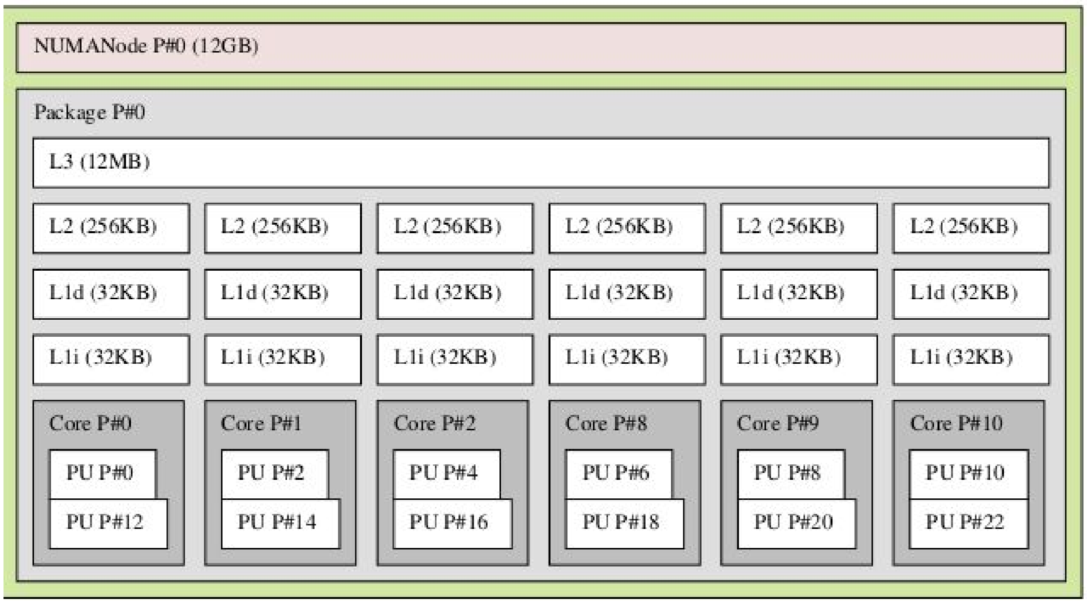
Lab1: Experimental setup and tools

par2103; Joan Sánchez García, Marc Domínguez de la Rocha

Fall 2017-2018 20/10/17

Node architecture and memory

1. Draw and briefly describe the architecture of the computer in which you are doing this lab session (number of sockets, cores per socket, threads per core, cache hierarchy size and sharing, and amount of main memory).



The figure above has been created by the execution of:

$user: **lstopo** --of fig map.fig

Actually, that's not the whole figure obtained by **lstopo**, it is just a portion. Our node is

composed by twelve Cores in two sockets, with two threads per core. Each core has three levels of cache, where two of these levels are **distributed** (private memory, only visible to it's owning core) and the third level is **shared** ("public" memory between the cores of a socket). At the highest level of memory, each socket of the node has 12 GB of RAM also **shared** between cores.
Note that the first level has a data cache but also an instruction cache.

Timing sequential and parallel executions

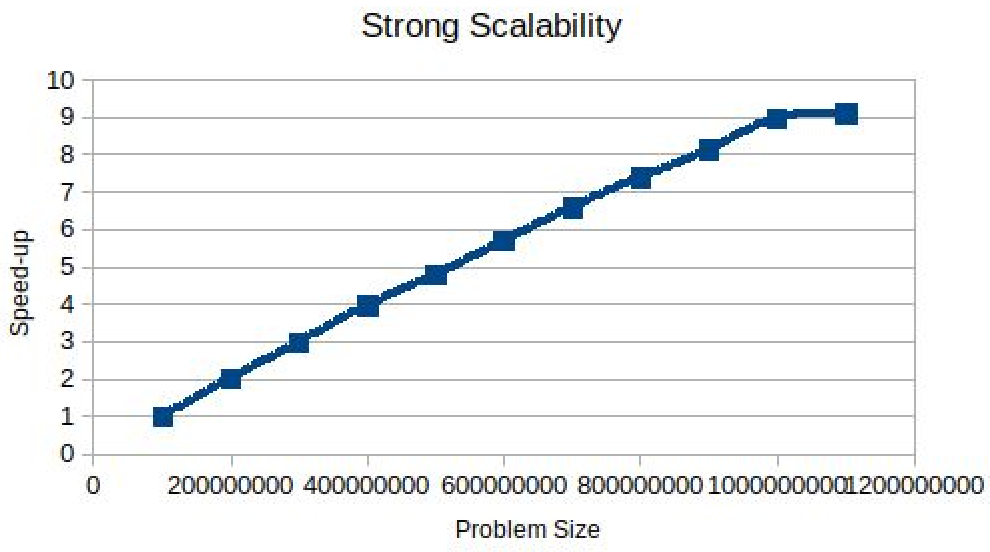
2. Describe what do you need to add to your program to measure the elapsed execution between a pair of points in the program, clearly indicating the library header file that needs to be included, the library functions that need to be invoked, the data structure and its fields.

You need to include the **sys/time.h** library in order to use:

int gettimeofday(struct timeval **tv*, **struct timezone** **tz***);**
Where *tv is a pointer to a **timeval** struct, which contains a variable in seconds and another in µseconds, and *tz is a pointer to a **timezone** pointer, which is not used in our case (set as a NULL pointer).

So we will define START, which will execute **getusec_()**. That function instantiates a timeval struct to call **gettimeofday(...)** and get the initial time of execution.
Once defined, we define STOP, which will do something like, **final_time - initial_time**.
Then, if we type **START** at the start of the main, and **STOP** at the end, we can obtain the **elapsed time** of execution.

3. Plot the speed-up obtained when varying the number of threads (strong scalability) and problem size (weak scalability) for pi_omp.c. Reason about how the scalability of the program.

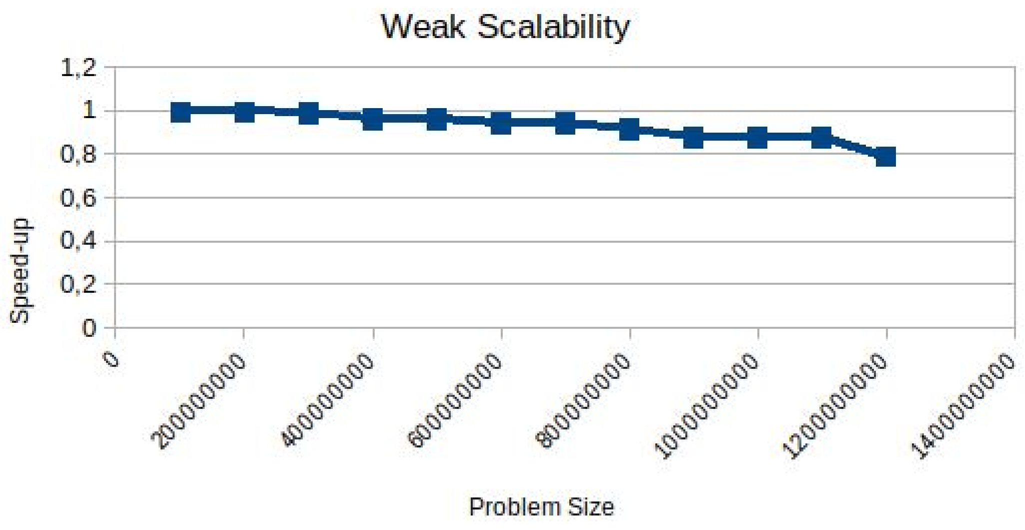


We first tested the **strong scalability** of pi_omp, which consists in increasing the number of threads our program uses, and then we tested the weak scalability increasing the number of threads proportionally to the problem size (i.e 100 iterations 1 thread, 200 iterations 2 threads…).

The program starts very well with the strong scalability one: for each thread you add there's an increment of the speed-up, obviously, the one thread speed-up must be one (or near) because is exactly the same as the sequential version (one thread). But the scalability we want to see, is in crescendo per thread added. If we observe from 1 thread to 4, the speed-up is almost perfect, **Nº of threads = Speed-up achieved**.
But from 4, if we add a new thread the speed-up decreases a little bit from the ideal we want to get (N = Speed-up), and the more threads you add, the more visible will get that non-ideal speed-up.

We could show this idea letting **k** be inversely proportional to N:

Speed-up = N*k



Weak scalability test shows how the speed-up behaves when changing the number of threads and the problem size. Theoretically, we expect the speed-up to maintain constant, but it's not true on practice.
If we have a K problem size with N threads, it's logical to think that for 2K problem size and N+1 threads we'll get the same speed-up, which is one. But we observe that this is not exactly true, in addition of threads and problem size the speed-up ends decreasing. You can see that for 12 threads and 1200000000 iterations we got 0.7876 speed-up (!= 1).

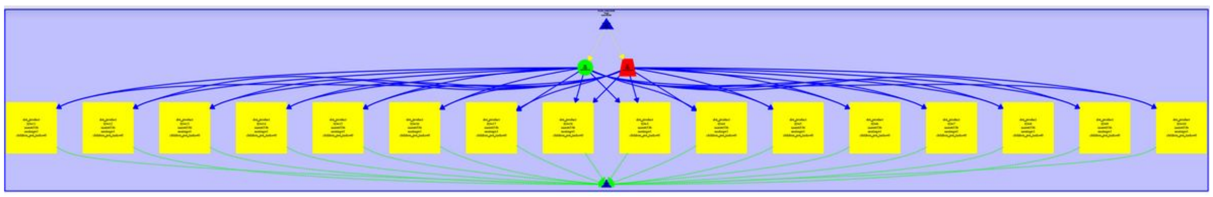
Visualizing the task graph and data dependences

4. Include the source code for function dot product in which you show the Tareador instrumentation that has been added to study the potential parallelism in the code. This instrumentation has to appropriately define tasks and filter the analysis of variable(s) that cause the dependence(s).

```
void dot_product (long N, double A[N], double B[N], double *acc){
        double prod;
        int i;
        *acc=0.0;
         for (i=0; i<N; i++) {
                tareador_start_task("dot_product");
                /* Code region to be a potential task */
                prod = my_func(A[i], B[i]);
                tareador_disable_object(acc);
                // ... code region with memory accesses to variable acc
                *acc += prod;
                tareador_enable_object(acc);
                tareador_end_task("dot_product");
   }
}
```

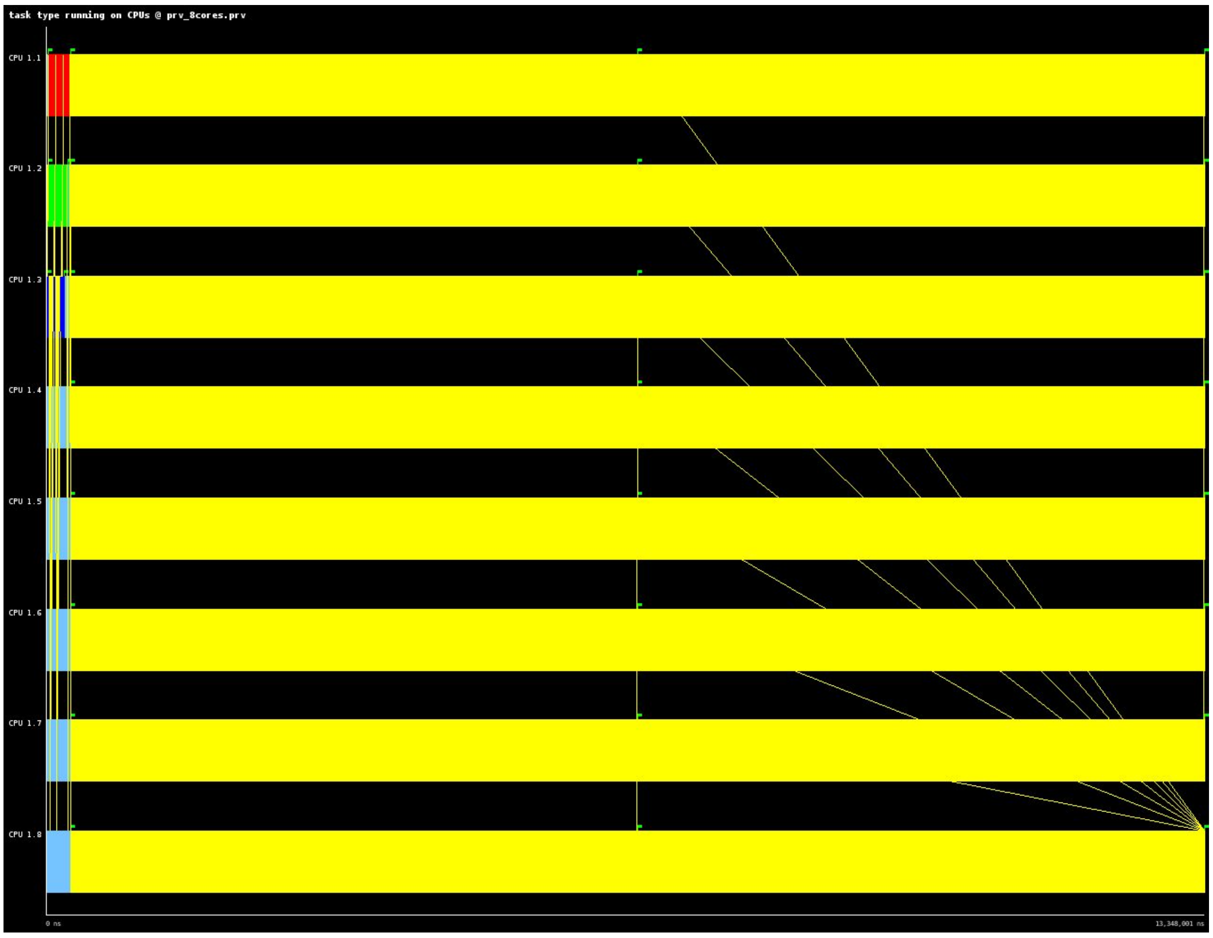
With tareador_disable_object(acc) we are removing the acc dependence from the analysis

5. Capture the task dependence graph for that task decomposition and the execution timelines (for 8 processors) that allow you to understand the potential parallelism attainable. Briefly comment the relevant information that is reported by the tools.



Dependency graph for 8 processors: HD image at Annex dependency_graph_8procs.pdf

The first two levels of the graph are the Tareador setup process and the tasks init_A and init_B, which can be performed at the same time due to they are working on two different vectors. Level 3 is composed by the 16 iterations done by the dot_product function. If we would have not disabled the acc dependence we would have seen 16 levels and not just 1 with 16 parallel operations. Each one of this operations have the same weight because it is the same code working on different parts of the vectors.



dot_product: execution timelines for 8 processors

The image above shows the tasks performed by each processor during the parallel execution. At the beginning of the graph (on the leftmost) we see a small portion with red blue and green colors which are init_A, init_B and the main tareador tasks. Although this just needs 3 processors, all the others must wait to them to finish due to dependences.
The yellow part corresponds to the parallel execution of the dot_product function. If we take a closer look we will see a vertical line just in the middle of execution, this is because a change of task: 8 processors must perform the 16 blocks, so they split it by taking two tasks each. At the end of the execution all tasks synchronize at the eighth processor.

Analysis of task decompositions

6. Complete the following table for the initial and different versions generated for 3dfft seq.c, briefly commenting the evolution of the metrics with the different versions.

Version	T1	T∞	Parallelism
Seq	593772	593758	1.00002

V1	593772	593758	1.00002
V2	593772	315523	1.88186
V3	593772	109063	5.44430
V4	593772	60148	9.87184

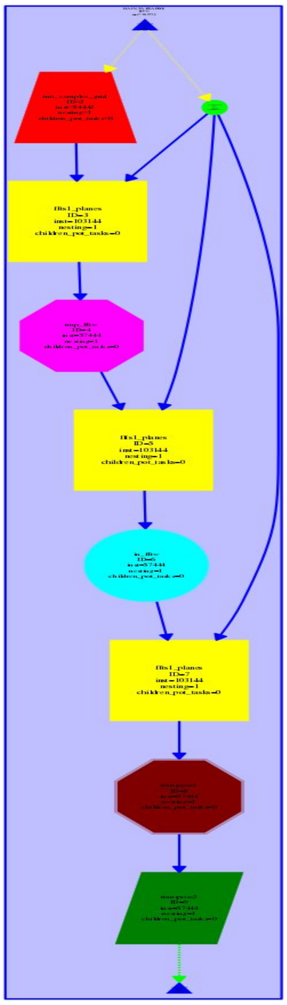
Seq

T1 is the number of instructions executed if we would just had one processor, so it is the sum of all the instructions, in the case of our code they are 593772. This is computed in the same way in all of the versions we are going to examine.

T∞ is how many instructions does the longest path execute (critical path) To compute the time we would have to multiply this result by the time/instruction . In this case
T∞ = 108 + 54442 + 539208 = 593758 instructions

Parallelism is obtained dividing T1/T∞. In the sequential execution is 1.00002

V1

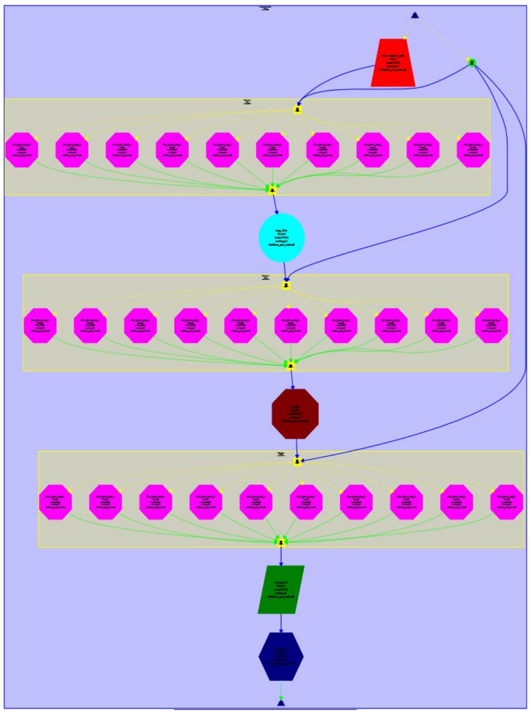


Dependency graph for v1: HD image at Annex dependency_graph_v1.png and code at 3dfft_v1.c

In the first improve we split the task "ffts1_and_transpositions" (block of 539208 instructions) into 7 smaller parts, but as any of them can be parallelized with the others due to dependencies, we obtain the same T∞ and parallelism.

T∞ = 108 + 54442 + 103144 + 57444 + 1003144 + 57444 + 103144 + 57444+ 57444 = 593758 ins

V2



Dependency graph for v2: HD image at Annex dependency_graphv2.pdf and code at 3dfft_v2.c

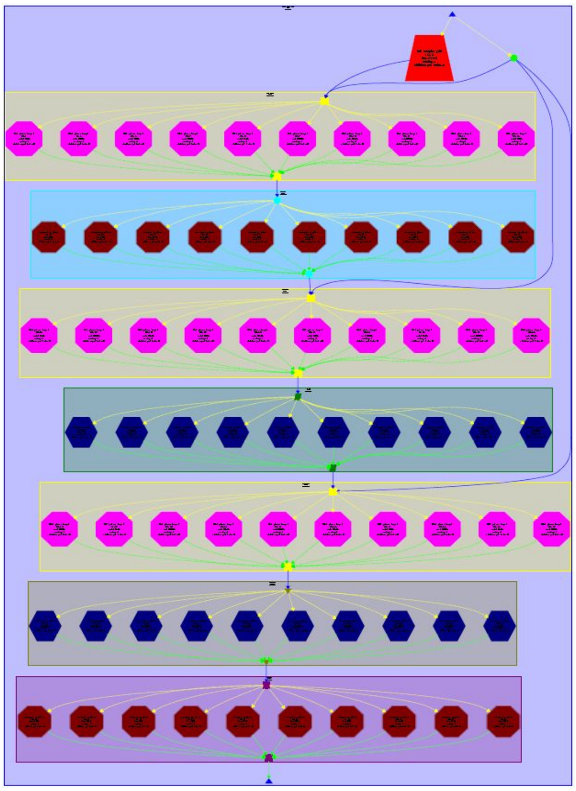
In this case we parallelize the function ffts1_planes.

T∞ = (108 + 54442 + 94 + 10305 + 57444 + 94 + 10305 + 57444 + 94 + 10305 + 57444 + 57444) instr. = 315523 ins

As we start applying parallelism to some loops, the execution time decreases, and the parallelism increases.

Parallelism = 593772/315.523 = 1.88186

V3



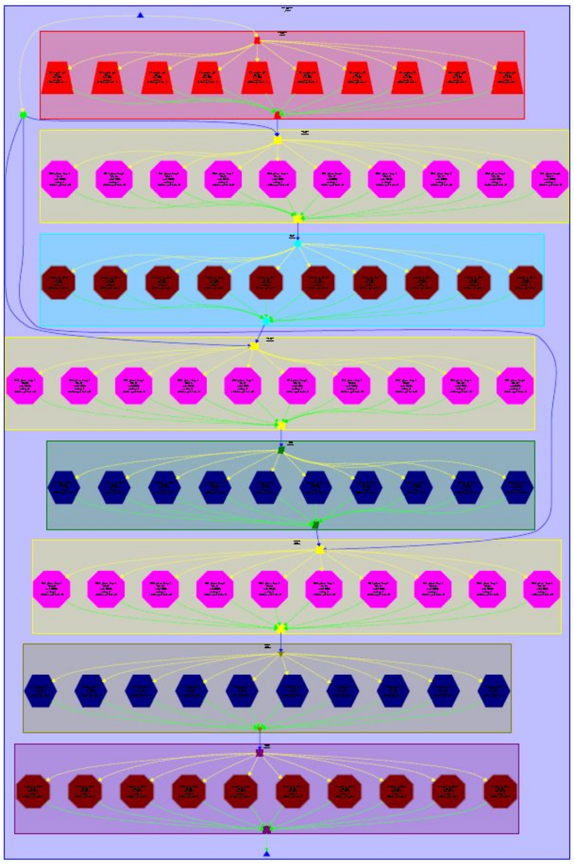
Dependency graph for v3: HD image at Annex dependency_graphv3.pdf and code at 3dfft_v3.c

As in v2, we replace transpose_xy_planes and transpose_zx_planes with fine-grained tasks, parallelizing the inner loops of the functions.

T∞ = (108 + 54442 + 94 + 10305 + 94 + 5735 + 94 + 10305 + 94 + 5735 + 94 + 10305 + 94 + 5735 + 94 + 5735) instr. = 109063 ins

Parallelism = 593772/109063 = 5.44430

V4



Dependency graph for v4: HD image at Annex dependency_graphv4.pdf and code at 3dfft_v4.c

In this part of the exercise we were asked which part we should now granulate to obtain even more parallelism. We opted for the init_complex_grid function, the only loop where we had not applied parallelization.

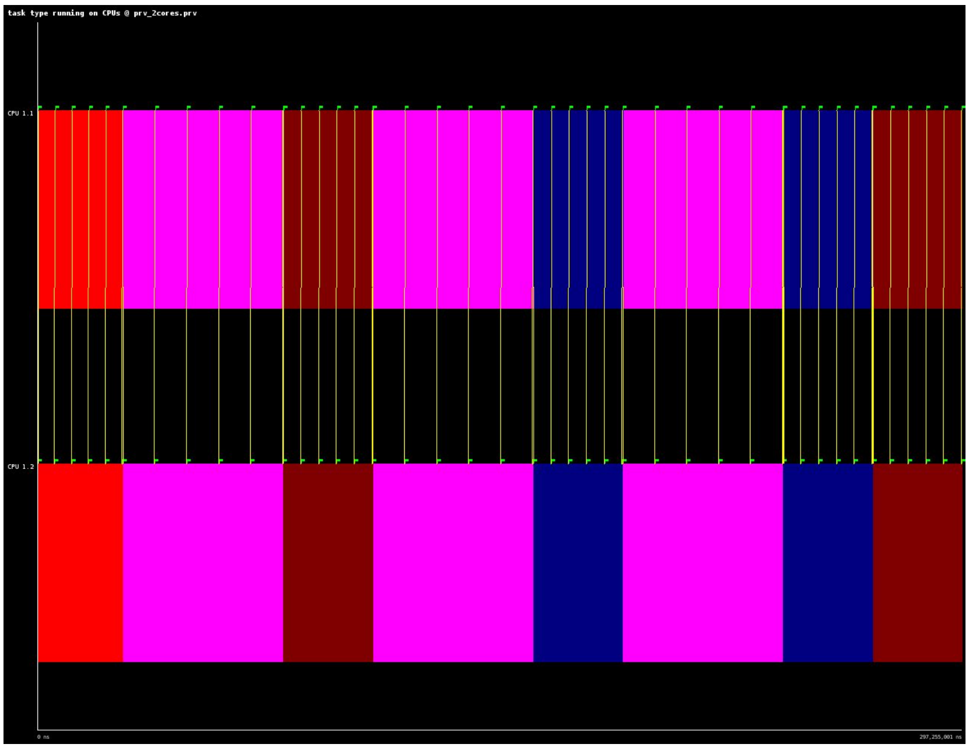
T∞ = (108 + 92 + 5435 + 94 x 7 + 10305 x 3 + 5735 x 4) instr. = 60148 ins

Parallelism = 593772/60148 = 9.87184

7. With the results from the parallel simulation with 2, 4, 8, 16 and 32 processors, draw the execution time and speedup plots for version v4 with respect to the sequential execution (that you can estimate from the simulation of the initial task decomposition that we provided in 3dfft seq.c, using just 1 processor). Briefly comment the scalability behaviour shown on these two plots.

Dividing the total number of instructions between the total execution time we obtain the time/instruction, which in our case is approximately 1000 ns/instrucción. With this we can obtain the execution time of each version just multiplying the number of instructions executed by 1000. We will take the execution time of the sequential version as a basis.
Sequential 1 proc -> 593772001 ns

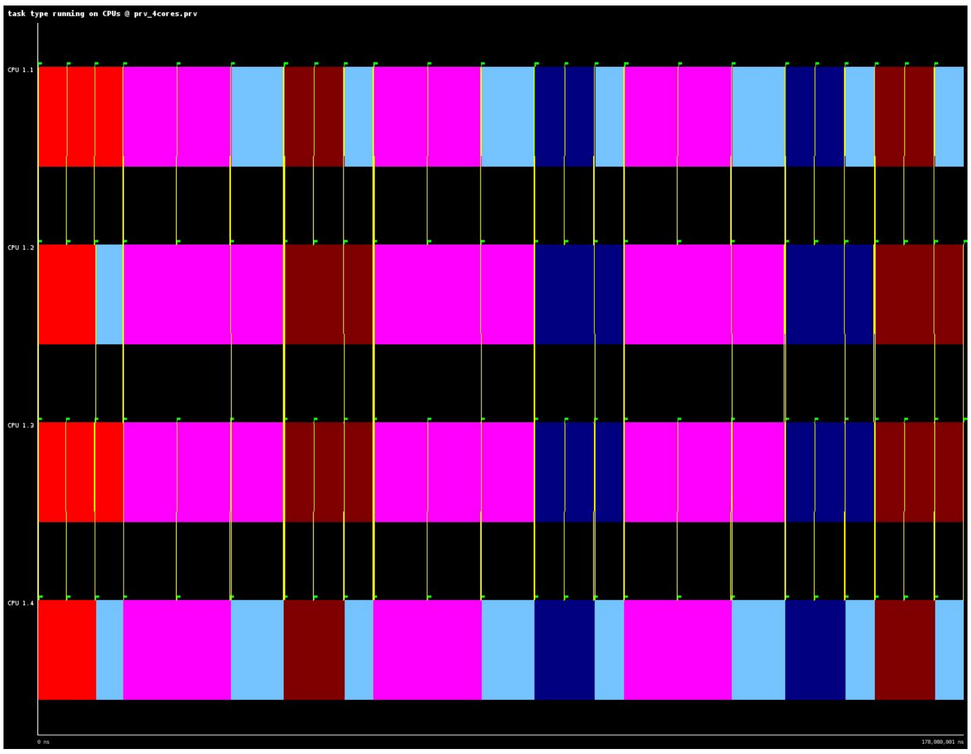
-v4 parallel 2 procs -> 297255001 ns
    speedup = 593.772.001 / 297.255.001 = 1,99751728



execution timelines for 2 processors

-v4 parallel 4 procs -> 178080001 ns

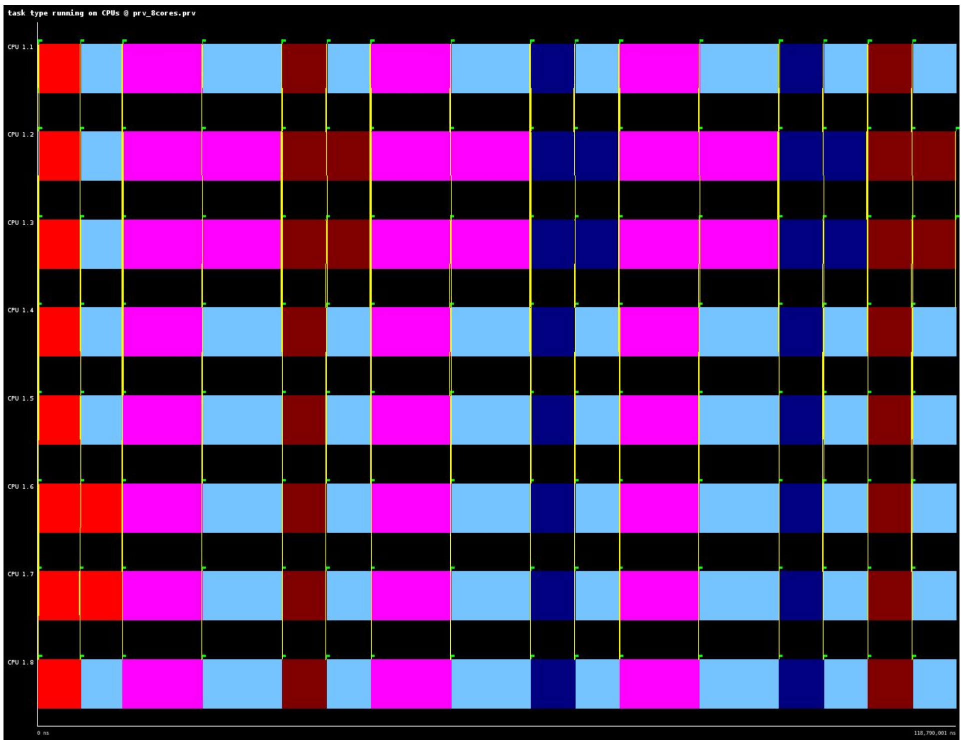
speedup = 593.772.001 / 178.080.001 = 3,334299178



execution timelines for 4 processors

-v4 parallel 8 procs -> 118790001 ns

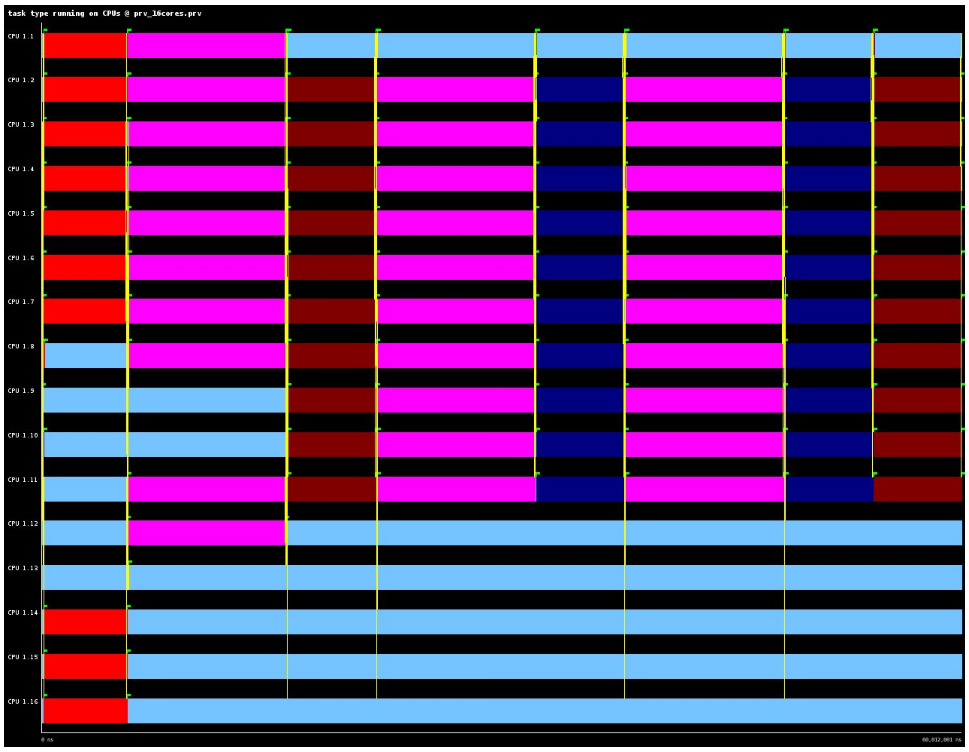
speedup = 593.772.001 / 118.790.001 = 4,998501566



execution timelines for 8 processors

-v4 parallel 16 procs -> 60012001 ns

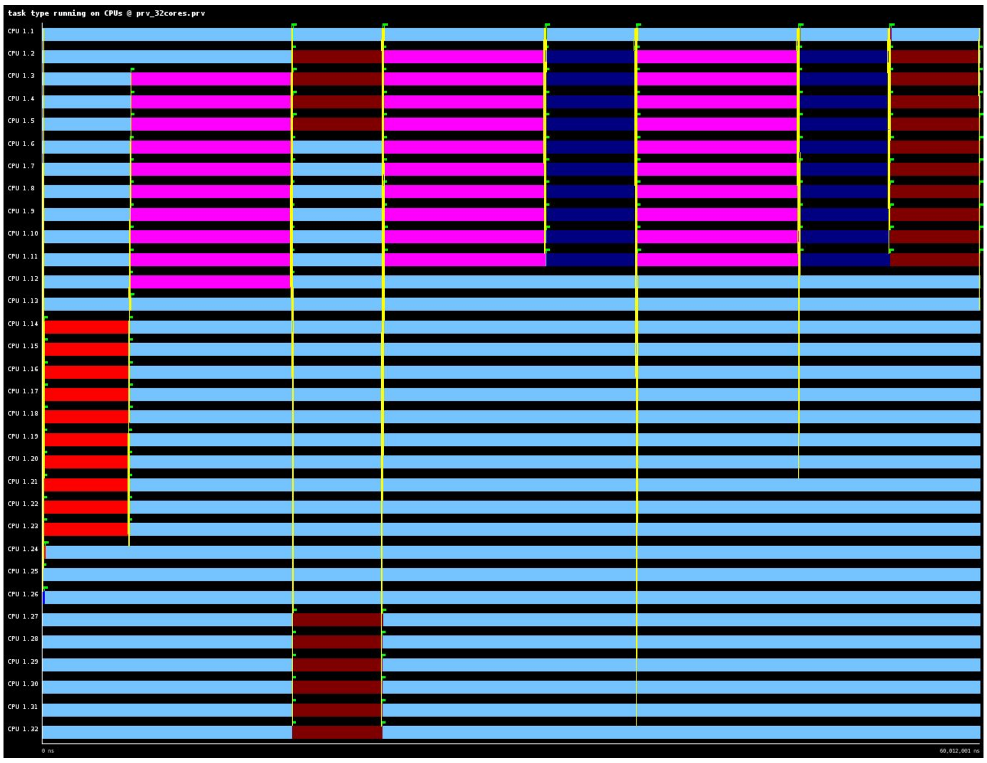
speedup = 593.772.001 / 60.012.001 = 9,894221008



execution timelines for 16 processors

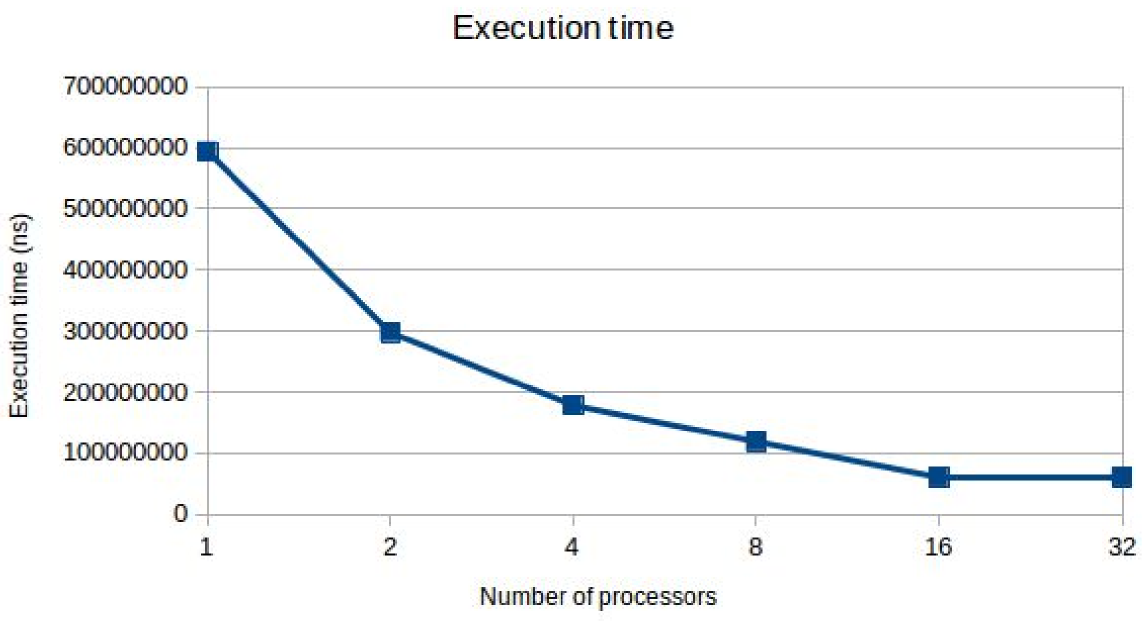
-v4 parallel 32 procs -> 60012001 ns

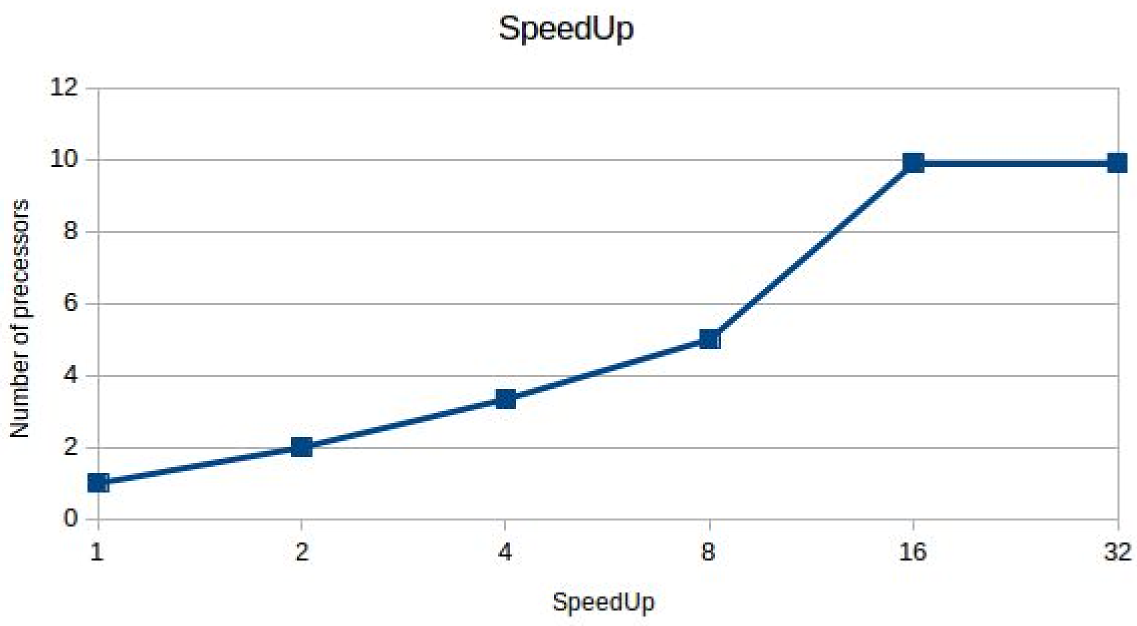
speedup = 593.772.001 / 60.012.001 = 9,894221008



execution timelines for 32 processors

Plots





In both plots we see a progress until we surpass the 16 processors. This happens because we do not need so much power. If we observe the execution timeline for 32 processors we will see that there are 20 that might not be used. In fact, if we take a look at the execution time with 8 or 16 processors, we will see that we just need 10 of them to obtain the same speedup than with 16.

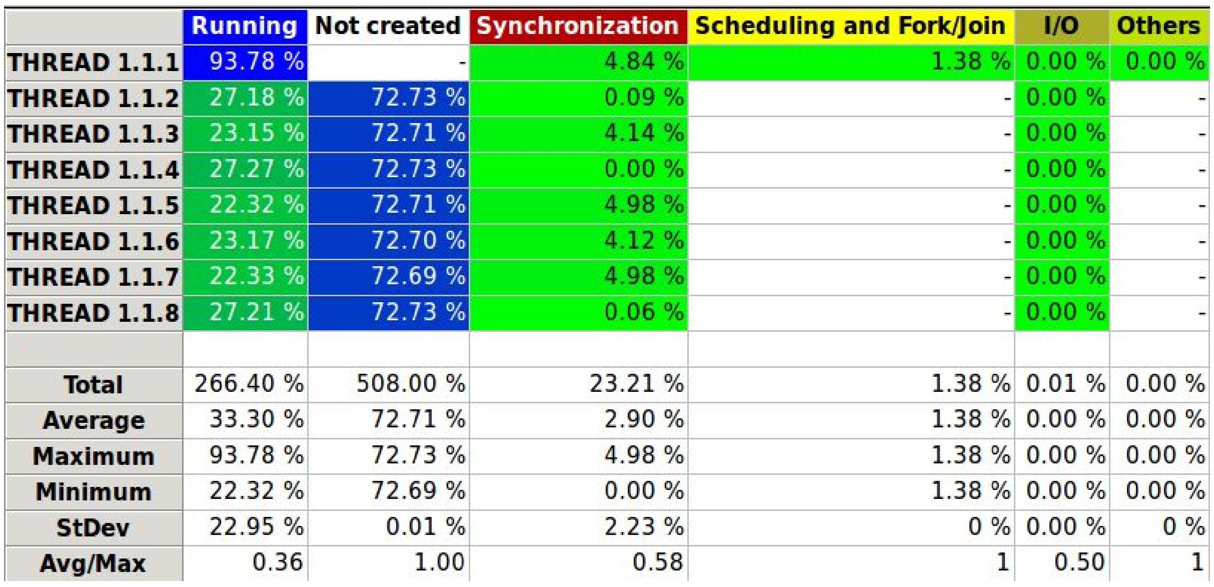
Tracing sequential and parallel executions

8. From the instrumented version of pi_seq.c and using the appropriate Paraver configuration file, obtain the value of the parallel fraction ø for this program when executed with 100.000.000 iterations, showing the steps you followed to obtain it. Clearly indicate which Paraver configuration file(s) did you use.

To obtain the value of the parallel fraction, we need the time spent on it, let's see how:

First of all, we need to compile the **pi_seq.c** (#user: make) and then execute the instrumented version with the **submit-seq-i.sh** file, this will generate a **.prv** which needed for the following steps.
Open **Paraver**, where we'll find our **pi_seq_i_100000000.prv,** double click on it to load.
Now, we generate a new timeline and load the **APP_userevents_profile.cfg** configuration, this will open a new window with the events already in the **pi_seq.c** displaying the values of each region (**SERIAL, END** and **PARALLEL**) so, only we have to take the parallel region percentage, that is: **67.17%**

9. From the instrumented version of pi_omp.c, and using the appropriate Paraver configuration file, show a profile of the % of time spent in the different OPenMP states when using 8 threads and for 100.000.000 iterations. Clearly indicate which Paraver configuration file(s) did you use and your own conclusions from that profile.



	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	93.78 %	-	4.84 %	1.38 %	0.00 %	0.00 %
THREAD 1.1.2	27.18 %	72.73 %	0.09 %	-	0.00 %	-
THREAD 1.1.3	23.15 %	72.71 %	4.14 %	-	0.00 %	-
THREAD 1.1.4	27.27 %	72.73 %	0.00 %	-	0.00 %	-
THREAD 1.1.5	22.32 %	72.71 %	4.98 %	-	0.00 %	-
THREAD 1.1.6	23.17 %	72.70 %	4.12 %	-	0.00 %	-
THREAD 1.1.7	22.33 %	72.69 %	4.98 %	-	0.00 %	-
THREAD 1.1.8	27.21 %	72.73 %	0.06 %	-	0.00 %	-
Total	266.40 %	508.00 %	23.21 %	1.38 %	0.01 %	0.00 %
Average	33.30 %	72.71 %	2.90 %	1.38 %	0.00 %	0.00 %
Maximum	93.78 %	72.73 %	4.98 %	1.38 %	0.00 %	0.00 %
Minimum	22.32 %	72.69 %	0.00 %	1.38 %	0.00 %	0.00 %
StDev	22.95 %	0.01 %	2.23 %	0 %	0.00 %	0 %
Avg/Max	0.36	1.00	0.58	1	0.50	1

To see the window above load the **pi_omp_i_10000000_8.prv** file with the **OMP_state_profile.cfg**. You'll see the same image, but remember to change the statistic to see the percentages and not the times.
We can observe that the first thread is the one who forks the other threads, does not have **Not created** time so always existed, and spends time on **Schedulng and Fork/Join** after that they all run parallel tasks. Lately they synchronize (**Synchronization**). To get the final result, they put in common the results, to create/calculate the real one.