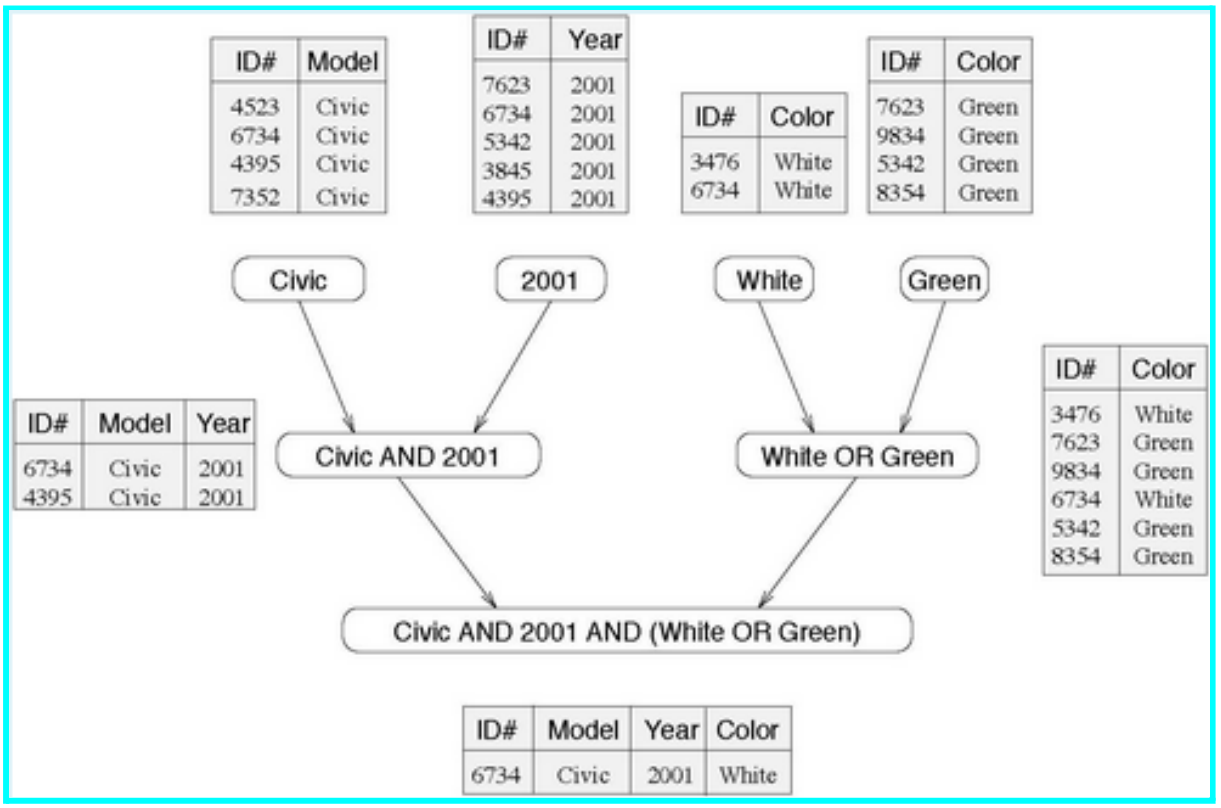
# DELIVERABLE
# LABORATORY PAR




Adrià Redondo
Miguel Gutiérrez Jariod

## Index:


1. **Experimental setup:**

**1.1** Execution modes: interactive vs queued **3**
**1.2** Node architecture and memory **4**
**1.3** Serial compilation and execution **5**
**1.4** Compilation and execution **7**
**1.5** Strong vs weak scalability **9**

2. **Systematically analysing task decomposition with Tareador**

**2.1** Tareador API **12**
**2.2** Brief Tareador hands-on **13**
**2.3** Exploring new task decompositions for 3DFFT **14**

3. **Understanding the execution of OpenMP programs**

**3.1** Short Paraver hands-on **21**
**3.1.1** Timelines: navigation and basic concepts **21**
**3.1.2** Explicit tasks **23**
**3.1.3** Profiles **23**
**3.2** Brief Tareador hands-on **24**
**3.2.1** Initial version **24**
**3.2.2** Improving **24**
**3.2.3** Reducing parallelisation overheads **25**

A point to observe during the practice is that we have created some parts called “guides for us” to show how we have developed the different tasks. These aren’t relevant for the demonstrations but for us is a support to understand what we are doing.

# 1. Experimental setup

## 1.1 Execution modes: interactive vs queued:

| | 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: | 2395 MHz |
| 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): | 12288 KB |
| Main memory size (per socket): | 12 GB |
| Main memory size (per node): | 12 GB |

Differences between the interactive and queued execution are that the first one is strongly suggested to execute when you want to ensure that the execution is done in isolation inside a single node of the machine (it starts as soon as a node is available). While in the second option the execution starts immediately but we share resources with other programs and interactive jobs, not ensuring representative timing results.

## 1.2 Node architecture and memory:

```
par2311@boada-1:~$ lscpu
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              24
On-line CPU(s) list: 0-23
Thread(s) per core:  2
Core(s) per socket:  6
Socket(s):           2
NUMA node(s):        2
Vendor ID:           GenuineIntel
CPU family:          6
Model:               44
Model name:          Intel(R) Xeon(R) CPU
Stepping:            2
CPU MHz:             1599.958
CPU max MHz:         2395.0000
CPU min MHz:         1596.0000
BogoMIPS:            4800.13
Virtualization:      VT-x
L1d cache:           32K
L1i cache:           32K
L2 cache:            256K
L3 cache:            12288K
NUMA node0 CPU(s):   0,2,4,6,8,10,12,14,16,18,20,22
NUMA node1 CPU(s):   1,3,5,7,9,11,13,15,17,19,21,23
Flags:               fpu vme de pse tsc msr pae mce
or ds_cpl vmx smx est tm2 ssse3 cx16 xtpr pdcm pcid
par2311@boada-1:~$
```

As we can see in this picture when we execute the command “lscpu” the machine shows us the different main characteristics as we had described above.

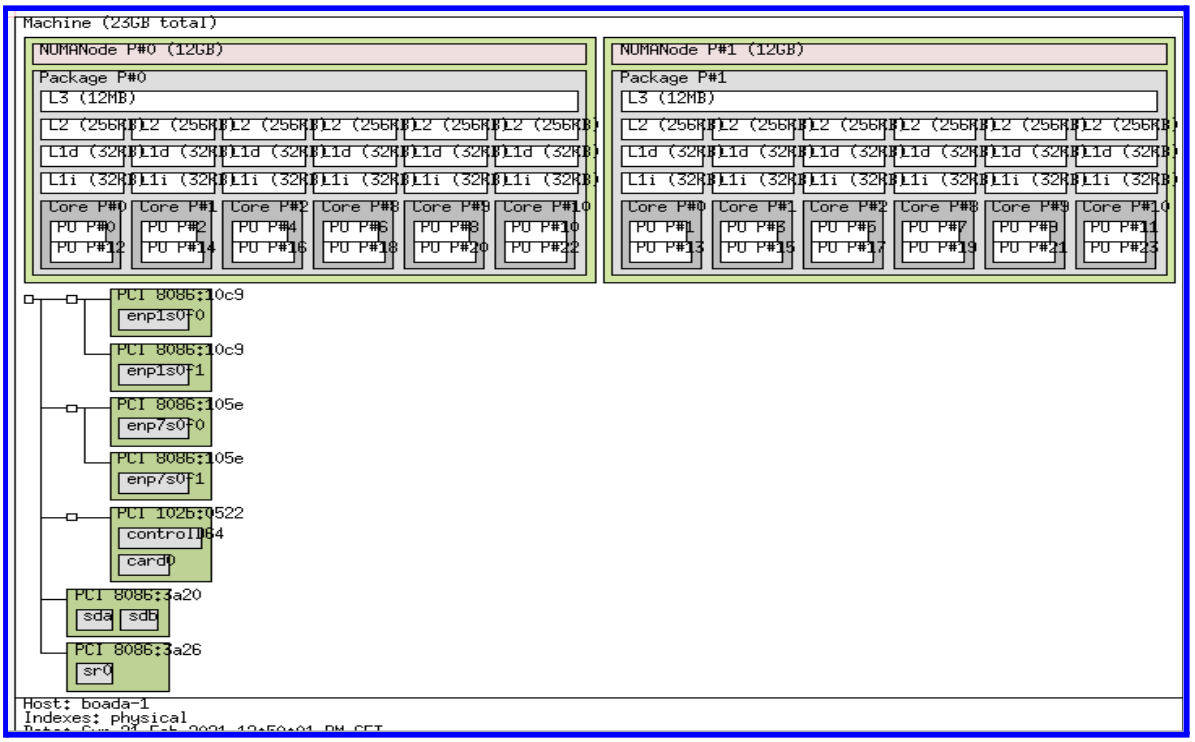
Some points to consider about the architecture are: the difference of frequency that can adopt the CPU in its maximum and minimum status, the differents cache sizes depending of the level that are situated and the huge amount of datums that can support, and finally the number of CPUs that has, which is a quite as we can see in the image.

Moreover we also share the picture where we execute the “lstopo” command to show the main memory sizes. In the image we can see the number of sockets (2), cores per socket (6) and threads per core (2) in a specific node.
Therefore we can appreciate the amount of main memory in a specific node and also for each NUMAnode.
Finally we also have constance about the caches memories (L1, L2, L3) in which demonstrate their private or shared status to each core/socked.

```
NUMA node0 CPU(s):   0,2,4,6,8,10,12,14,16,18,20,22
NUMA node1 CPU(s):   1,3,5,7,9,11,13,15,17,19,21,23
Flags:               fpu vme de pse tsc msr pae mce cx8 apic sep mtr
or ds_cpl vmx smx est tm2 ssse3 cx16 xtpr pdcm pcid dca sse4_1 sse4_
par2311@boada-1:~$ lstopo
Machine (23GB total)
  NUMANode L#0 (P#0 12GB) + Package L#0 + L3 L#0 (12MB)
    L2 L#0 (256KB) + L1d L#0 (32KB) + L1i L#0 (32KB) + Core L#0
      PU L#0 (P#0)
      PU L#1 (P#12)
    L2 L#1 (256KB) + L1d L#1 (32KB) + L1i L#1 (32KB) + Core L#1
      PU L#2 (P#2)
      PU L#3 (P#14)
    L2 L#2 (256KB) + L1d L#2 (32KB) + L1i L#2 (32KB) + Core L#2
      PU L#4 (P#4)
      PU L#5 (P#16)
    L2 L#3 (256KB) + L1d L#3 (32KB) + L1i L#3 (32KB) + Core L#3
      PU L#6 (P#6)
      PU L#7 (P#18)
    L2 L#4 (256KB) + L1d L#4 (32KB) + L1i L#4 (32KB) + Core L#4
      PU L#8 (P#8)
      PU L#9 (P#20)
    L2 L#5 (256KB) + L1d L#5 (32KB) + L1i L#5 (32KB) + Core L#5
      PU L#10 (P#10)
      PU L#11 (P#22)
  NUMANode L#1 (P#1 12GB) + Package L#1 + L3 L#1 (12MB)
    L2 L#6 (256KB) + L1d L#6 (32KB) + L1i L#6 (32KB) + Core L#6
      PU L#12 (P#1)
      PU L#13 (P#13)
    L2 L#7 (256KB) + L1d L#7 (32KB) + L1i L#7 (32KB) + Core L#7
      PU L#14 (P#3)
      PU L#15 (P#15)
    L2 L#8 (256KB) + L1d L#8 (32KB) + L1i L#8 (32KB) + Core L#8
      PU L#16 (P#5)
      PU L#17 (P#17)
    L2 L#9 (256KB) + L1d L#9 (32KB) + L1i L#9 (32KB) + Core L#9
      PU L#18 (P#7)
      PU L#19 (P#19)
    L2 L#10 (256KB) + L1d L#10 (32KB) + L1i L#10 (32KB) + Core L#10
      PU L#20 (P#9)
      PU L#21 (P#21)
    L2 L#11 (256KB) + L1d L#11 (32KB) + L1i L#11 (32KB) + Core L#11
      PU L#22 (P#11)
      PU L#23 (P#23)
```

```
  HostBridge L#0
    PCIBridge
      PCI 8086:10c9
        Net L#0 "enp1s0f0"
      PCI 8086:10c9
        Net L#1 "enp1s0f1"
    PCIBridge
      PCI 8086:105e
        Net L#2 "enp7s0f0"
      PCI 8086:105e
        Net L#3 "enp7s0f1"
    PCIBridge
      PCI 102b:0522
        GPU L#4 "controlD64"
        GPU L#5 "card0"
    PCI 8086:3a20
      Block(Disk) L#6 "sda"
      Block(Disk) L#7 "sdb"
    PCI 8086:3a26
      Block(Removable Media Device) L#8 "sr0"
par2311@boada-1:~$
```

Picture of “lstopo --of fig map.fig”:




## 1.3 Serial compilation and execution:

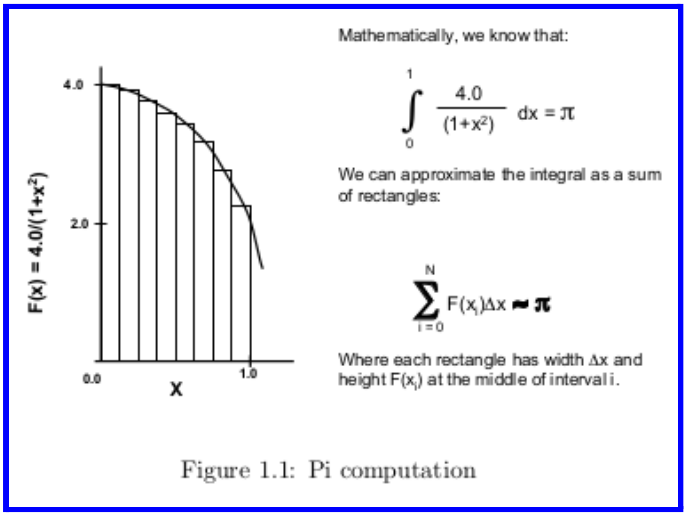



Figure 1.1: Pi computation

```
static long num_steps = 100000;
void main () {
    double x, pi, step, sum = 0.0;

    step = 1.0/(double) num_steps;
    for (long int i=0; i<num_steps; ++i) {
            x = (i+0.5)*step;
            sum += 4.0/(1.0+x*x);
    }
    pi = step * sum;
}
```

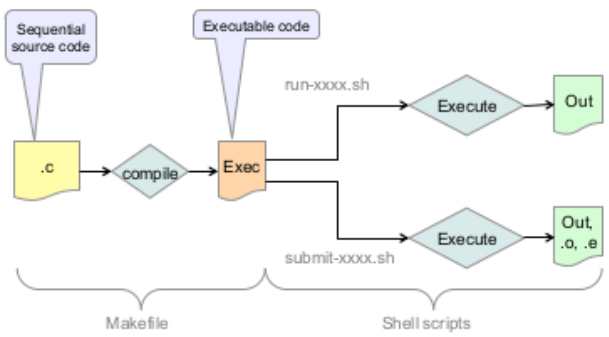



Figure 1.3: Compilation and execution flow for sequential program.

In this part of the practice we will see how to compile and execute a sequential program:

**Part 1:**

```
par2311@boada-1:~/lab1/pi$ cat Makefile
CC = icc # gcc
OPENMP    = -fopenmp
CFLAGS    = -g -O3 -std=c99

TARGETS  = pi_seq pi_omp
all: $(TARGETS)

pi_seq: pi_seq.c
        $(CC) $(CFLAGS) $< -o $@

pi_omp: pi_omp.c
        $(CC) $(CFLAGS) $(OPENMP) $< -o $@

clean:
        rm -rf $(TARGETS)

ultraclean:
        rm -rf TRACE*.mpit $(TARGETS) *.prv *.pcf *.row s
par2311@boada-1:~/lab1/pi$ ./run-seq.sh pi_seq 1000000000
```

**Part 2:**

```
Number pi after 1000000000 iterations = 3.141592653589768
Execution time (secs.): 3.944225
3.94user 0.00system 0:03.94elapsed 99%CPU (0avgtext+0avgdata 2104maxresident)k
0inputs+0outputs (0major+83minor)pagefaults 0swaps
par2311@boada-1:~/lab1/pi$ sbatch ./submit-seq.sh pi_seq 1000000000
Submitted batch job 62640
par2311@boada-1:~/lab1/pi$ ls -ltr
total 136
-rwxr-xr-x 1 par2311 par2311  3068 Feb 12 16:54 submit-weak-omp.sh
-rwxr-xr-x 1 par2311 par2311  3072 Feb 12 16:54 submit-strong-omp.sh
-rwxr-xr-x 1 par2311 par2311   517 Feb 12 16:54 submit-seq.sh
-rwxr-xr-x 1 par2311 par2311   582 Feb 12 16:54 submit-omp.sh
-rwxr-xr-x 1 par2311 par2311   729 Feb 12 16:54 submit-extrae.sh
-rwxr-xr-x 1 par2311 par2311   245 Feb 12 16:54 run-seq.sh
-rwxr-xr-x 1 par2311 par2311   293 Feb 12 16:54 run-omp.sh
-rw-r--r-- 1 par2311 par2311  1223 Feb 12 16:54 pi_seq.c
-rw-r--r-- 1 par2311 par2311  1515 Feb 12 16:54 pi_omp.c
-rw-r--r-- 1 par2311 par2311   352 Feb 12 16:54 Makefile
-rw-r--r-- 1 par2311 par2311   979 Feb 12 16:54 extrae.xml
-rwxrwxr-x 1 par2311 par2311 35360 Feb 19 13:21 pi_seq
-rwxrwxr-x 1 par2311 par2311 37552 Feb 19 13:21 pi_omp
-rw-rw-r-- 1 par2311 par2311     0 Feb 19 13:38 submit-seq.sh.e62238
-rw-rw-r-- 1 par2311 par2311    94 Feb 19 13:38 submit-seq.sh.o62238
-rw-rw-r-- 1 par2311 par2311     0 Feb 21 13:06 submit-seq.sh.e62590
-rw-rw-r-- 1 par2311 par2311   121 Feb 21 13:06 submit-seq.sh.o62590
-rw-rw-r-- 1 par2311 par2311     0 Feb 21 18:20 submit-seq.sh.e62640
-rw-rw-r-- 1 par2311 par2311   121 Feb 21 18:20 submit-seq.sh.o62640
-rw-rw-r-- 1 par2311 par2311   130 Feb 21 18:20 time-pi_seq-boada-2
```

**Part 3:**

```
par2311@boada-1:~/lab1/pi$ squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
             62608 interacti _interac  par4120  R    2:08:50      1 boada-1
             62638 interacti _interac  par2311  R       3:41      1 boada-1
             62632 interacti _interac  par1215  R      20:50      1 boada-1
             62631 interacti _interac  par4211  R      20:57      1 boada-1
             62615 interacti _interac  par4217  R    1:02:07      1 boada-1
             62612 interacti _interac  par2107  R    1:40:33      1 boada-1
             62610 interacti _interac  pap0015  R    2:00:57      1 boada-1
par2311@boada-1:~/lab1/pi$ cat time-pi_seq-boada-2
3.94user 0.00system 0:03.95elapsed 99%CPU (0avgtext+0avgdata 2140maxresident)k
0inputs+8outputs (0major+85minor)pagefaults 0swaps
par2311@boada-1:~/lab1/pi$ cat submit-seq.sh.e62640
par2311@boada-1:~/lab1/pi$ cat submit-seq.sh.o62640
make: 'pi_seq' is up to date.
Number pi after 1000000000 iterations = 3.141592653589768
Execution time (secs.): 3.944833
par2311@boada-1:~/lab1/pi$
```

In the first part we have seen the Makefile and we have generated the targets that show the picture. Later in the second part we have executed the run-seq.sh with the main arguments and we obtain the results of the image. Finally in the third part we have executed the squeue command to see how the job works and at least we have seen the submit-seq.sh program with the corresponding arguments to obtain an answer respect the execution.

## 1.4 Execution and compilation of OpenMP programs:

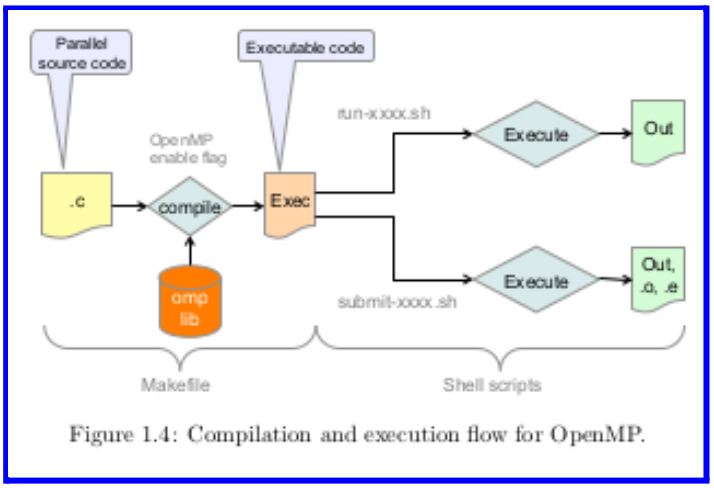



Figure 1.4: Compilation and execution flow for OpenMP.

First of all we have compare both programs to see their differences: (left: pi_seq.c, right: pi_omp.c)

```
        return ((double)time.tv_sec * (double)1e6 + (double)time.tv_use
}

#define START_COUNT_TIME stamp = getusec_();
#define STOP_COUNT_TIME(_m) stamp = getusec_() - stamp;\
                        stamp = stamp/1e6;\
                        printf ("%s%0.6f\n",(_m), stamp);

int main(int argc, char *argv[]) {
    double stamp;
    START_COUNT_TIME;

    double x, sum=0.0, pi=0.0;
    double step;

    const char Usage[] = "Usage: pi <num_steps> (try 1000000000)\n";
    if (argc < 2) {
        fprintf(stderr, Usage);
        exit(1);
    }
    long int num_steps = atoi(argv[1]);
    step = 1.0/(double) num_steps;

    /* do computation */
    for (long int i=0; i<num_steps; ++i) {
            x = (i+0.5)*step;
            sum += 4.0/(1.0+x*x);
    }
    pi = step * sum;

    /* print results */
    printf("Number pi after %ld iterations = %.15f\n", num_steps, pi);

    STOP_COUNT_TIME("Execution time (secs.): ");

    return EXIT_SUCCESS;
}
par2311@boada-1:~/lab1/pi$
```

```
                        printf ("%s%0.6f\n",(_m), stamp);

int main(int argc, char *argv[]) {
    double stamp;
    START_COUNT_TIME;

    double x, sum=0.0, pi=0.0;
    double step;

    const char Usage[] = "Usage: pi <num_steps> (try 1000000000)\n";
    if (argc < 2) {
        fprintf(stderr, Usage);
        exit(1);
    }
    long int num_steps = atoi(argv[1]);
    step = 1.0/(double) num_steps;

    /* do computation -- using all available threads */
    //omp_set_num_threads(8);
    #pragma omp parallel private(x) reduction(+: sum) // num_threads(8)
    {
        long int myid = omp_get_thread_num();
        long int howmany = omp_get_num_threads();
        for (long int i=myid; i<num_steps; i+=howmany) {
            x = (i+0.5)*step;
            sum += 4.0/(1.0+x*x);
        }
    }
    pi = step * sum;

    /* print results */
    printf("Number pi after %ld iterations = %.15f\n", num_steps, pi);

    STOP_COUNT_TIME("Execution time (secs.): ");

    return EXIT_SUCCESS;
}
par2311@boada-1:~/lab1/pi$
```

The main difference is the use of the threads in the right program with the pragma omp parallel privative.

Once we have created the executable, we have compiled the “run-omp.sh” program with the differents values of the threads and we have obtained these results:

```
par2311@boada-1:~/lab1/pi$ ./run-omp.sh pi_omp 1000000000 1
Number pi after 1000000000 iterations = 3.141592653589768
Execution time (secs.): 3.942477
3.94user 0.00system 0:03.94elapsed 99%CPU (0avgtext+0avgdata 3152maxresident)k
0inputs+0outputs (0major+135minor)pagefaults 0swaps
par2311@boada-1:~/lab1/pi$ ./run-omp.sh pi_omp 1000000000 8
Number pi after 1000000000 iterations = 3.141592653589845
Execution time (secs.): 4.052017
8.01user 0.09system 0:04.05elapsed 199%CPU (0avgtext+0avgdata 3312maxresident)k
0inputs+0outputs (0major+264minor)pagefaults 0swaps
par2311@boada-1:~/lab1/pi$
```

We can appreciate a few differences in the execution time, the user, the system, the elapsed time, the CPU and the minor.

```
par2311@boada-1:~/lab1/pi$ sbatch ./submit-omp.sh pi_omp 1000000000 1
Submitted batch job 62652
par2311@boada-1:~/lab1/pi$ sbatch ./submit-omp.sh pi_omp 1000000000 8
Submitted batch job 62653
par2311@boada-1:~/lab1/pi$ ls
extrae.xml  pi_seq      submit-extrae.sh      submit-omp.sh.o62652  submit-seq.sh
Makefile    pi_seq.c    submit-omp.sh         submit-omp.sh.o62653  submit-seq.sh
pi_omp      run-omp.sh  submit-omp.sh.e62652  submit-seq.sh         submit-seq.sh
pi_omp.c    run-seq.sh  submit-omp.sh.e62653  submit-seq.sh.e62238  submit-seq.sh
par2311@boada-1:~/lab1/pi$ cat time-pi_omp-1-boada-2
3.94user 0.00system 0:03.98elapsed 99%CPU (0avgtext+0avgdata 3128maxresident)k
80inputs+8outputs (1major+136minor)pagefaults 0swaps
par2311@boada-1:~/lab1/pi$ cat time-pi_omp-8-boada-2
5.36user 0.02system 0:00.68elapsed 782%CPU (0avgtext+0avgdata 3328maxresident)k
0inputs+8outputs (0major+182minor)pagefaults 0swaps
par2311@boada-1:~/lab1/pi$ cat submit-omp.sh.o62652
make: 'pi_omp' is up to date.
Number pi after 1000000000 iterations = 3.141592653589768
Execution time (secs.): 3.949041
par2311@boada-1:~/lab1/pi$ cat submit-omp.sh.o62653
make: 'pi_omp' is up to date.
Number pi after 1000000000 iterations = 3.141592653589845
Execution time (secs.): 0.676715
par2311@boada-1:~/lab1/pi$
```

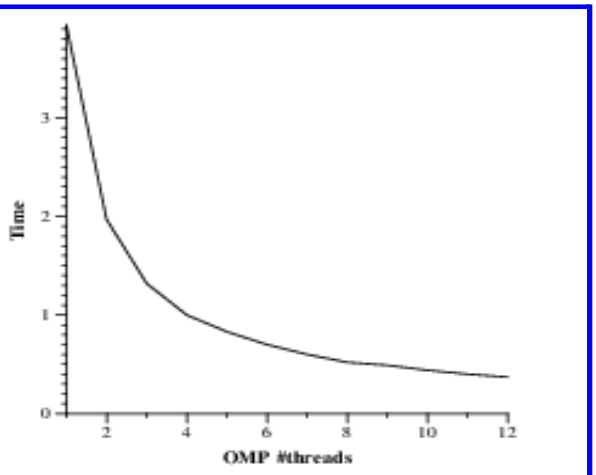
Finally we can consider that there is a difference between the iterative and queued execution.

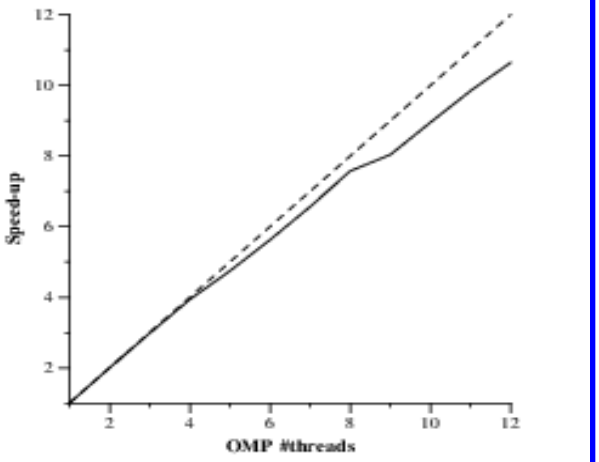
## 1.5 Strong vs weak scalability:

- In *strong* scalability the number of threads is changed with a fixed problem size. In this case parallelism is used to reduce the execution time of the program.
- In *weak* scalability the problem size is proportional to the number of threads. In this case parallelism is used to increase the problem size for which the program is executed.

In essence, both weak and strong scalability try to improve parallelism altering one of its main two affecting variables (number of threads and problem size). Strong scalability alters the number of threads keeping a constant problem size whereas weak scalability varies the problem size with a constant number of threads.

Program: “submit-strong-omp.sh” (with nthreads = 12)

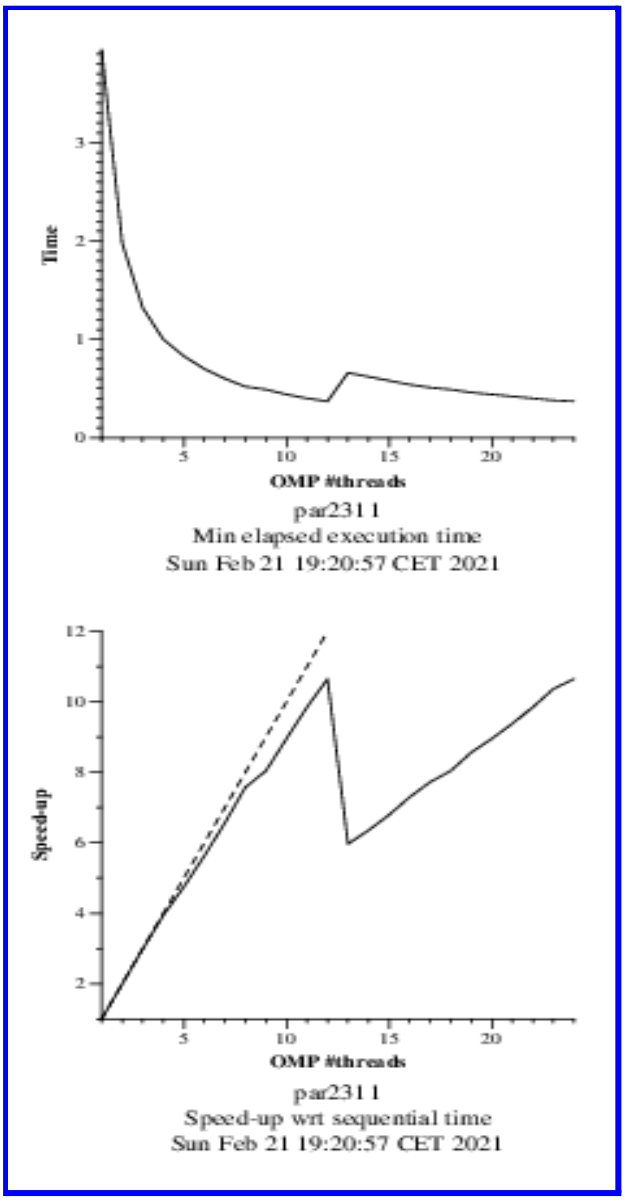



```
Run 0...Elapsed time = 1.98
Run 1...Elapsed time = 1.97
Run 2...Elapsed time = 1.98
ELAPSED TIME MIN OF 3 EXECUTIONS =1.97

Executing pi_omp with 3 threads
Run 0...Elapsed time = 1.33
Run 1...Elapsed time = 1.32
Run 2...Elapsed time = 1.33
ELAPSED TIME MIN OF 3 EXECUTIONS =1.32

Executing pi_omp with 4 threads
Run 0...Elapsed time = 1.00
Run 1...Elapsed time = 1.00
Run 2...Elapsed time = 1.00
ELAPSED TIME MIN OF 3 EXECUTIONS =1.00

Executing pi_omp with 5 threads
Run 0...Elapsed time = 0.83
Run 1...Elapsed time = 0.84
Run 2...Elapsed time = 0.83
ELAPSED TIME MIN OF 3 EXECUTIONS =0.83

Executing pi_omp with 6 threads
Run 0...Elapsed time = 0.70
Run 1...Elapsed time = 0.70
Run 2...Elapsed time = 0.70
ELAPSED TIME MIN OF 3 EXECUTIONS =0.70

Executing pi_omp with 7 threads
Run 0...Elapsed time = 0.60
Run 1...Elapsed time = 0.60
Run 2...Elapsed time = 0.60
ELAPSED TIME MIN OF 3 EXECUTIONS =0.60

Executing pi_omp with 8 threads
Run 0...Elapsed time = 0.53
par2311@boada-1:~/lab1/pi$
```

We can see how the number of threads (up to 12 threads) influence execution time and speedup respectively. Being Texe and speedup inversely proportional, an increment in #threads decreases Texe and increases speedup.

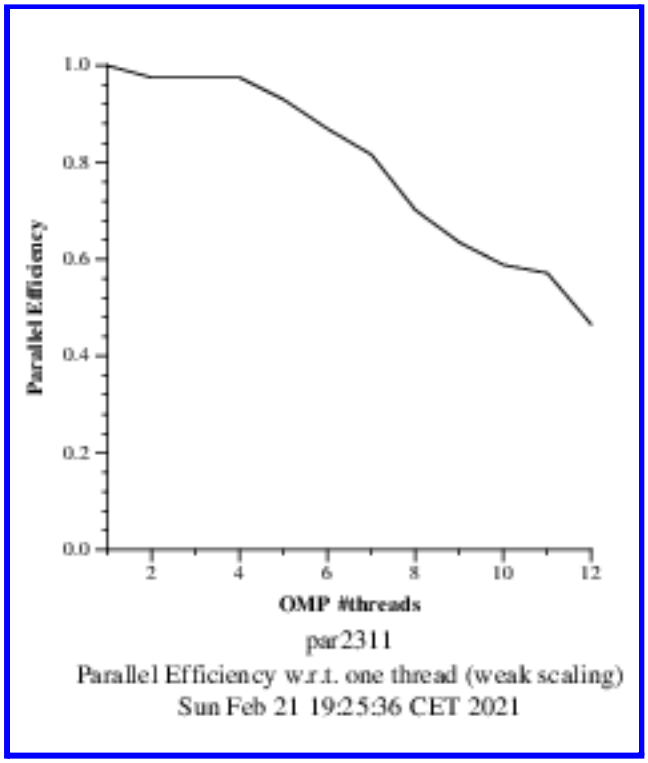
Program: “submit-strong-omp.sh” (with nthreads = 24)




```
Run 1...Elapsed time = 0.60
Run 2...Elapsed time = 0.60
ELAPSED TIME MIN OF 3 EXECUTIONS =0.60

Executing pi_omp with 8 threads
Run 0...Elapsed time = 0.52
Run 1...Elapsed time = 0.53
Run 2...Elapsed time = 0.53
ELAPSED TIME MIN OF 3 EXECUTIONS =0.52

Executing pi_omp with 9 threads
Run 0...Elapsed time = 0.49
Run 1...Elapsed time = 0.49
Run 2...Elapsed time = 0.49
ELAPSED TIME MIN OF 3 EXECUTIONS =0.49

Executing pi_omp with 10 threads
Run 0...Elapsed time = 0.44
Run 1...Elapsed time = 0.44
Run 2...Elapsed time = 0.44
ELAPSED TIME MIN OF 3 EXECUTIONS =0.44

Executing pi_omp with 11 threads
Run 0...Elapsed time = 0.40
Run 1...Elapsed time = 0.40
Run 2...Elapsed time = 0.40
ELAPSED TIME MIN OF 3 EXECUTIONS =0.40

Executing pi_omp with 12 threads
Run 0...Elapsed time = 0.37
Run 1...Elapsed time = 0.37
Run 2...Elapsed time = 0.37
ELAPSED TIME MIN OF 3 EXECUTIONS =0.37

Executing pi_omp with 13 threads
Run 0...Elapsed time = 0.66
Run 1...Elapsed time = 0.66
Run 2...par2311@boada-1:~/lab1/pi$
```

In this case, we can appreciate the same trend. Nevertheless, there is an obvious disruption with 14 threads working. It seems there exists a speedup limit around 10.5 and adding extra threads only increases the parallelization time (overheads and thread synchronization time) not paying off.

Program: “submit-weak-omp.sh”




```
ELAPSED TIME Min OF 3 EXECUTIONS =0.70

Executing pi_omp with 12 threads
Run 0 with size 1200000000 ...Elapsed time = 0.86
Run 1 with size 1200000000 ...Elapsed time = 0.87
Run 2 with size 1200000000 ...Elapsed time = 0.87
ELAPSED TIME Min OF 3 EXECUTIONS =0.86

Resultat de l'experiment (tambe es troben a  ./exe
#threads        Elapsed min
1   0.40
2   0.41
3   0.41
4   0.41
5   0.43
6   0.46
7   0.49
8   0.57
9   0.63
10   0.68
11   0.70
12   0.86

#threads        Speedup
1   1.00000000000000000000
2   .97560975609756097560
3   .97560975609756097560
4   .97560975609756097560
5   .93023255813953488372
6   .86956521739130434782
7   .81632653061224489795
8   .70175438596491228070
9   .63492063492063492063
10   .58823529411764705882
11   .57142857142857142857
12   .46511627906976744186

par2311@boada-1:~/lab1/pi$
```

In this graph we can see that efficiency decreases as the number of threads increases. This is due to overheads and thread synchronization time increasing with the number of threads.

| #threads | Interactive | | | | Queued | | | |
|---|---|---|---|---|---|---|---|---|
| | user | system | elapsed | % of CPU | user | system | elapsed | % of CPU |
| 1 | 3.93 | 0.00 | 0:03.94 | 99 | 3.94 | 0.00 | 0:03.98 | 99 |
| 2 | 7.95 | 0.00 | 0:03.99 | 199 | 3.95 | 0.00 | 0:01.99 | 198 |
| 4 | 7.93 | 0.05 | 0:04.00 | 199 | 3.98 | 0.01 | 0:01.01 | 392 |
| 8 | 7.98 | 0.05 | 0:04.02 | 199 | 4.17 | 0.01 | 0:00.54 | 768 |

We can assume that in interactive mode as many threads we increase, the elapsed time is near the same and the CPU changes only in the path of having one or more threads. Moreover in the queued version we can see that having more threads reduces the elapsed time and the CPU percentage increases in big steps.

## 2. Systematically analysing task decomposition with Tareador

### 2.1 Tareador API: (guide for us)

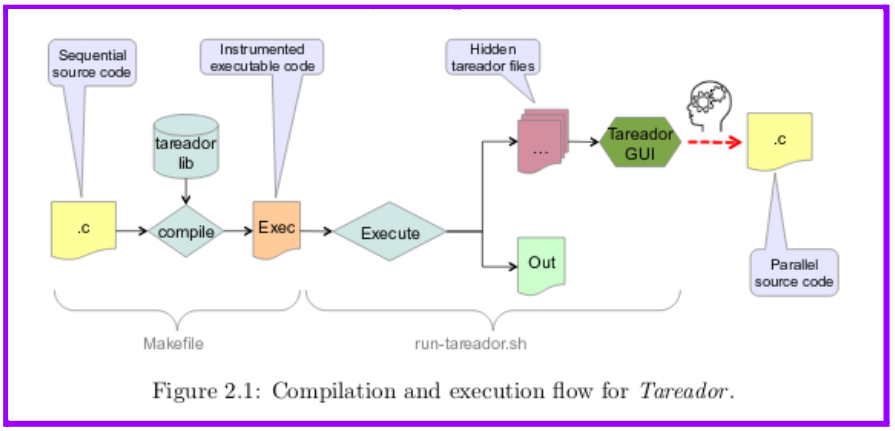



Figure 2.1: Compilation and execution flow for *Tareador*.

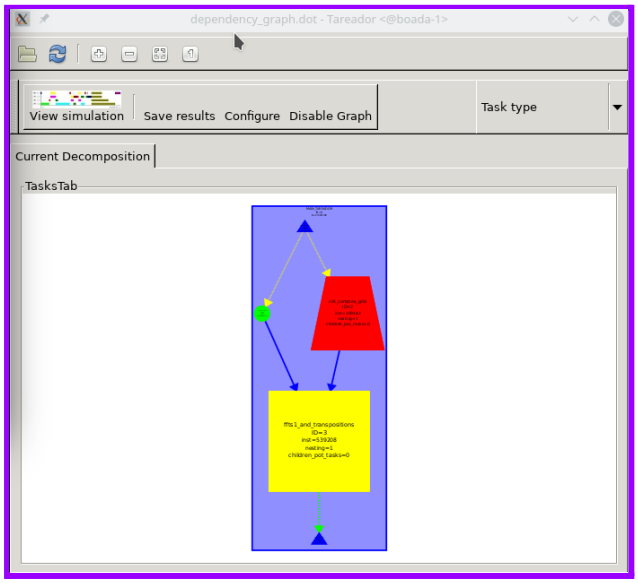
```
** (gedit:2475): WARNING **: 12:13:25.313: Set document metadata failed: Setting attribute metadata::gedi
par2311@boada-1:~/lab1/3dfft$ cat Makefile
CC = icc # gcc
OPENMP   = -fopenmp

TAREADORCC = tareador-clang
TAREADOR_FLAGS = -tareador-lite

CFLAGS  = -Wall -fno-inline -ffloat-store
OPTG0   = -g -O0
OPT3    = -g -O3

CINCL   = -I. -I${FFTW3_HOME}/include
CLIBS   = -L. ${FFTW3_HOME}/lib/libfftw3f.a -lm

TARGETS  = 3dfft_seq 3dfft_omp 3dfft_tar
all: $(TARGETS)

3dfft_seq:      3dfft_seq.c const.h
        $(CC) $(CFLAGS) $(OPT3) $(CINCL) $< -o $@ $(CLIBS)

3dfft_omp:      3dfft_omp.c const.h
        $(CC) $(CFLAGS) $(OPT3) $(OPENMP) $(CINCL) $< -o $@ $(CLIBS)

3dfft_tar:      3dfft_tar.c const.h
        $(TAREADORCC) -DTEST $(CFLAGS) $(OPTG0) $(CINCL) $< -o $@ $(CLIBS) $(TAREADOR_FLAGS)

clean:
        rm -rf $(TARGETS)

ultraclean:
        rm -rf $(TARGETS) .tareador_precomputed_* logs *.log *.prv *.pcf *.row *.txt *.sh.o* *.sh.e* *.ps
par2311@boada-1:~/lab1/3dfft$
```

Image of the Makefile

```
/* Initialize Tareador analysis */
tareador_ON ();
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");

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("fftsl_and_transpositions");
fftsl_planes(p1d, in_fftw);

transpose_xy_planes(tmp_fftw, in_fftw);

fftsl_planes(p1d, tmp_fftw);

transpose_zx_planes(in_fftw, tmp_fftw);

fftsl_planes(p1d, in_fftw);

transpose_zx_planes(tmp_fftw, in_fftw);

transpose_xy_planes(in_fftw, tmp_fftw);
tareador_end_task("fftsl_and_transpositions");

STOP_COUNT_TIME("Execution FFT3D");

/* Finalize Tareador analysis */
tareador_OFF ();
```

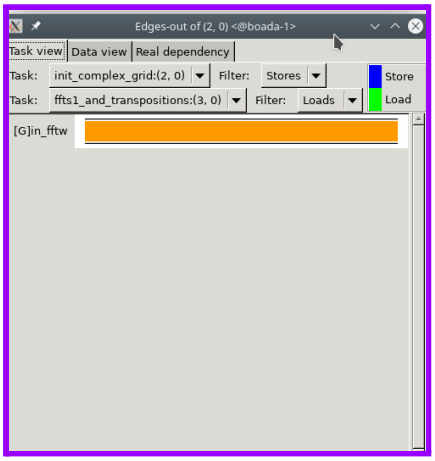
Picture of the code “3dfft_tar.c”

# 2.2 Brief Tareador hands-on: (guide for us)

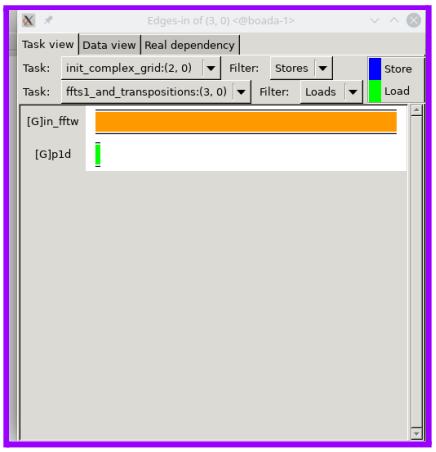



Tareador, ./run-tareador.sh 3dfft_tar
execution with 4 threads, task dependence graph.


Paraver visualisation of
the simulated execution

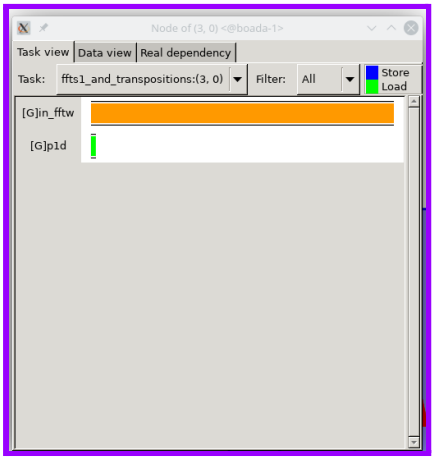



**ffts1 and transpositions**

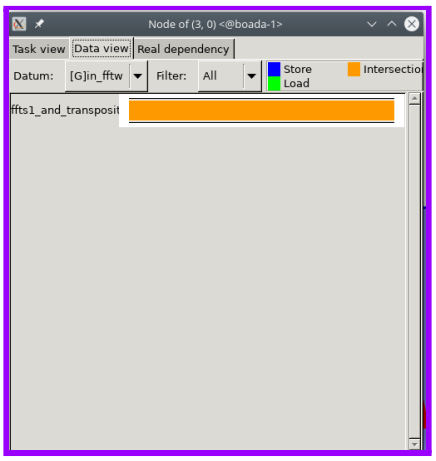



Edges in the graph represent dependencies
between task instances.

the edges going out of a task



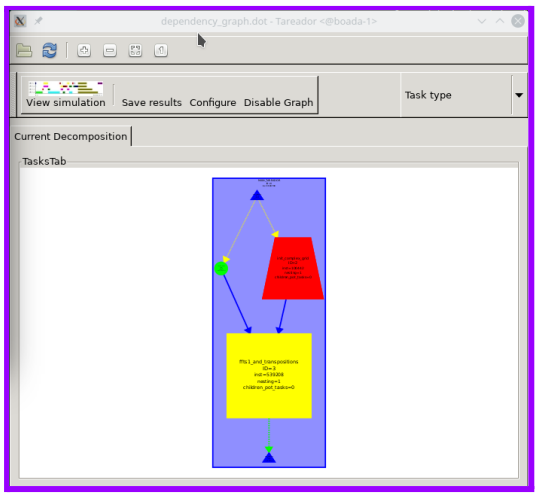

Visualisation of
variables provoking
data dependencies
between tasks



## 2.3 Exploring new task decomposition 3DFFT:

Now we will proceed to study the evolution of parallelism in 5 modifications of the 3dfft_tar.c script redefining task decomposition granularity.

### V0 modification





(v0: tareador graph/ T1 = 639780/critical path: ID0 -> ID2 -> ID3: Tinf = 639760)

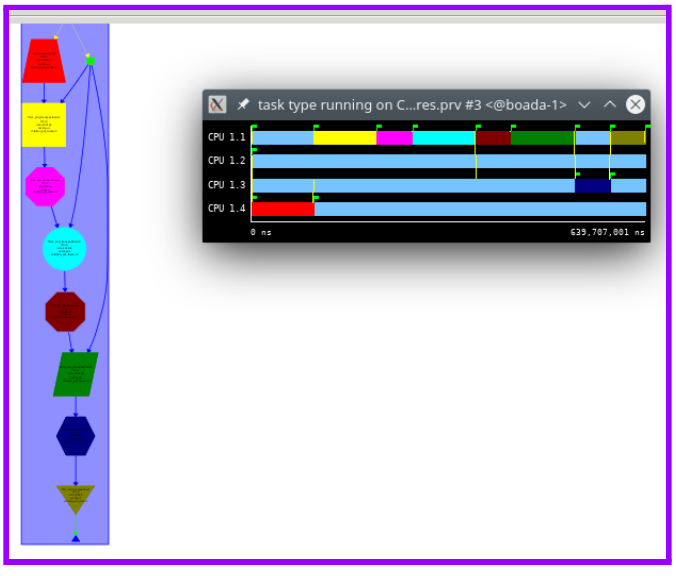
### V1 modification

1. Version v1: REPLACE[1] the task named `ffts1_and_transpositions` with a sequence of finer grained tasks, one for each function invocation inside it.

(v1: code modification)

```
START_COUNT_TIME;

//tareador_end_task("ffts1_and_transpositions");        //previous task decomposition START

tareador_start_task("ffts1_and_transpositions1");
ffts1_planes(p1d, in_fftw);
tareador_end_task("ffts1_and_transpositions1");

tareador_start_task("ffts1_and_transpositions2");
transpose_xy_planes(tmp_fftw, in_fftw);
tareador_end_task("ffts1_and_transpositions2");

tareador_start_task("ffts1_and_transpositions3");
ffts1_planes(p1d, tmp_fftw);
tareador_end_task("ffts1_and_transpositions3");

tareador_start_task("ffts1_and_transpositions4");
transpose_zx_planes(in_fftw, tmp_fftw);
tareador_end_task("ffts1_and_transpositions4");

tareador_start_task("ffts1_and_transpositions5");
ffts1_planes(p1d, in_fftw);
tareador_end_task("ffts1_and_transpositions5");

tareador_start_task("ffts1_and_transpositions6");
transpose_zx_planes(tmp_fftw, in_fftw);
tareador_end_task("ffts1_and_transpositions6");

tareador_start_task("ffts1_and_transpositions7");
transpose_xy_planes(in_fftw, tmp_fftw);
tareador_end_task("ffts1_and_transpositions7");

//tareador_end_task("ffts1_and_transpositions");        //previous task decomposition END


STOP_COUNT_TIME("Execution FFT3D");

/* Finalize Tareador analysis */
```

(v1: tareador graph and cpu management)



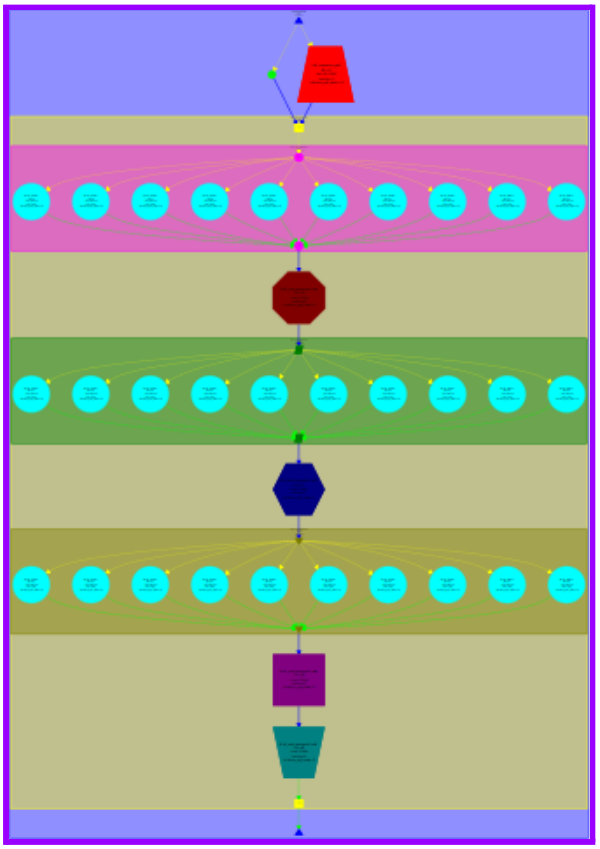
In this first version we have modified the code as the image above shows and we have obtained the results of the picture of the right where we can see the distribution of the different tasks for each CPU. In this case the major time of processing the tasks corresponds to the CPU 1.1 while the others have developed less activity.

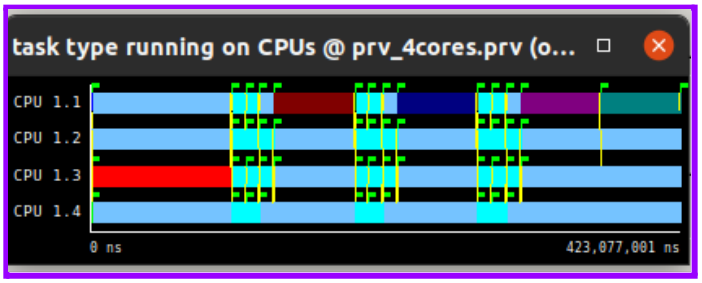
## V2 modification

2. Version v2: starting from v1, REPLACE the definition of tasks associated to function invocations `ffts1_planes` with fine-grained tasks defined inside the function body and associated to individual iterations of the `k` loop, as shown below:

```
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");
     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_start_task("ffts1_planes");

    }
}
```

(v2: code modification, areador graph and cpu management)

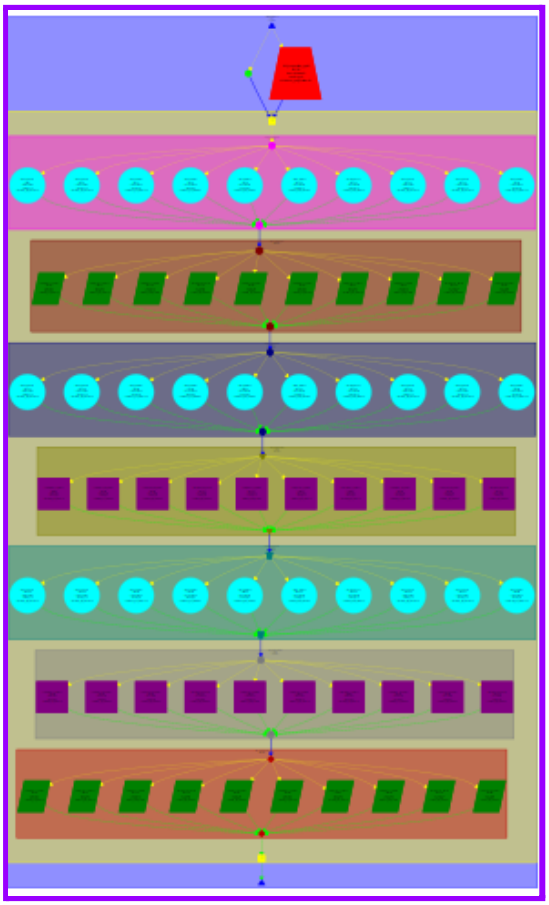





In this second version of the code we have added these two new tareador tasks to the ffts1_planes routine (as shown in the picture) and the simulation has given these images of the parallel process. Now we can see a few regions where the code is parallelized (light blue) and the sequential part (red). The main time processing goes to the CPU 1.1.

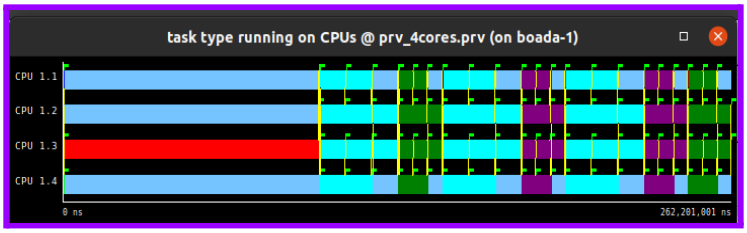
# V3 modification

3. Version v3: starting from v2, REPLACE the definition of tasks associated to function invocations `transpose_xy_planes` and `transpose_zx_planes` with fine-grained tasks inside the corresponding body functions and associated to individual iterations of the k loop, as you did in version v2 for `ffts1_planes`. Again, make sure you understand what is causing data dependences.



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

    for (k=0; k<N; k++) {
    tareador_start_task("transpose_zx_planes");
     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_planes");
    }
}

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");
     for (j=0; j<N; j++) {
       fftwf_execute_dft( p1d, (fftwf_complex *)in_fftw[k][j][0], (fftwf_comple
     }
     tareador_end_task("ffts1_planes");

    }
}
```

(v3: code modification)

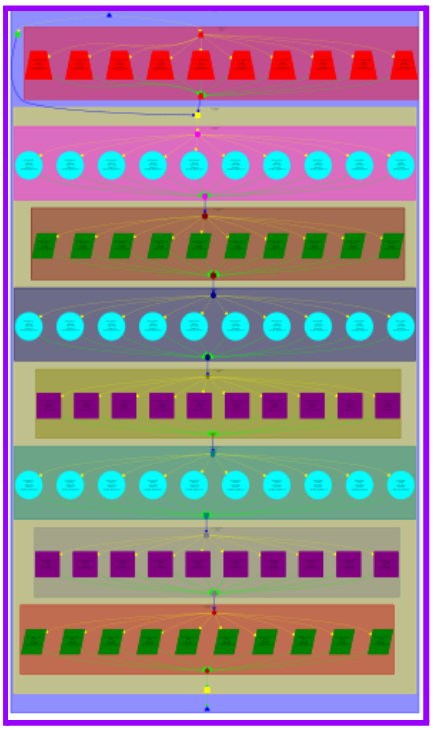



(v3: tareador graph and cpu management)

In this version we have included the different main tasks in the routines explained in the document and we have introduced these tareador tasks as the image above shows. As a conclusion in the results we can see that there is an important time executing the red part in the 1.3 CPU (the sequential ones) and later we expect the parallelization in the different CPUs and how they share the work between them.

# V4 modification

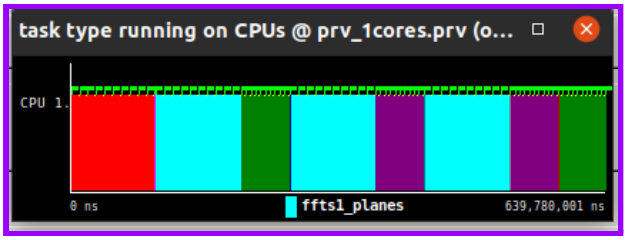
4. Version v4: starting from v3, REPLACE the definition of task for the `init_complex_grid` function with fine-grained tasks inside the body function. For this version v4, also simulate the parallel execution for 1, 2, 4, 8, 16 and 32 processors, drawing a graph or table showing the potential strong scalability. What is limiting the scalability of this version v4?

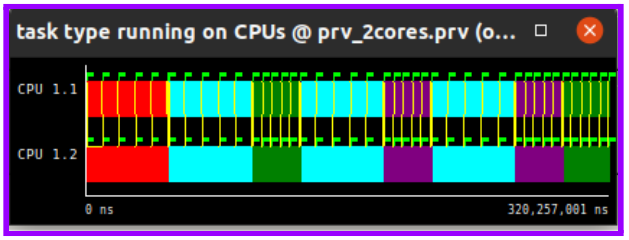


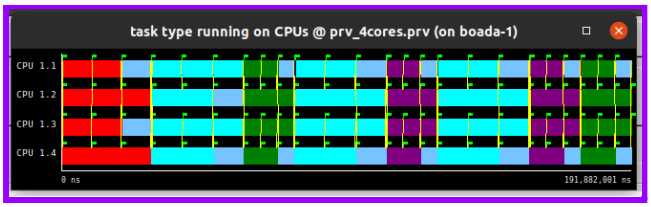
```
void init_complex_grid(fftwf_complex in_fftw[][N][N]) {
  int k,j,i;

  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*((floa
        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");
  }
}
```

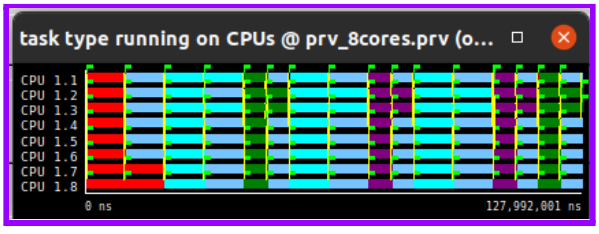
(v4: code modification, tareador graph and cpu management)

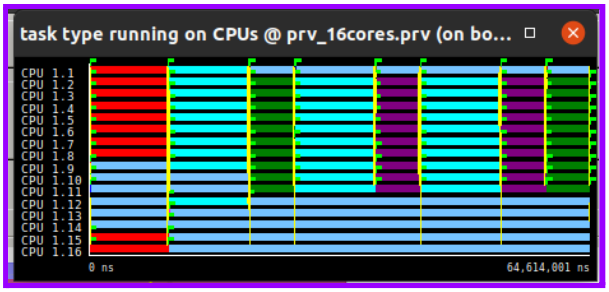
In this part we have added the main tareador tasks in the code, as we can see in the image, and with the different processor we have obtain these simulations:

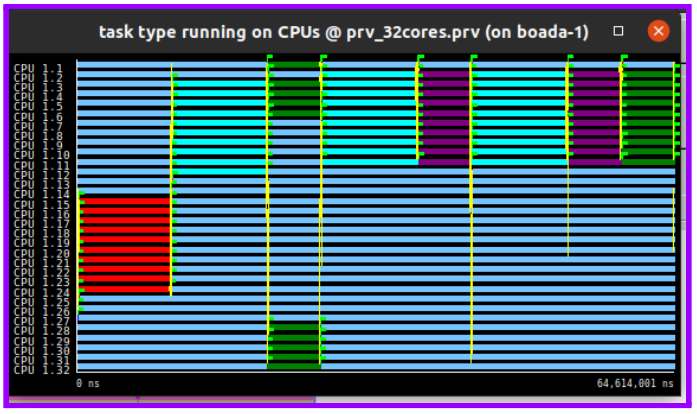


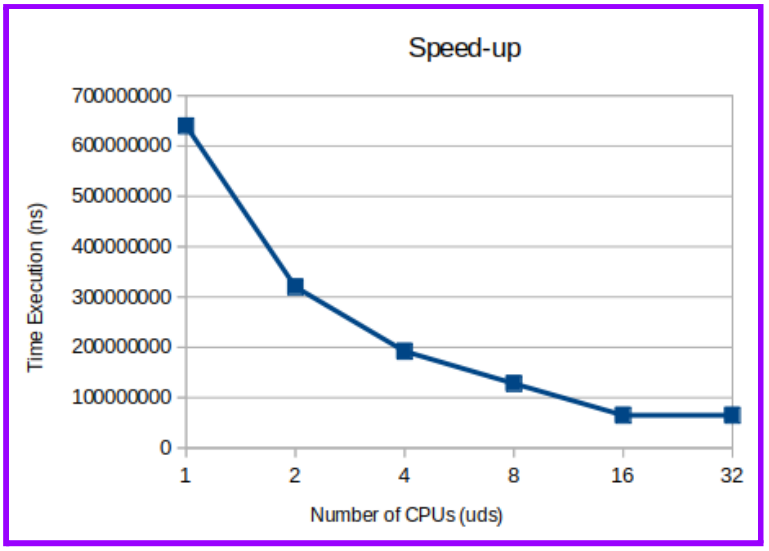



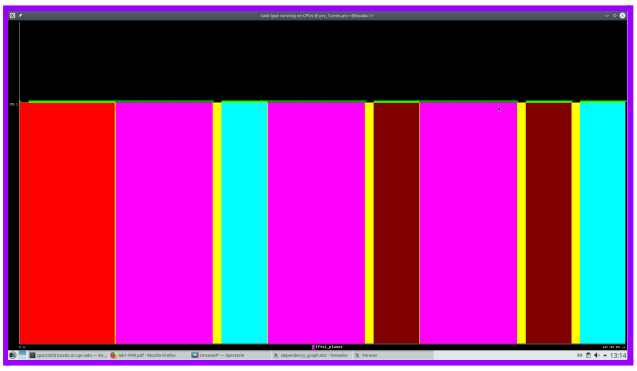



As we can appreciate in the sequential part of the execution there's not a significant benefit. This graphic presents a decreasing line (so the time execution reduces) but it goes deeper on big steps till the number of processors arrive at 10.

So as a conclusion we can see clearly that there is a greater speed-up in the parallel part of the execution till the value of the processors change to 16 where it stops because of the existent dependencies cannot be executed more fast by having more cores free.

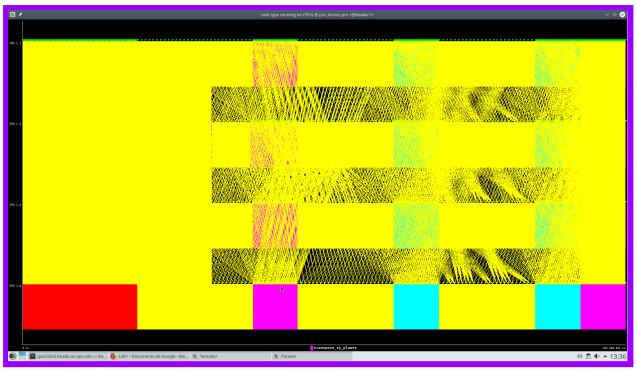
## V5 modification

5. Version v5: finally create a new version in which you explore even finer-grained tasks. Due to the large number of tasks, *Tareador* may take a while to compute and draw the task dependence graph. Please be patient! Again, simulate the parallel execution for 1, 2, 4, 8, 16 and 32 processors, completing the previous graph or plot with the results obtained for version v5. According to the results, is it worth going to this granularity level? When?

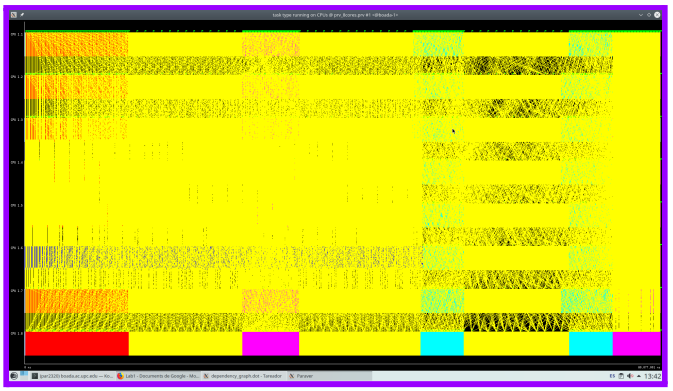


(#Processors = 1)

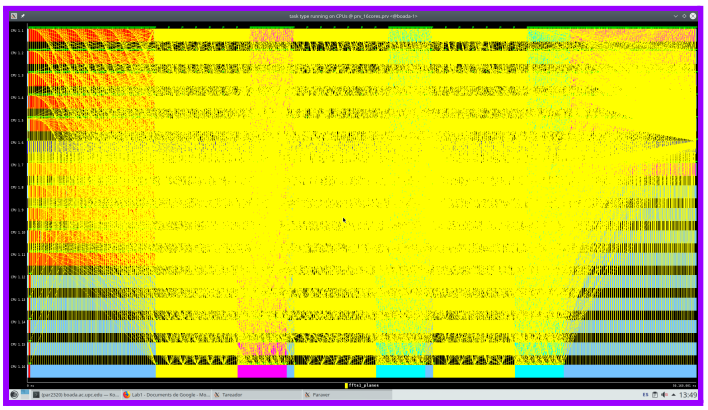
(#Processors = 2)



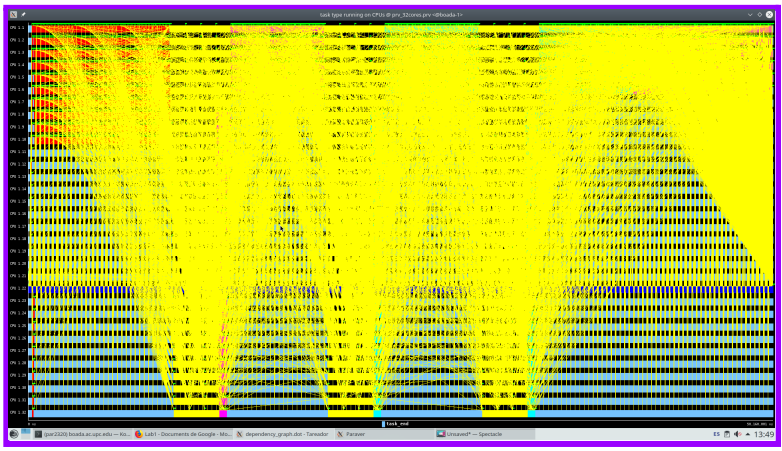
(#Processors = 4)



(#Processors= 8)



(#Processors = 16)

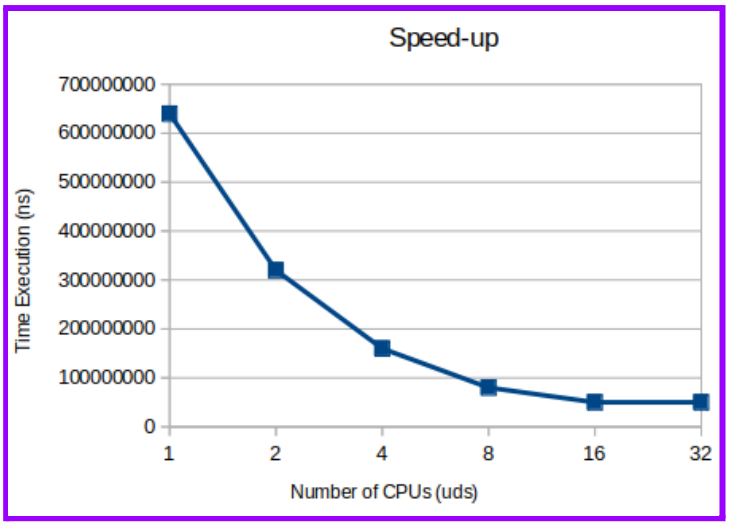


(#Processors = 32)

(v5: code modification)

```
tareador_start_task("init_complex_grid");
        in_fftw[k][j][i][0] = (float) (sin(M_PI*((float)i)/64.0)+sin(M_PI*((float)i)/32.0));
        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");
      }
    }
  }
}

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++) {
       for (i=0; i<N; i++)
       {
tareador_start_task("transpose_xy_planes");
         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");
       }
     }
    }
}
```

```
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++) {
     for (j=0; j<N; j++) {
       for (i=0; i<N; i++)
       {
tareador_start_task("transpose_zx_planes");
         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_planes");
       }
     }
    }
}

void ffts1_planes(fftwf_plan p1d, fftwf_complex in_fftw[][N][N]) {
    int k,j;

    for (k=0; k<N; k++) {
     for (j=0; j<N; j++) {
           tareador_start_task("ffts1_planes");
       fftwf_execute_dft( p1d, (fftwf_complex *)in_fftw[k][j][0], (fftwf_complex *)in_fftw[k][j][0]);
    tareador_end_task("ffts1_planes");
     }
```




In this version, the program's speed up is similar to v4. Although time reduces faster as the number of CPUs increases and we have modified the code to a more parallelizable version, the improvement cannot be fully appreciated in this graph.

## COMPARING VERSIONS

| Version | T1 (ms) | Tinfinte (ms) | Parallelism |
|---|---|---|---|
| **seq** | 639780 | 639760 | 1.00003 |
| **v1** | 639780 | 639760 | 1.00003 |
| **v2** | 639780 | 361199 | 1.77 |
| **v3** | 639780 | 154354 | 4.15 |
| **v4** | 639780 | 64018 | 9.99 |
| **v5** | 639780 | 55820 | 11.46 |

# 3. Understanding the execution of OpenMP programs

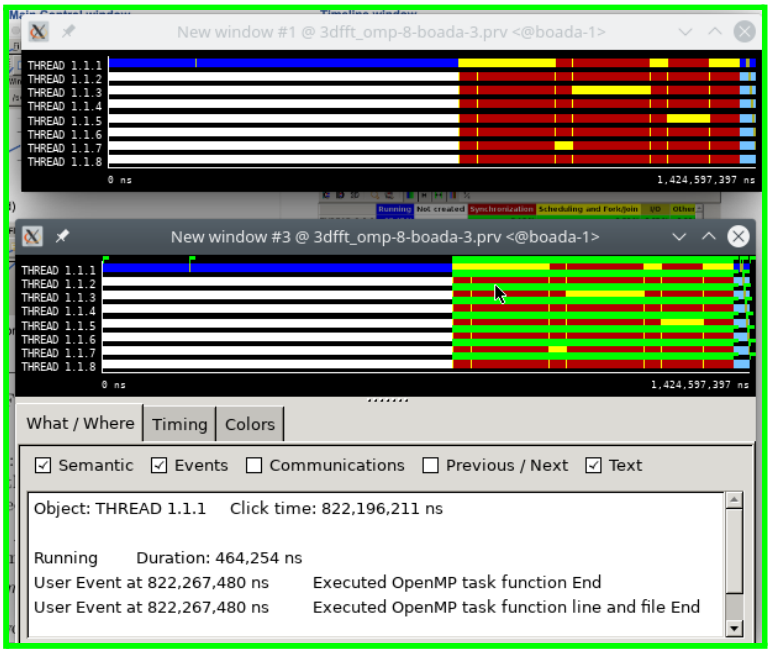
```
eme breeze
par2311@boada-1:~/lab1/3dfft$ cat  3dfft_seq-1-boada-2.txt
3D FFT Plan Generation:0.247698s
Init Complex Grid FFT3D:0.577545s
Execution FFT3D:1.687955s
par2311@boada-1:~/lab1/3dfft$ cat  3dfft_seq-8-boada-2.txt
3D FFT Plan Generation:0.000464s
Init Complex Grid FFT3D:1.188924s
Execution FFT3D:1.739951s
par2311@boada-1:~/lab1/3dfft$
```

```
par2311@boada-1:~/lab1/3dfft$ cat 3dfft_omp-1-boada-3.txt
3D FFT Plan Generation:0.093020s
Init Complex Grid FFT3D:0.580076s
Execution FFT3D:1.717685s
par2311@boada-1:~/lab1/3dfft$ cat 3dfft_omp-8-boada-2.txt
3D FFT Plan Generation:0.008607s
Init Complex Grid FFT3D:0.583731s
Execution FFT3D:0.638115s
```

3dff_seq (1 and 8 threads).txt execution time, 3dff_omp (1 and 8 threads).txt execution time.

## 3.1 Short Paraver hands-on:

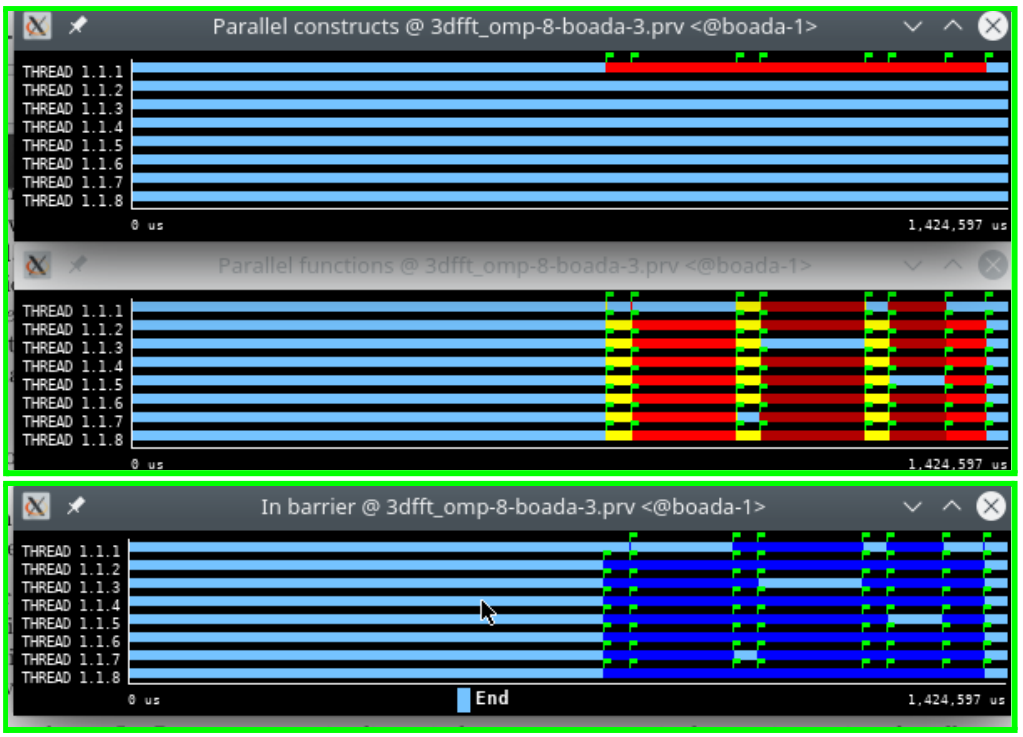
### 3.1.1 Timelines: navigation and basic concepts (guide for us)



Picture of the timeline window (top graphic) and timeline window with flags activated (bottom graphic)

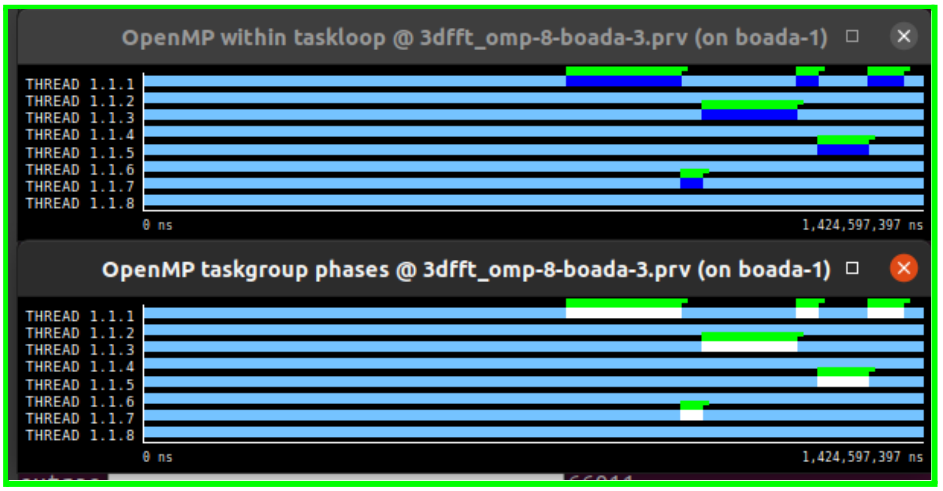
| | Running | Not created | Synchronization | Scheduling and Fork/Join | I/O | Others |
|---|---|---|---|---|---|---|
| THREAD 1.1.1 | 1,320,096,509 ns | - | 61,313,450 ns | 39,315,642 ns | 3,869,561 ns | 2,235 ns |
| THREAD 1.1.2 | 548,639,474 ns | 769,736,412 ns | 66,887,616 ns | 11,422 ns | 2,628,611 ns | - |
| THREAD 1.1.3 | 536,665,575 ns | 769,684,122 ns | 65,713,382 ns | 13,201,034 ns | 2,840,292 ns | - |
| THREAD 1.1.4 | 548,730,873 ns | 769,736,380 ns | 66,796,396 ns | 11,288 ns | 2,563,112 ns | - |
| THREAD 1.1.5 | 540,286,202 ns | 769,715,252 ns | 65,649,420 ns | 9,615,494 ns | 2,856,800 ns | - |
| THREAD 1.1.6 | 549,314,764 ns | 769,736,430 ns | 66,212,345 ns | 11,381 ns | 2,610,548 ns | - |
| THREAD 1.1.7 | 541,104,845 ns | 769,780,534 ns | 64,714,253 ns | 9,665,409 ns | 2,735,435 ns | - |
| THREAD 1.1.8 | 549,008,958 ns | 769,862,264 ns | 66,397,004 ns | 9,350 ns | 2,564,501 ns | - |
| | | | | | | |
| Total | 5,133,847,200 ns | 5,388,251,394 ns | 523,683,866 ns | 71,841,020 ns | 22,668,860 ns | 2,235 ns |
| Average | 641,730,900 ns | 769,750,199.14 ns | 65,460,483.25 ns | 8,980,127.50 ns | 2,833,607.50 ns | 2,235 ns |
| Maximum | 1,320,096,509 ns | 769,862,264 ns | 66,887,616 ns | 39,315,642 ns | 3,869,561 ns | 2,235 ns |
| Minimum | 536,665,575 ns | 769,684,122 ns | 61,313,450 ns | 9,350 ns | 2,563,112 ns | 2,235 ns |
| StDev | 256,439,167.01 ns | 52,964.77 ns | 1,699,931.51 ns | 12,553,504.57 ns | 406,322.73 ns | 0 ns |
| Avg/Max | 0.49 | 1.00 | 0.98 | 0.23 | 0.73 | 1 |

Picture of the analyzer window of the execution form the above page




Picture of the constructs, functions and barrier configuration files of the omp simulation.

## 3.1.2 Explicit task (guide for us)




Pictures of the OMP_taskloop and OMP_in_taskgroup.

## 3.1.3 Profile (guide for us)

| | Running | Not created | Synchronization | Scheduling and Fork/Join | I/O | Others |
|---|---|---|---|---|---|---|
| THREAD 1.1.1 | 92.66 % | - | 4.30 % | 2.76 % | 0.27 % | 0.00 % |
| THREAD 1.1.2 | 39.53 % | 55.46 % | 4.82 % | 0.00 % | 0.19 % | - |
| THREAD 1.1.3 | 38.66 % | 55.45 % | 4.73 % | 0.95 % | 0.20 % | - |
| THREAD 1.1.4 | 39.54 % | 55.46 % | 4.81 % | 0.00 % | 0.18 % | - |
| THREAD 1.1.5 | 38.92 % | 55.45 % | 4.73 % | 0.69 % | 0.21 % | - |
| THREAD 1.1.6 | 39.58 % | 55.46 % | 4.77 % | 0.00 % | 0.19 % | - |
| THREAD 1.1.7 | 38.98 % | 55.46 % | 4.66 % | 0.70 % | 0.20 % | - |
| THREAD 1.1.8 | 39.56 % | 55.47 % | 4.78 % | 0.00 % | 0.18 % | - |
| | | | | | | |
| Total | 367.44 % | 388.21 % | 37.62 % | 5.10 % | 1.63 % | 0.00 % |
| Average | 45.93 % | 55.46 % | 4.70 % | 0.64 % | 0.20 % | 0.00 % |
| Maximum | 92.66 % | 55.47 % | 4.82 % | 2.76 % | 0.27 % | 0.00 % |
| Minimum | 38.66 % | 55.45 % | 4.30 % | 0.00 % | 0.18 % | 0.00 % |
| StDev | 17.67 % | 0.01 % | 0.16 % | 0.88 % | 0.03 % | 0 % |
| Avg/Max | 0.50 | 1.00 | 0.98 | 0.23 | 0.75 | 1 |

Picture of the OMP_state_profile were shows on each cell value shows the absolute time spent by a thread in a specific state.

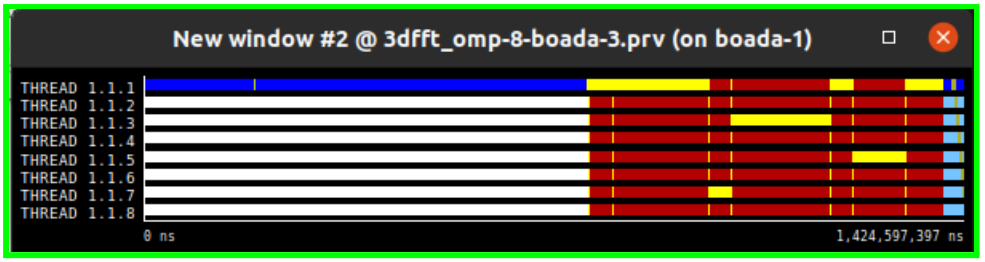
Task profile (execution and instantiation) @ 3dfft_omp-8-boada-3.prv (on boada-1)

| | Executed OpenMP task function | Instantiated OpenMP task function |
|---|---|---|
| THREAD 1.1.1 | 20,064 | 1,024 |
| THREAD 1.1.2 | 16,996 | - |
| THREAD 1.1.3 | 18,648 | 256 |
| THREAD 1.1.4 | 16,985 | - |
| THREAD 1.1.5 | 18,546 | 256 |
| THREAD 1.1.6 | 16,971 | - |
| THREAD 1.1.7 | 18,184 | 256 |
| THREAD 1.1.8 | 16,966 | - |
| | | |
| Total | 143,360 | 1,792 |
| Average | 17,920 | 448 |
| Maximum | 20,064 | 1,024 |
| Minimum | 16,966 | 256 |
| StDev | 1,068.12 | 332.55 |
| Avg/Max | 0.89 | 0.44 |

Picture of the OMP_task_profile to count tasks.

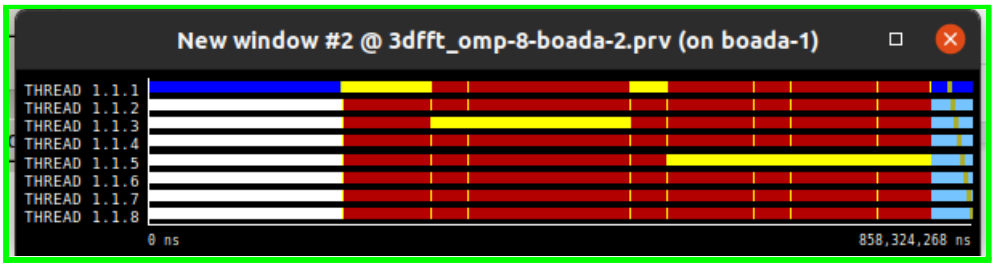
## 3.2 Obtaining parallelisation metrics for 3DFFT using Paraver

### 3.2.1 Initial Verison



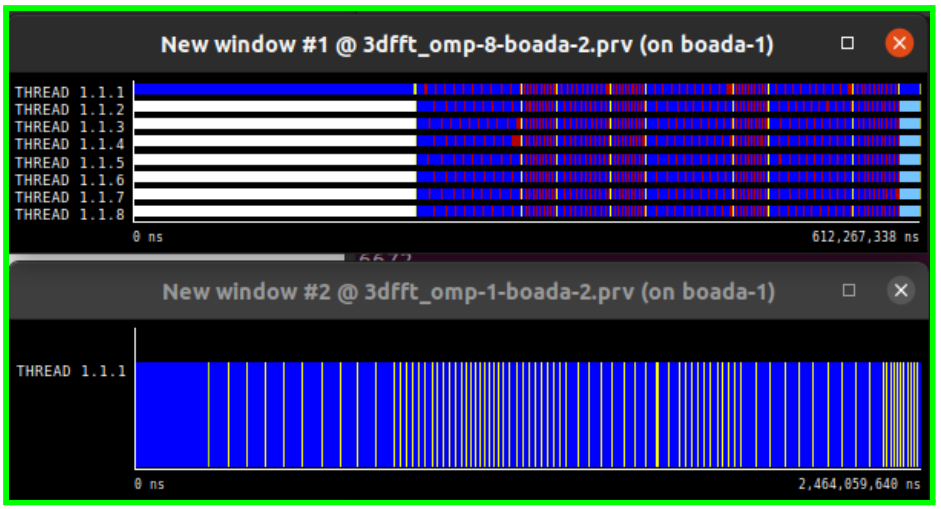

Picture of the Initial version of the execution (main reference to compare the others)

### 3.2.2 Improving



Picture of the improving version of the execution.
We have discomment the comments on the initgrid function assuming a bigger parallelizable time but increasing also the synchronisation.

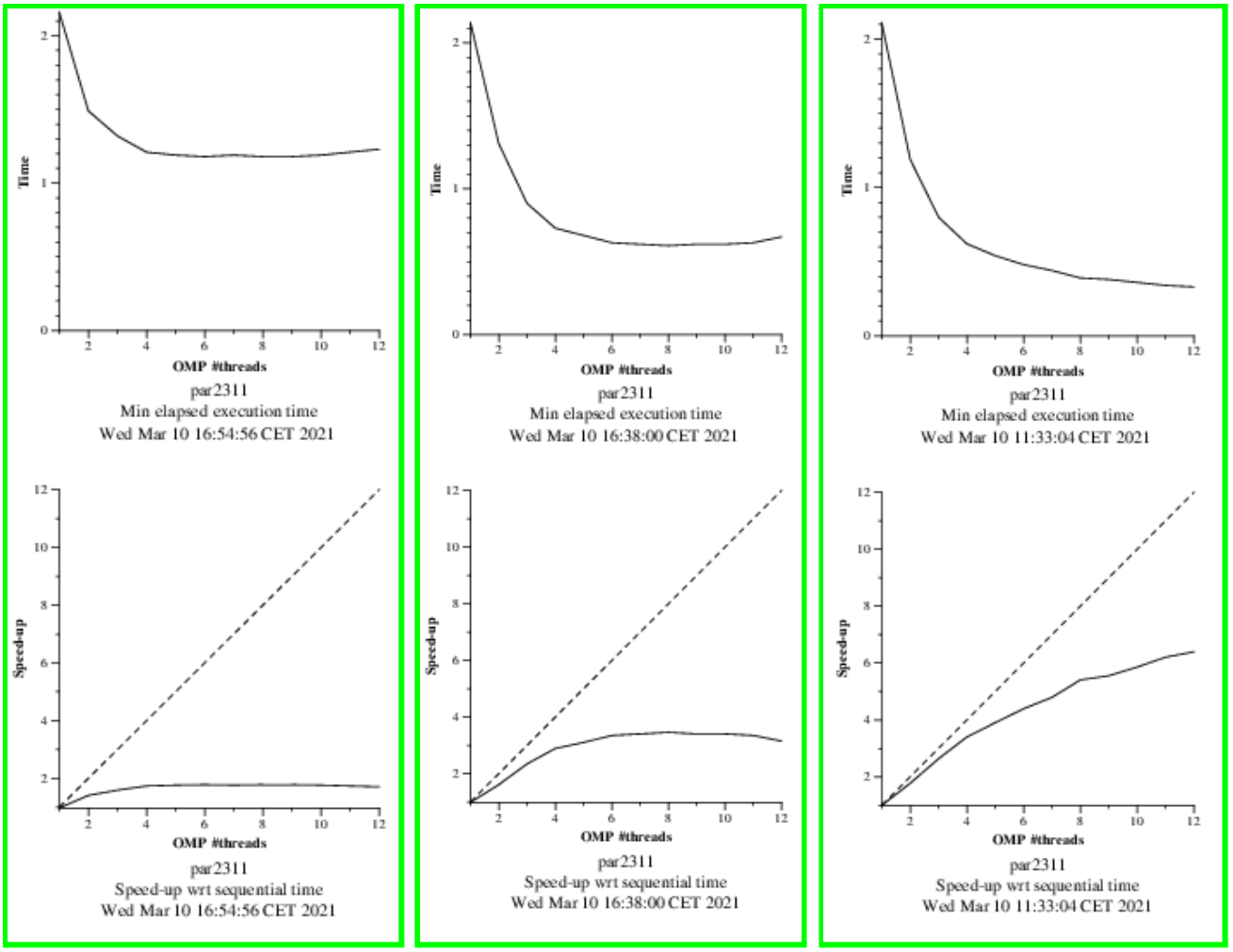
## 3.2.3 Reducing parallelisation overheads




Picture of the last version of the execution with 8 processors and 1 processor.
we have moved the taskloop sentences a step above so we have managed to obtain a bigger granularity to complete big tasks and also we decrease the synchronism.

## Understanding the parallel execution of 3DFFT

| Version | Phi | S(inf) | T1 (ns) | T8 (ns) | S8 |
|---|---|---|---|---|---|
| Initial version 3dfft_omp.c | 0,65 | 2,9 | 2.406.815.125 | 1.424.597.397 | 1,69 |
| New version with improved phi | 0.9 | 10 | 2.499.025.252 | 858.324.268 | 2,91 |
| Final version with reduced parallelisation overheads | 0.89 | 9.09 | 2.464.059.640 | 612.267.338 | 4,02 |




Plot of the initial version  Plot of the improving version  Plot reducing overheads

On the left plot we can appreciate that the speed up maintains as the value of the threads is 4. So that means that we don't need to increase the number of threads because the velocity won't increase as the plot shows.

On the middle plot  we can see that the speed up increases more with 4 threads, compared with the left plot, but also we observe that it makes a little curve that increases a  little bit as we raise the number of threads till the 7 value. Later it maintains and finally decreases a quite.

Finally on the right plot shows that the speed up goes up while the number of threads increases but it belongs to create a stable curve (we can appreciate it in the graph). Compared with the rest of the plots this one shows that it is faster.