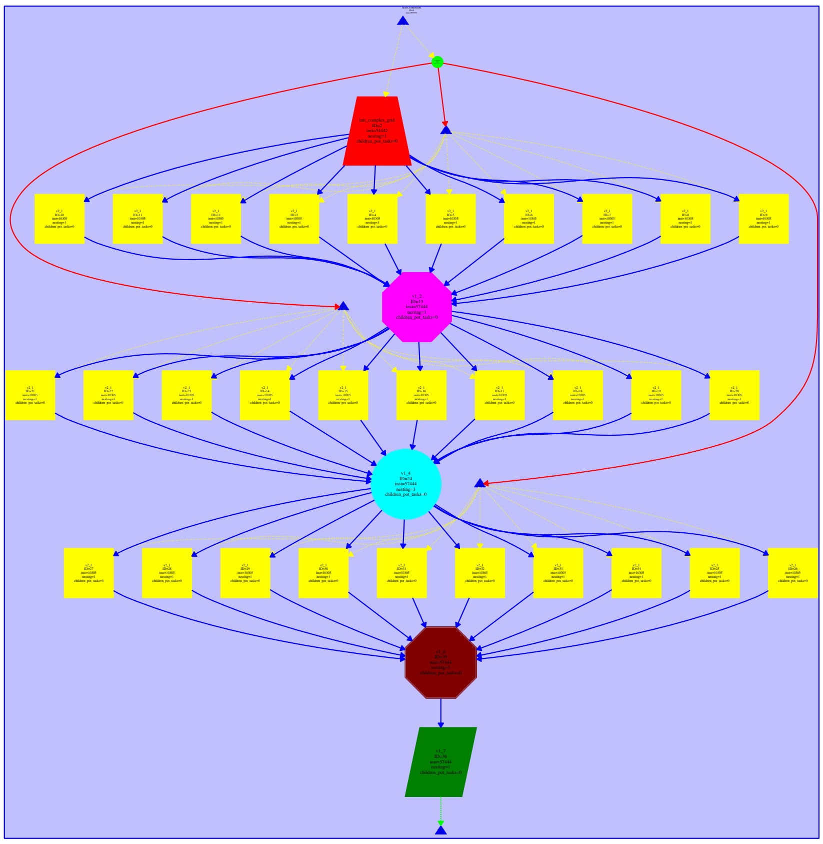


Figure 19: Dependence graph for 3dfft_v2.c

```
void ffts1_planes(fftwf_plan p1d, fftwf_complex in_fftw[][N][N]) {
    int k,j;
    for (k=0; k<N; k++)  {
        tareador_start_task("v2_1");
        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("v2_1");
    }
}
int main() {
    ...

    START_COUNT_TIME;

    ffts1_planes(p1d, in_fftw);

    tareador_start_task("v1_2");
    transpose_xy_planes(tmp_fftw, in_fftw);
    tareador_end_task("v1_2");

    ffts1_planes(p1d, tmp_fftw);

    tareador_start_task("v1_4");
    transpose_zx_planes(in_fftw, tmp_fftw);
    tareador_end_task("v1_4");

    ffts1_planes(p1d, in_fftw);

    tareador_start_task("v1_6");
    transpose_zx_planes(tmp_fftw, in_fftw);
    tareador_end_task("v1_6");

    tareador_start_task("v1_7");
    transpose_xy_planes(in_fftw, tmp_fftw);
    tareador_end_task("v1_7");

    STOP_COUNT_TIME("Execution FFT3D");
    ...
}
```

Figure 20: Part from **3dfft_seq_v2.c** code

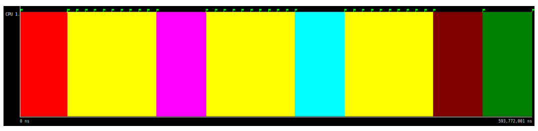


Figure 21: Traces with 1 processador

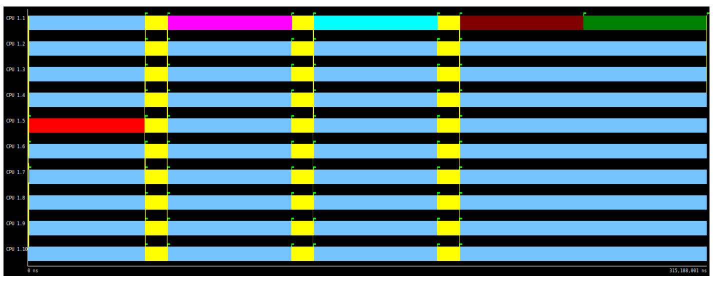


Figure 22: Traces with 10 processadors

4.4 Version 3

In the third version, we have continued like in the previous version and we have made the Tareador tasks inside the other 2 funcions: **transpose_xy_planes(..)** and **transpose_zx_planes(..)** now, the only one that does not have the tareador task inside the function is the **init_complex_grid(in_fftw);** function. We can see this is the code below (figure 24)

For the manual calculation, we know that the time is 1000 time units (ns) per instruction. In the dependence graph (figure 23) we can see that there are many different paths, but the major part are similar (like in version 2). This is because the is more parallelism in each version. We need to calculate T_1, T_∞ and the Paralellism. For calculating T_∞ we need to calculate the number of instructions for the critical path (that is the one that repeats a many times). In the dependence graph we can see that the size of the tasks is proporcional to the number of instructions they have. This helps us to know the worst case which is the one that have the biggest nodes.

$$T_1 = totalInstructions * 1.000 = \mathbf{593.772.000}$$

$$T_\infty = (55 + 54442 + 10305 + 5735 + 10305 + 5735 + 10305 + 5735 + 5735) * 1.000 = \mathbf{108.352.000}$$

However, we can see from the 1 processador paraver trace (figure 25), that the time for all the instructions is similar as the calculated manually. The same happens with the 10 processadors paraver trace (figure 26) where the time is approximately the same we calculate for T_∞ (which is the critical path) and also we see that with 10 processors this version is the maximum parallel possible. For calculating the Parallelism we do T_1/T_∞.

$$Parallelism = T_1/T_\infty = 593.772.000/108.352.000 = \mathbf{5{,}4800280567}$$

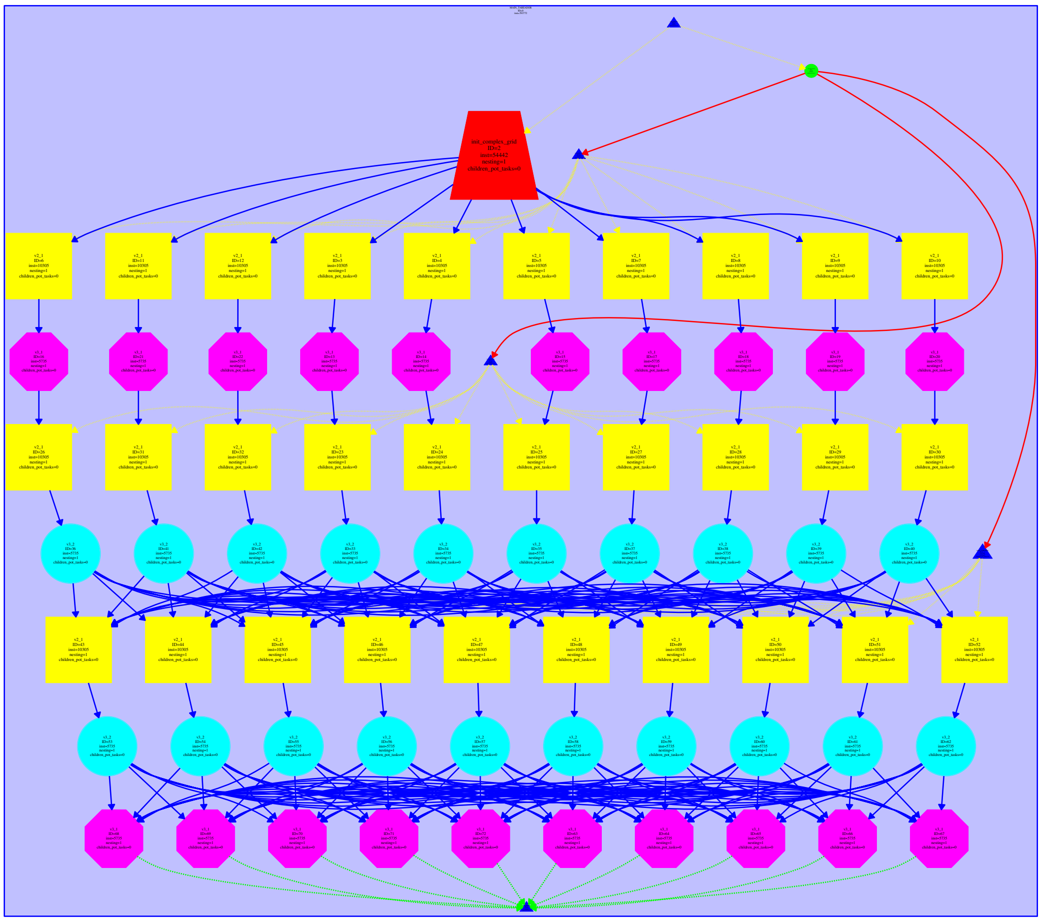


Figure 23: Dependence graph for 3dfft_v3.c

```
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("v3_1");
     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("v3_1");
    }
}

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("v3_2");
        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("v3_2");
    }
}

int main() {
    ...
    START_COUNT_TIME;

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

    START_COUNT_TIME;

    ffts1_planes(p1d, in_fftw);
    transpose_xy_planes(tmp_fftw, in_fftw);
    ffts1_planes(p1d, tmp_fftw);
    transpose_zx_planes(in_fftw, tmp_fftw);
    ffts1_planes(p1d, in_fftw);
    transpose_zx_planes(tmp_fftw, in_fftw);
    transpose_xy_planes(in_fftw, tmp_fftw);

    STOP_COUNT_TIME("Execution FFT3D");
    ...
}
```

Figure 24: Part from **3dfft_seq_v3.c** code

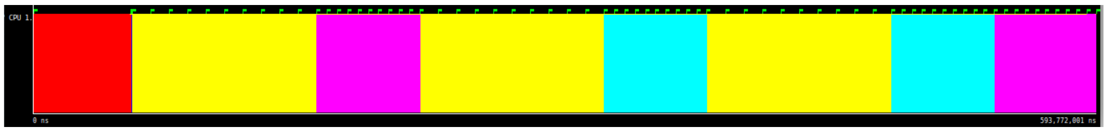


Figure 25: Traces with 1 processador

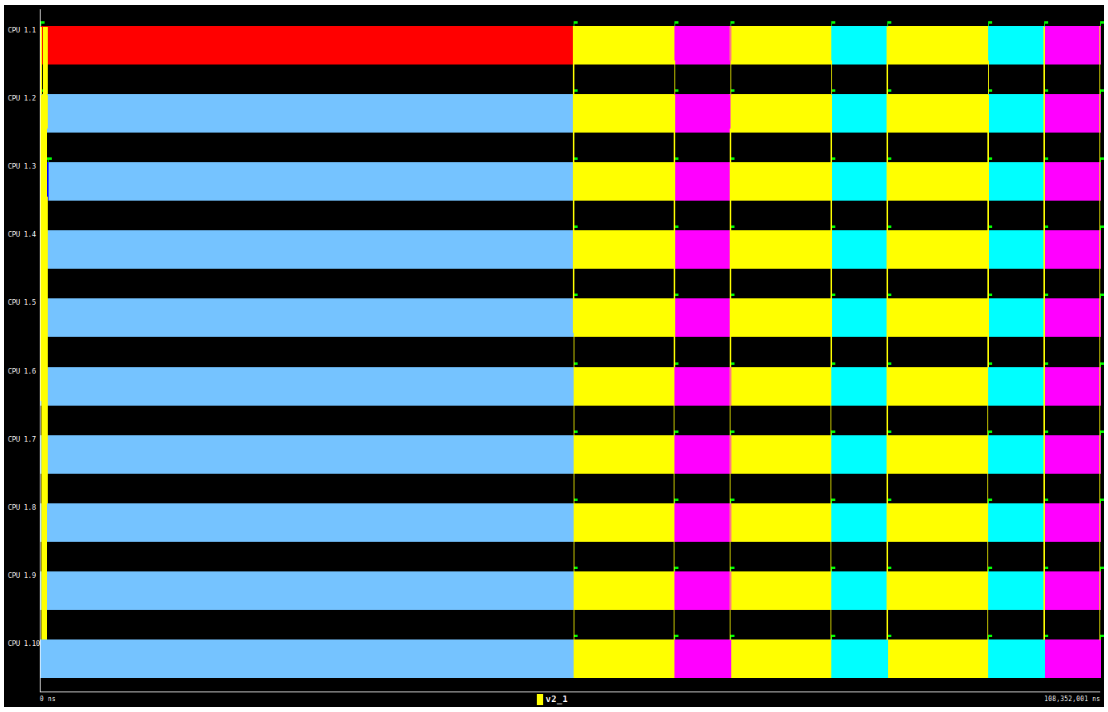


Figure 26: Traces with 10 processadors

4.5 Version 4

In the fourth version, we have continued like in the previous version and we have made the Tareador tasks inside the function **init_complex_grid(..)**. This change gives us a better time for execution. We can see this is the code bellow (figure 28)

For the manual calculation, we know the time is 1.000 time units (ns) for instruction. In the dependence graph (figure 27) we can see that there are many different paths, but the major part are similar (like in version 2 and 3). This fact is because the we are achieving more parallelism in each version. We need to calculate T_1, T_∞ and the Paralellism. For calculating T_∞ we need to calculate the number of instructions for the worst path (that is the one that repeats a many times). In the dependence graph we can see that the size of the tasks are proporcionally with the number of instructions they have. This can help us to know the worst case which is the one that have the biggest nodes.

$$T_1 = totalInstructions * 1.000 = \mathbf{593.772.000}$$

$$T_\infty = (125 + 5435 + 10305 + 5735 + 10305 + 5735 + 10305 + 5735 + 5735) * 1.000 = \mathbf{59.415.000}$$

However, we can see from the 1 processador paraver trace (figure 29), that the time for all the instructions is similar as the calculated manually. The same happens with the 16 processadors paraver trace (figure 33) that time is approximately the same we calculate for T_∞ which is the critical path. For calculating the Parallelism we do T_1/T_∞.

$$Parallelism = T_1/T_\infty = 593.772.000/59.415.000 = \mathbf{9,99363797021}$$

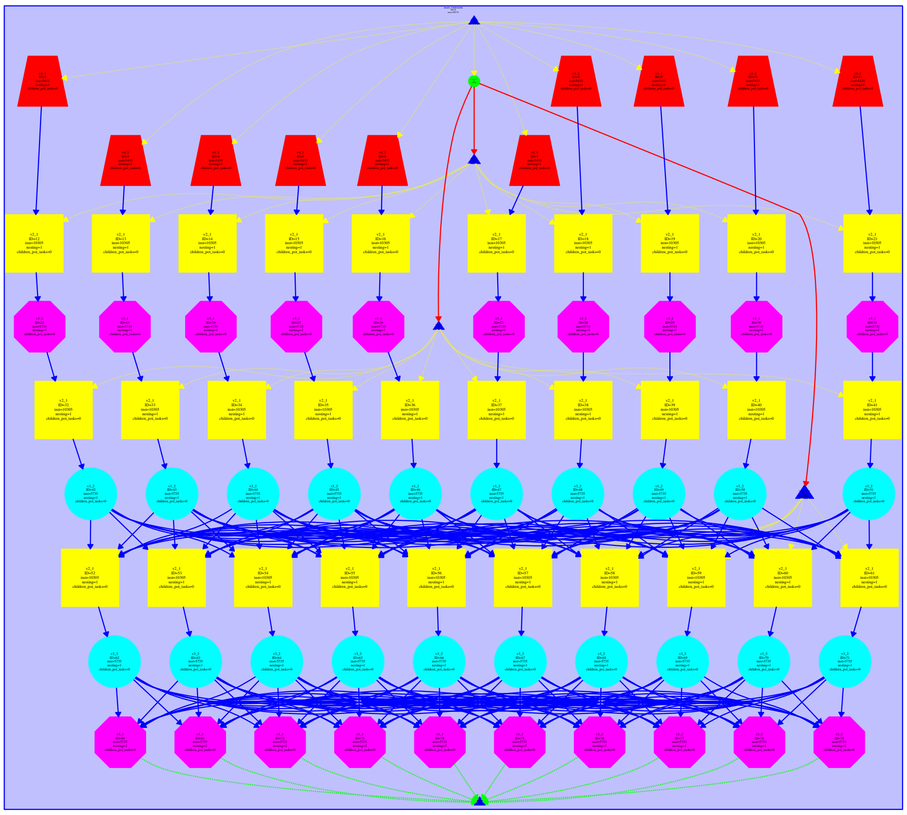


Figure 27: Dependence graph for 3dfft_v4.c

```
void init_complex_grid(fftwf_complex in_fftw[][N][N]) {
    int k,j,i;
    for (k = 0; k < N; k++) {
        tareador_start_task("v4_1");
        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("v4_1");
    }
}

int main() {
    ...
    START_COUNT_TIME;
    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");

    printf("Start Init Grid and Execute FFT3D\n");

    START_COUNT_TIME;
    init_complex_grid(in_fftw);
    STOP_COUNT_TIME("Init Complex Grid FFT3D");

    START_COUNT_TIME;

    ffts1_planes(p1d, in_fftw);
    transpose_xy_planes(tmp_fftw, in_fftw);
    ffts1_planes(p1d, tmp_fftw);
    transpose_zx_planes(in_fftw, tmp_fftw);
    ffts1_planes(p1d, in_fftw);
    transpose_zx_planes(tmp_fftw, in_fftw);
    transpose_xy_planes(in_fftw, tmp_fftw);

    STOP_COUNT_TIME("Execution FFT3D");
    ...
}
```

Figure 28: Part from **3dfft_seq_v3.c** code

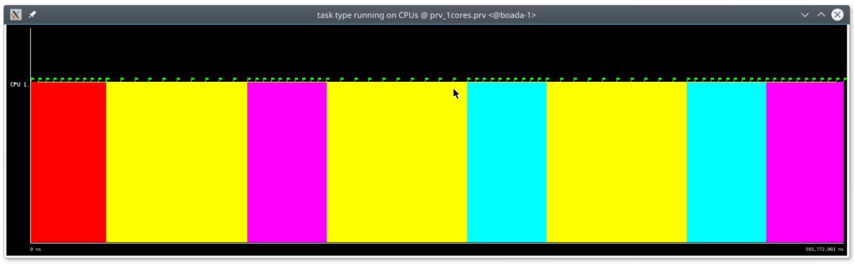


Figure 29: Traces with 1 processador

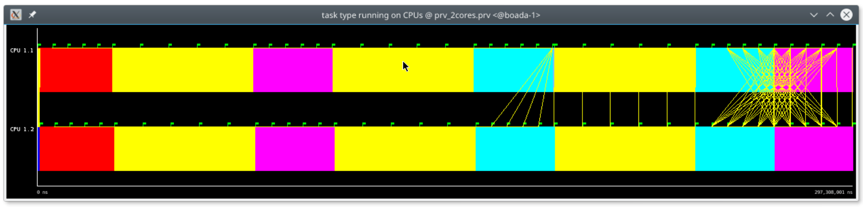


Figure 30: Traces with 2 processadors

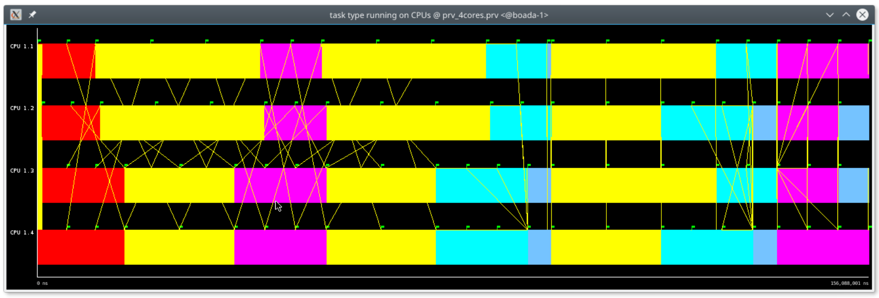


Figure 31: Traces with 4 processadors

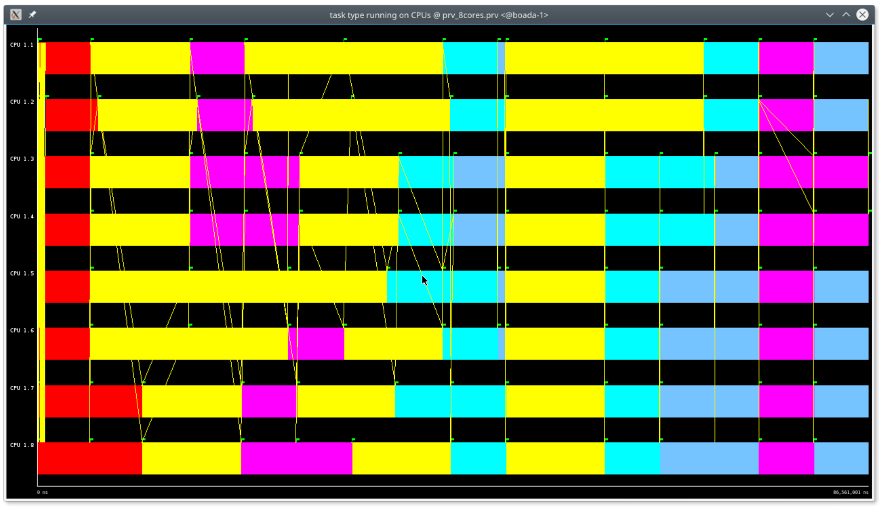


Figure 32: Traces with 8 processadors

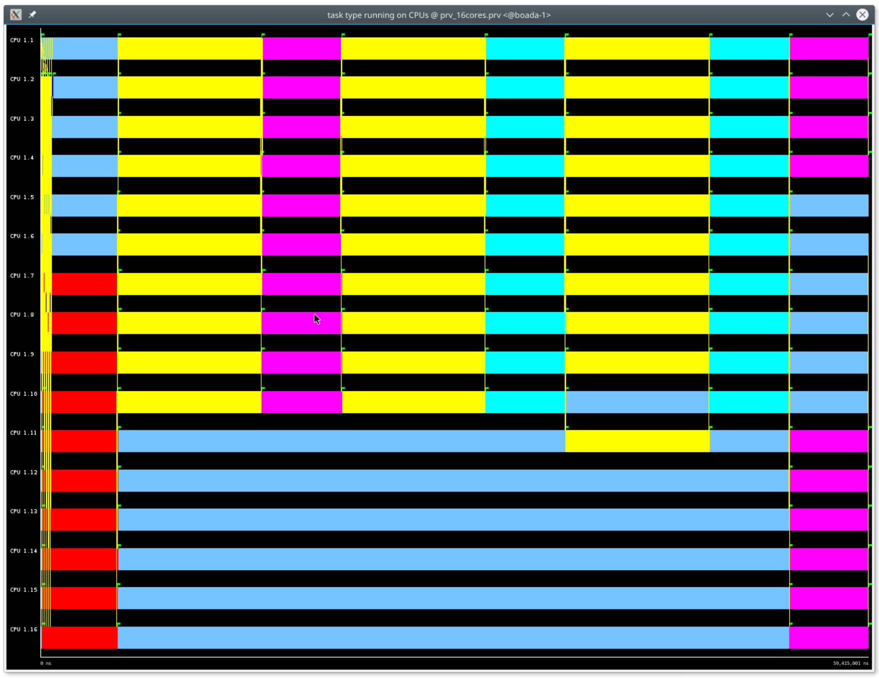


Figure 33: Traces with 16 processadors

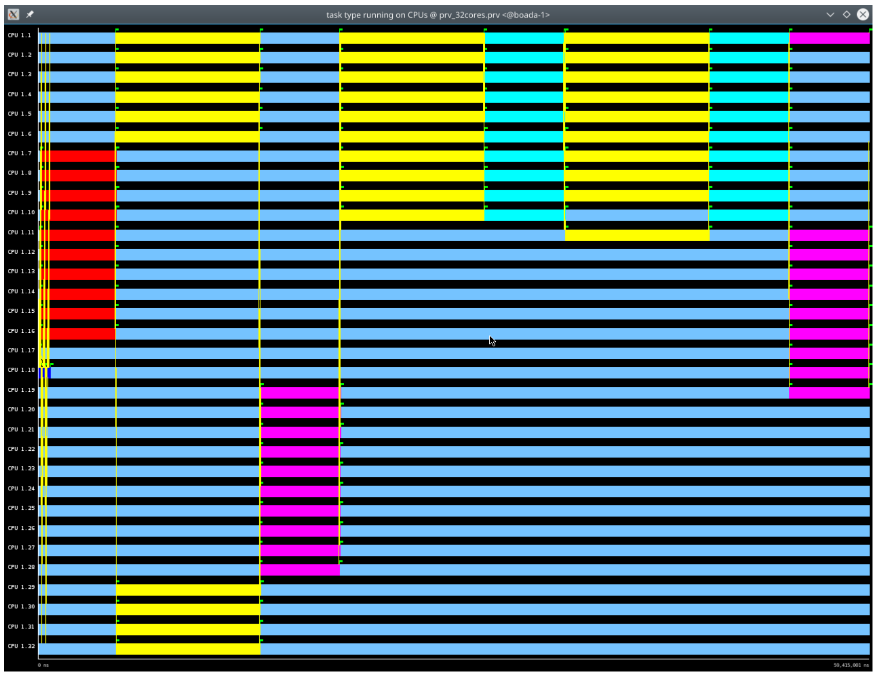
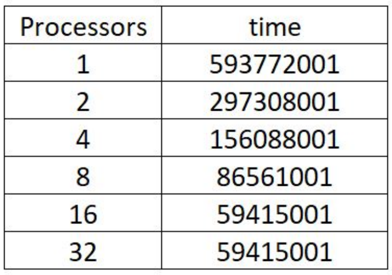


Figure 34: Traces with 32 processadors

For this version we can see that when we increase the number of processors, the time for the execution reduces to a limit where the numbers of processos doen't matter because we do not use all the processors, in this case, we see that the minimum time is reached with 16 processors, we can se this in the grafics and tables below, figure 35 for Execution Time Plot and figure 36 for Speed-Up Plot.



Processors	time
1	593772001
2	297308001
4	156088001
8	86561001
16	59415001
32	59415001

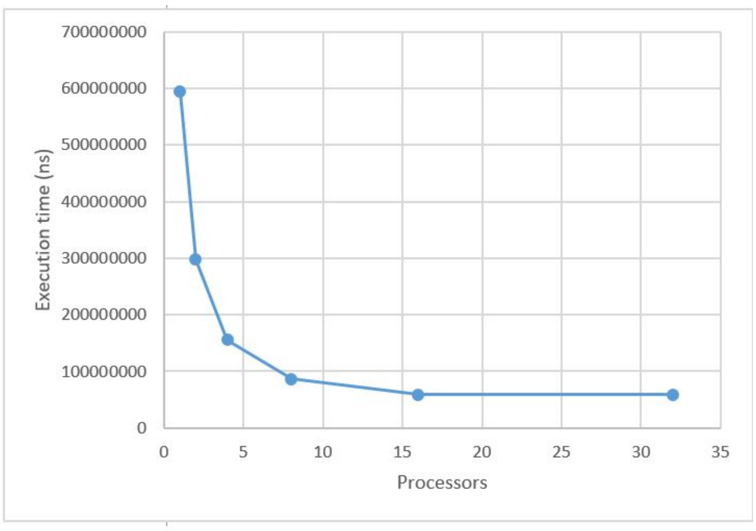
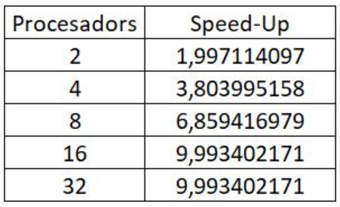


Figure 35: Execution Time Plot



Procesadors	Speed-Up
2	1,997114097
4	3,803995158
8	6,859416979
16	9,993402171
32	9,993402171

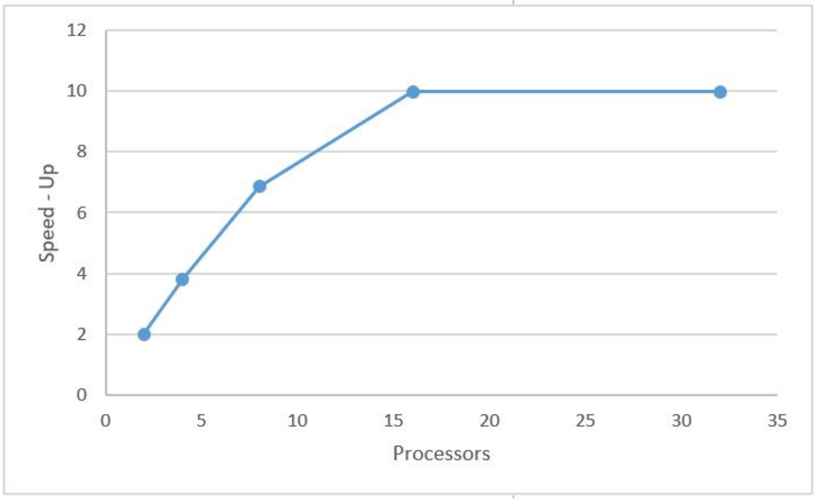


Figure 36: Speed-Up Plot

4.6 Summary Table

To conclude, we can find the table resuming all the data calculed in the previous steps, we can see that in all the versions we increse the Parallelism, in the first version the parallelism do not increse, but for the others it goes to best in all the new versions presented.

Version		**T1**	**Tinfinito**	**Parallelism**
Seq	**Manual**	593.772.000	593.758.000	1,00002358
	Tareador	593.772.001	593.758.001	1,00002358
V1	**Manual**	593.772.000	593.758.000	1,00002359
	Tareador	593.772.001	593.705.001	1,00011285
V2	**Manual**	593.772.000	315.188.000	1,88386614
	Tareador	593.772.001	315.188.001	1,88386613
V3	**Manual**	593.772.000	108.352.000	5,48002806
	Tareador	593.772.001	108.352.001	5,48002802
V4	**Manual**	593.772.000	59.415.000	9,99363797
	Tareador	593.772.001	59.415.001	9,99363782

Table 1: Analisy of dependency for the versions from **3dfft_seq.c** code

5 Tracing the execution of parallel programs

In this session we learn about tracing execution on parallel programs, we learn more about *Paraver* and the OMP_event that this have.

In this case, we worked with the code **3dfft_omp.c** in 2 ways always with 4 threads, first with out the pragma events in the original verion of the code (figure 40), and then with the pragma events in the version 2 of the code (figure 41). We can see the differences in the imatges bellow. In this 2 imatges we can see that the % time for the different versions has a important diference, we can see that the one that uses pragma is most efficient, also we can see in figures **??** and **??** where we see the time in ns.

For the original version we have 2 qüestions to answer:

How many parallel regions have been defined by the programmer in 3dff_omp.c? 1 (figure 37)
And how many work–sharing constructs inside each parallel region? 8

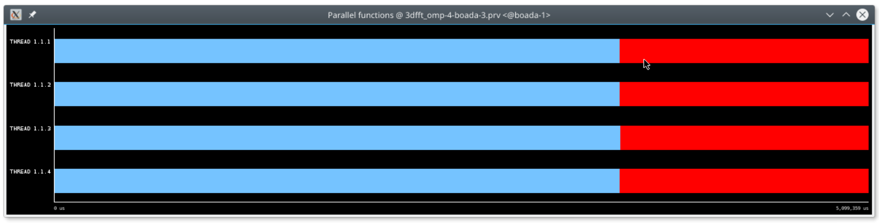
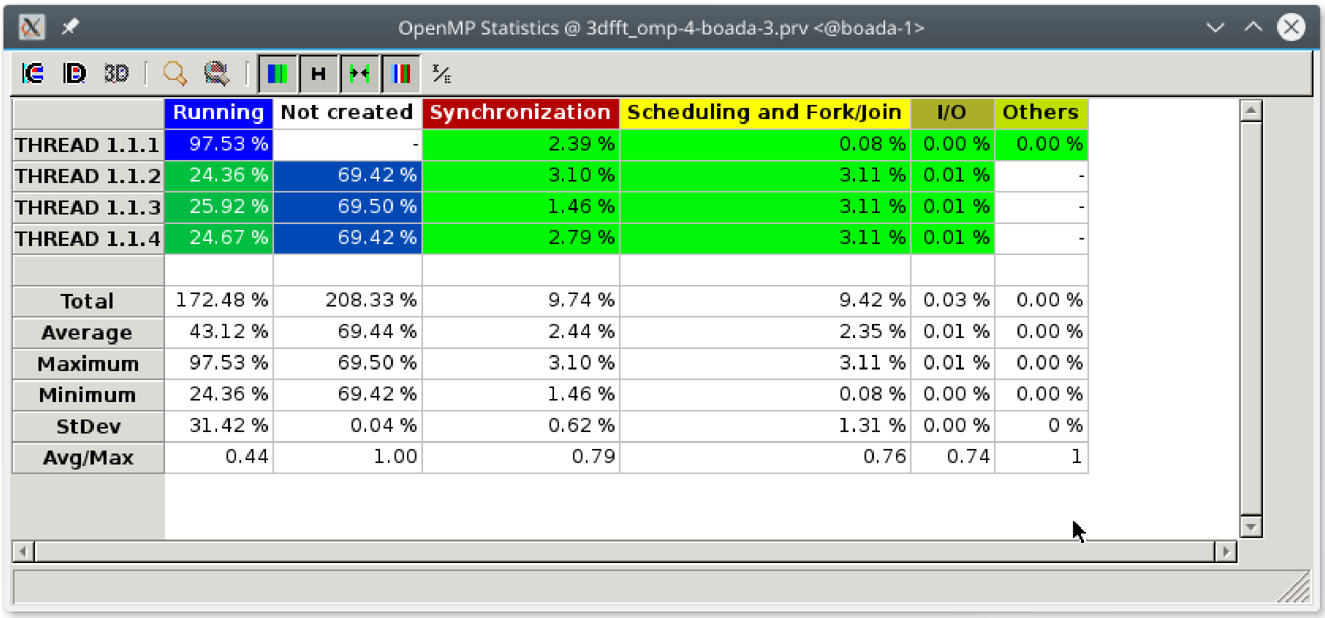
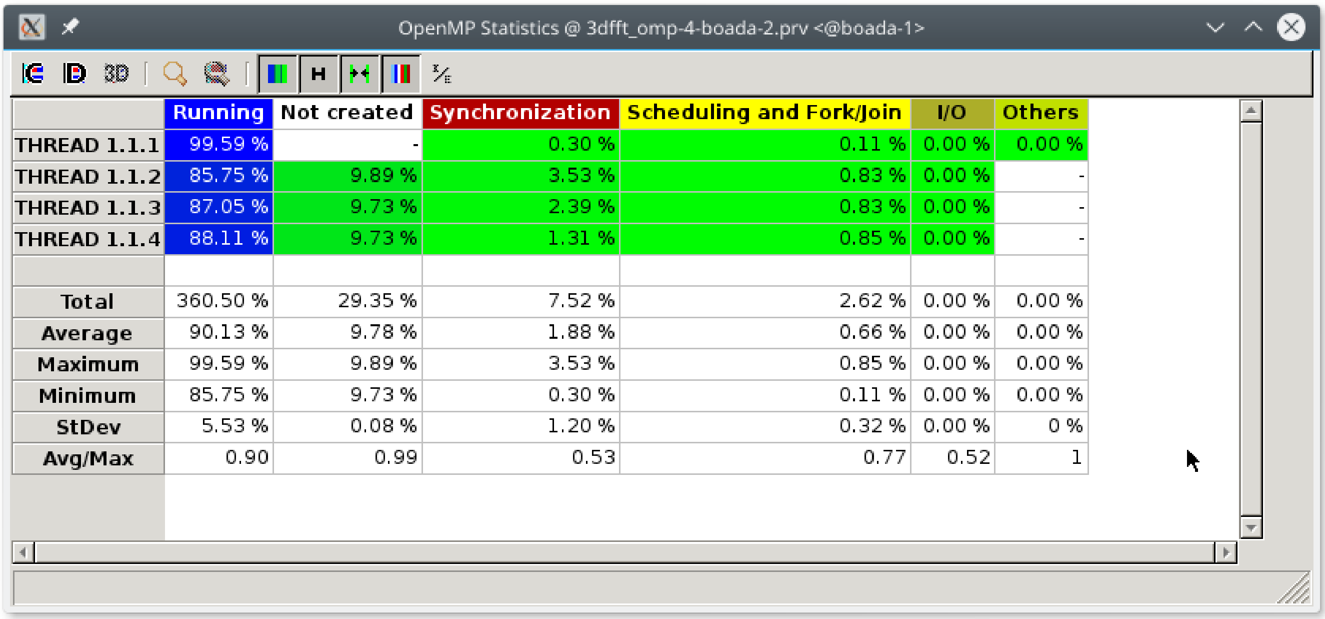


Figure 37: Parallel regions 4 Threads execution without pragma



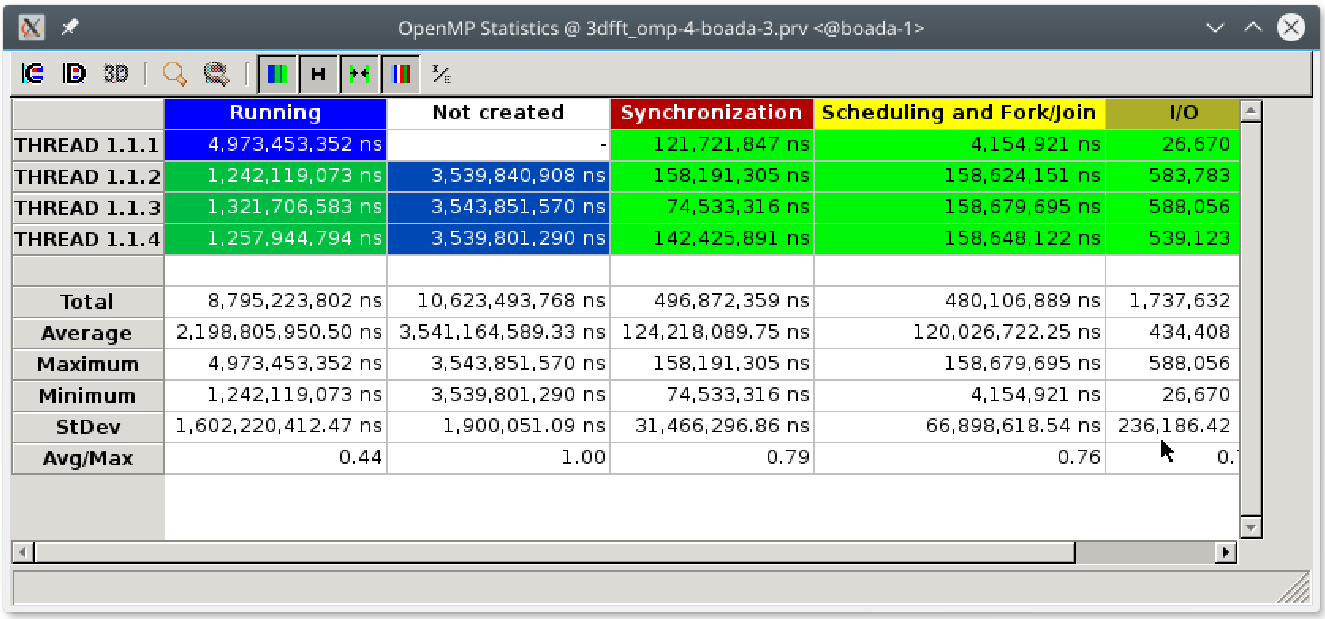
	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	97.53 %	-	2.39 %	0.08 %	0.00 %	0.00 %
THREAD 1.1.2	24.36 %	69.42 %	3.10 %	3.11 %	0.01 %	-
THREAD 1.1.3	25.92 %	69.50 %	1.46 %	3.11 %	0.01 %	-
THREAD 1.1.4	24.67 %	69.42 %	2.79 %	3.11 %	0.01 %	-
Total	172.48 %	208.33 %	9.74 %	9.42 %	0.03 %	0.00 %
Average	43.12 %	69.44 %	2.44 %	2.35 %	0.01 %	0.00 %
Maximum	97.53 %	69.50 %	3.10 %	3.11 %	0.01 %	0.00 %
Minimum	24.36 %	69.42 %	1.46 %	0.08 %	0.00 %	0.00 %
StDev	31.42 %	0.04 %	0.62 %	1.31 %	0.00 %	0 %
Avg/Max	0.44	1.00	0.79	0.76	0.74	1

Figure 38: Statics time 4 Threads execution without pragma (%)


OpenMP Statistics @ 3dfft_omp-4-boada-2.prv <@boada-1>

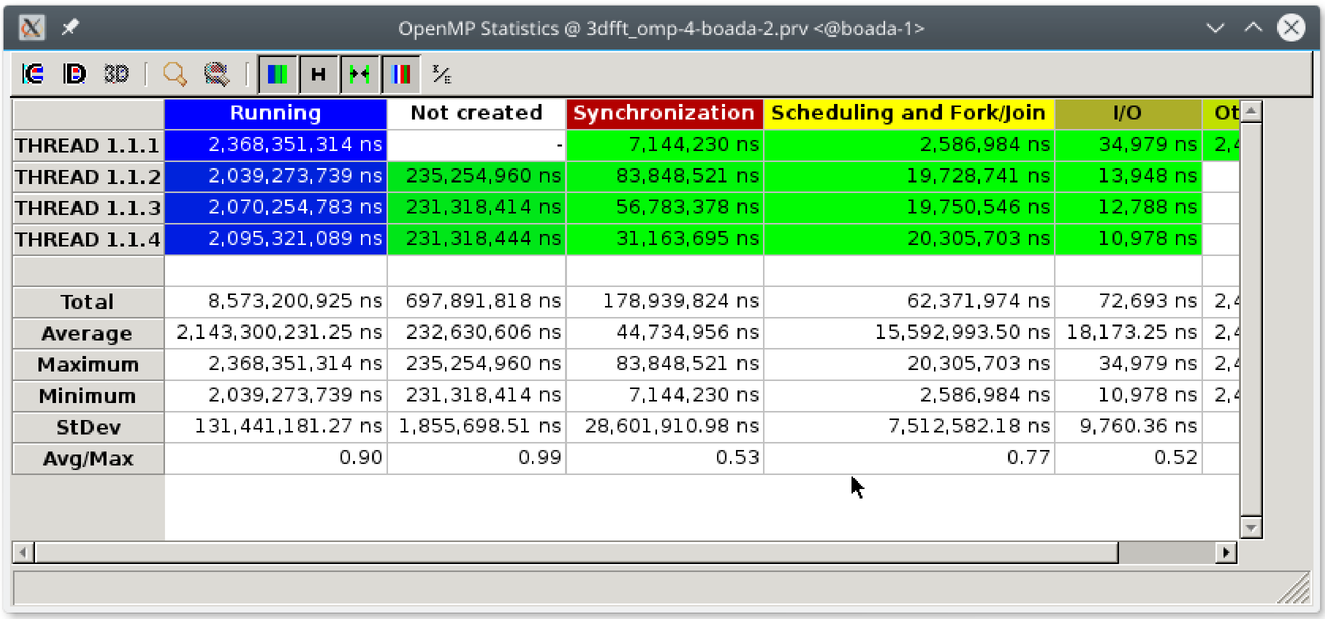
	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	99.59 %	-	0.30 %	0.11 %	0.00 %	0.00 %
THREAD 1.1.2	85.75 %	9.89 %	3.53 %	0.83 %	0.00 %	-
THREAD 1.1.3	87.05 %	9.73 %	2.39 %	0.83 %	0.00 %	-
THREAD 1.1.4	88.11 %	9.73 %	1.31 %	0.85 %	0.00 %	-
Total	360.50 %	29.35 %	7.52 %	2.62 %	0.00 %	0.00 %
Average	90.13 %	9.78 %	1.88 %	0.66 %	0.00 %	0.00 %
Maximum	99.59 %	9.89 %	3.53 %	0.85 %	0.00 %	0.00 %
Minimum	85.75 %	9.73 %	0.30 %	0.11 %	0.00 %	0.00 %
StDev	5.53 %	0.08 %	1.20 %	0.32 %	0.00 %	0 %
Avg/Max	0.90	0.99	0.53	0.77	0.52	1

Figure 39: Statics time 4 Threads execution with pragma (%)


OpenMP Statistics @ 3dfft_omp-4-boada-3.prv <@boada-1>

	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O
THREAD 1.1.1	4,973,453,352 ns	-	121,721,847 ns	4,154,921 ns	26,670
THREAD 1.1.2	1,242,119,073 ns	3,539,840,908 ns	158,191,305 ns	158,624,151 ns	583,783
THREAD 1.1.3	1,321,706,583 ns	3,543,851,570 ns	74,533,316 ns	158,679,695 ns	588,056
THREAD 1.1.4	1,257,944,794 ns	3,539,801,290 ns	142,425,891 ns	158,648,122 ns	539,123
Total	8,795,223,802 ns	10,623,493,768 ns	496,872,359 ns	480,106,889 ns	1,737,632
Average	2,198,805,950.50 ns	3,541,164,589.33 ns	124,218,089.75 ns	120,026,722.25 ns	434,408
Maximum	4,973,453,352 ns	3,543,851,570 ns	158,191,305 ns	158,679,695 ns	588,056
Minimum	1,242,119,073 ns	3,539,801,290 ns	74,533,316 ns	4,154,921 ns	26,670
StDev	1,602,220,412.47 ns	1,900,051.09 ns	31,466,296.86 ns	66,898,618.54 ns	236,186.42
Avg/Max	0.44	1.00	0.79	0.76	0.

Figure 40: Statics time 4 Threads execution without pragma (ns)



OpenMP Statistics @ 3dfft_omp-4-boada-2.prv <@boada-1>

	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Ot
THREAD 1.1.1	2,368,351,314 ns	-	7,144,230 ns	2,586,984 ns	34,979 ns	2,4
THREAD 1.1.2	2,039,273,739 ns	235,254,960 ns	83,848,521 ns	19,728,741 ns	13,948 ns	
THREAD 1.1.3	2,070,254,783 ns	231,318,414 ns	56,783,378 ns	19,750,546 ns	12,788 ns	
THREAD 1.1.4	2,095,321,089 ns	231,318,444 ns	31,163,695 ns	20,305,703 ns	10,978 ns	
Total	8,573,200,925 ns	697,891,818 ns	178,939,824 ns	62,371,974 ns	72,693 ns	2,4
Average	2,143,300,231.25 ns	232,630,606 ns	44,734,956 ns	15,592,993.50 ns	18,173.25 ns	2,4
Maximum	2,368,351,314 ns	235,254,960 ns	83,848,521 ns	20,305,703 ns	34,979 ns	2,4
Minimum	2,039,273,739 ns	231,318,414 ns	7,144,230 ns	2,586,984 ns	10,978 ns	2,4
StDev	131,441,181.27 ns	1,855,698.51 ns	28,601,910.98 ns	7,512,582.18 ns	9,760.36 ns	
Avg/Max	0.90	0.99	0.53	0.77	0.52	

Figure 41: Statics time 4 Threads execution with pragma (ns)

6 Conclusions

One of the biggest mistakes about parallelism is thinking that generating a parallel program for a sequential one is free. However, after doing the practise we have noticed just the opposite.

We would point out that the session 1 shows that talking about parallelism is synonym than talking about computer architectures. The plots obtained after executing calculus of number pi using strong and weak scaling is the prove of that rule: the same software run with the same number of threads obtain different speed-ups based in the architecture of the machine. In adittion we can observe that depending of the architecture the maximum speed-up that we can obtain by generating parallelism changes, provoked by the overhead produced by the parallelism itself (proces that have different costs depending on the architecture). In an extreme situation, this parallelism can generate negative speed-ups, due to the fact that with more threads than the needed the synchronization tasks can require more time than the program itself.

The session 2 shows that the size of the tasks in which we divide the program at generating parallelism is one of the most important decisions that an engineer have to take to improve the speed of the programs by generating parallelism. It is easy to understand that the parallel fraction of a program is pointless if we have a sequential part that consumes nearly all the execution time of the application. In this cases, trying the correct granularity of each task is vital in terms of making a good parallelism and improving the execution time.

Finally, the session 3 shows how converting a little part of the sequential fraction of the code into a parallel one can help into reducing the total execution time and increasing the parallelism in orders of magnitude.