PAR Laboratory Assignment

Lab 1: Experimental setup and tools

Semester 2 - 2018

Group par2203

Bernat Gené Ŝkrabec

Ferran Ramírez Navajón

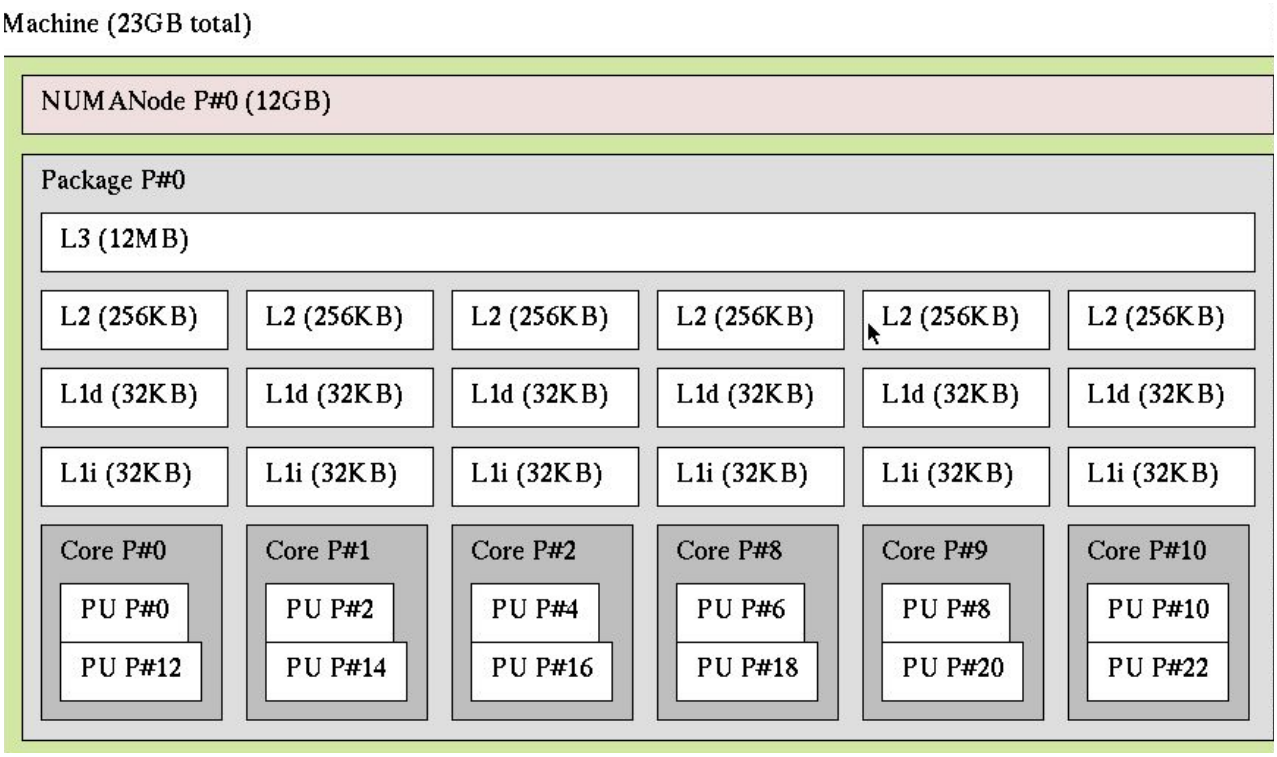
Session 1

Node Architecture and memory

1. Complete the following table with the relevant architectural characteristics of the different node types available in boada:

	boada-1 to boada-4	**boada-5**	**boada-6 to boada-8**
Number of sockets per node	2	2	2
Number of cores per socket	6	6	8
Number of threads per core	2	2	1
Maximum core frequency	2395MHz	2600MHz	1700MHz
L1-I cache size (per-core)	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)	23GB	63GB	31GB
Main memory size (per node)	12GB	31GB	16GB

2. Include in the document the architectural diagram for one of the nodes boada-1 to boada-4 as obtained when using the lstopo command.



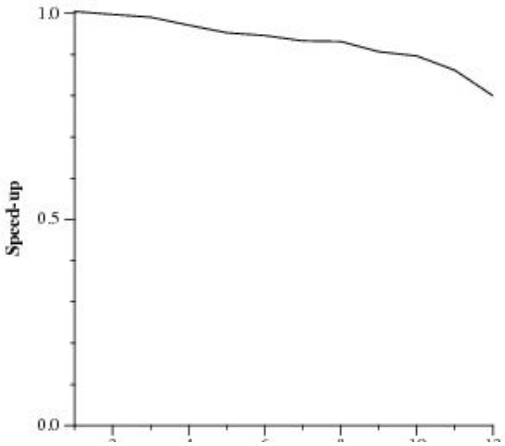
Timing sequential and parallel executions

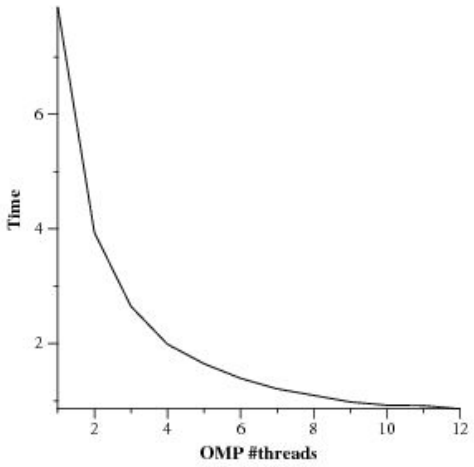
3. Plot the execution time and speed–up that is obtained when varying the number of threads (strong scalability) and problem size (weak scalability) for pi omp.c on the different node types available in boada. Reason about the results that are obtained.

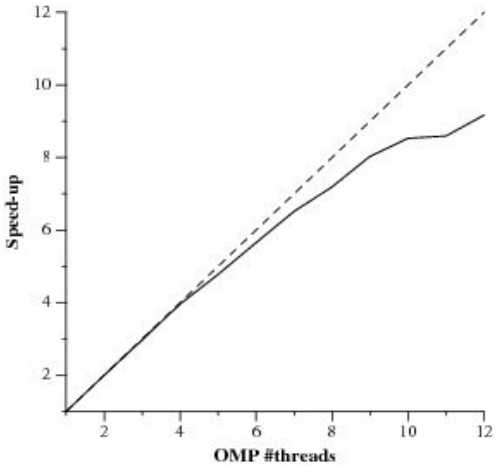
In this plot we can observe how the speed-up behaves when applying weak scalability.

This changes the number of threads and the problem size. In theory, the speed-up should be constant, but we can see that it is not the case.

That may be because there is a penalty involved in adding additional threads.







The previous two plots show us the behavior of both the elapsed time and the speed-up of our problem when applying strong-scalability.

After testing the strong scalability of pi_omp, which consists in increasing the number of threads the program uses, we can see how the speed-up is increasing by adding more threads.

However each time we add a thread, the proportional gain is reduced. In theory, we should observe that when doubling the amount of threads, the speed-up is also doubled, but in our test we see that after a certain number of threads, the benefits are diminished.

This may be due to the fact that there is a limit to how much the program can be parallelized.

Session 2

Analysis of task decompositions with Tareador

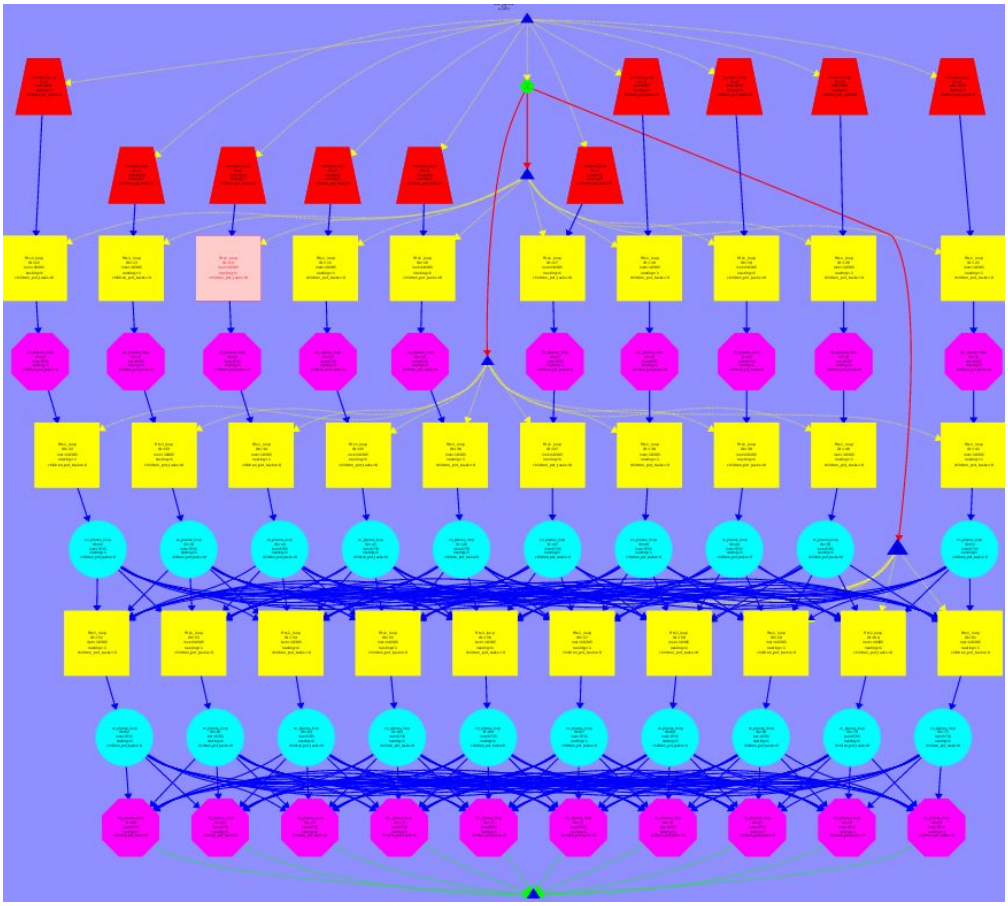
4. Include the relevant(s) part(s) of the code to show the new task definition(s) in v4 of 3dfft_seq.c. Capture the final task dependence graph that has been obtained after version v4.

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

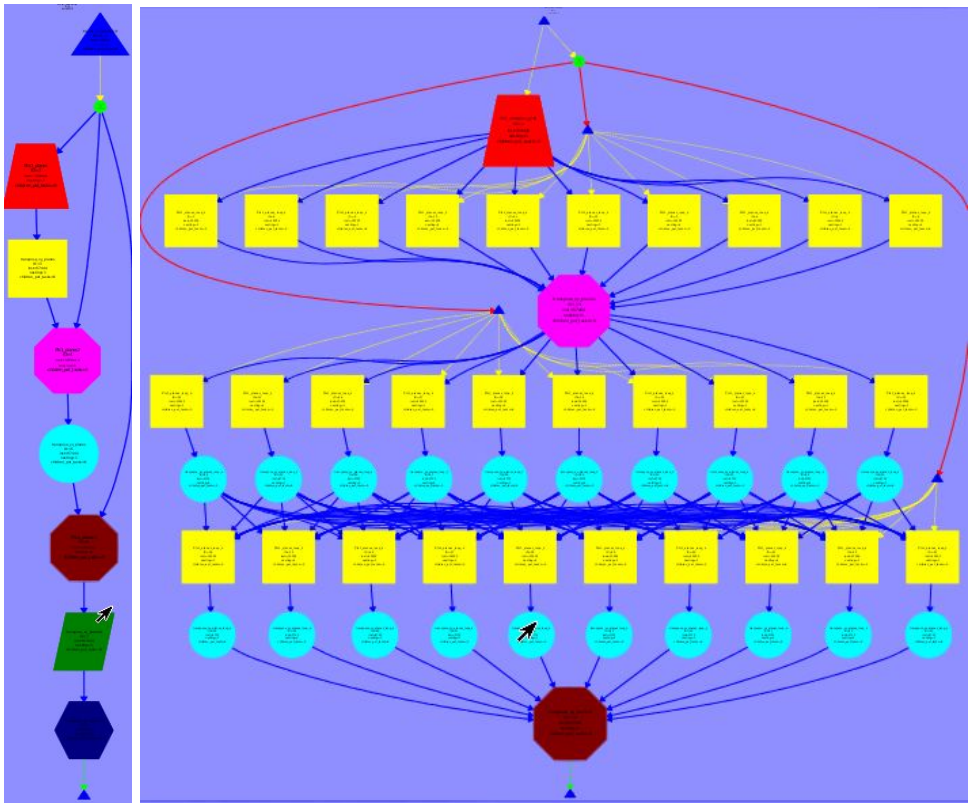
Similar definitions have been made to all other intermediate functions.

This is the graph obtained with version v4:



5. 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	**T_1**	**T_∞**	**Parallelism**
seq	593772	593758	1.000020
v1	593772	593758	1.000020
v2	593772	315523	1.881866
v3	593772	108878	5.453554
v4	593772	59622	9.958941



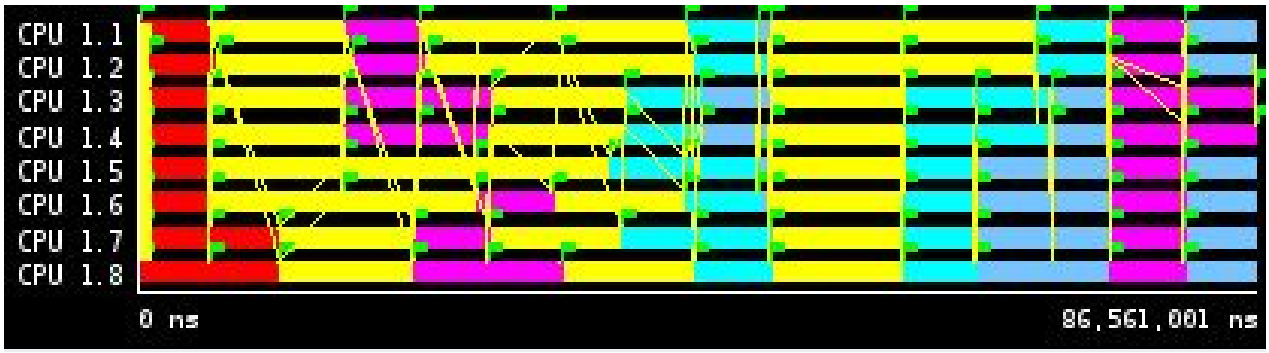
(v1) (v2)

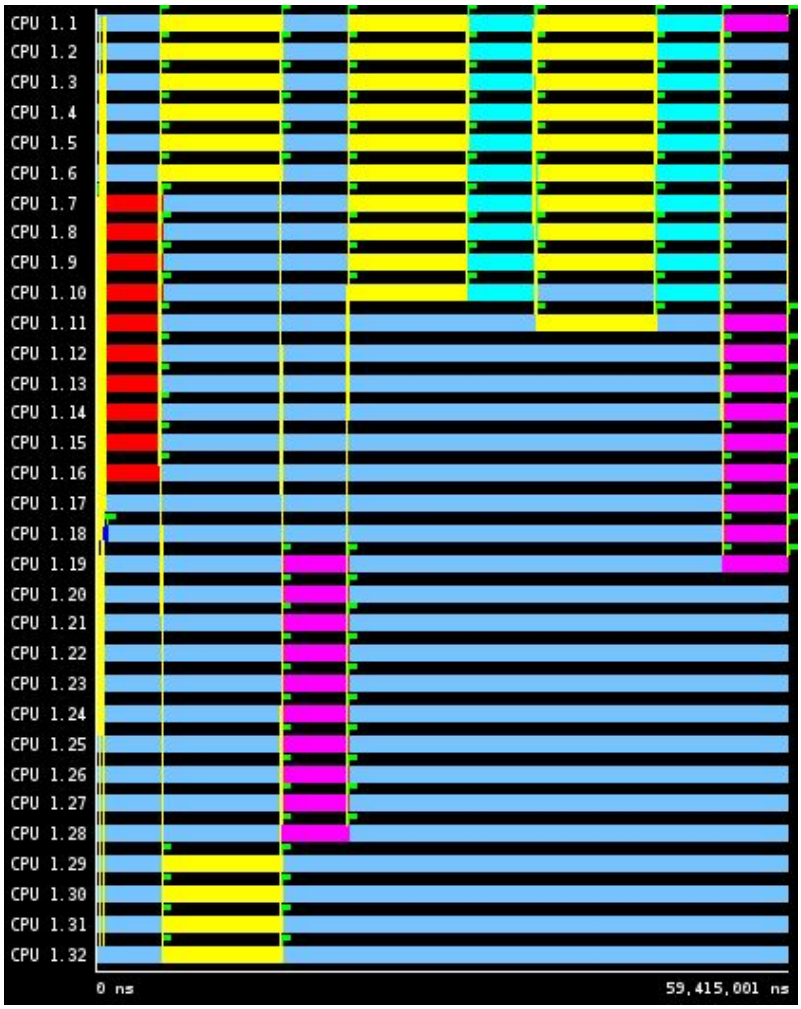
The first version decomposes our program in smaller tasks, but given the dependencies between these tasks, it is not possible to parallelize anything. This dependencies can be observed in the dependence graph(v1).

With the second version, we have divided some of the previous tasks into even smaller ones that do not have data dependencies between them, this can be seen in the second dependence graph (v2). As a result we observe a significant improvement.

We can see that each version allows for a higher degree of parallelism. We progressively increase the granularity of the various tasks and in doing so, more of our program can be parallelized.

6. 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.





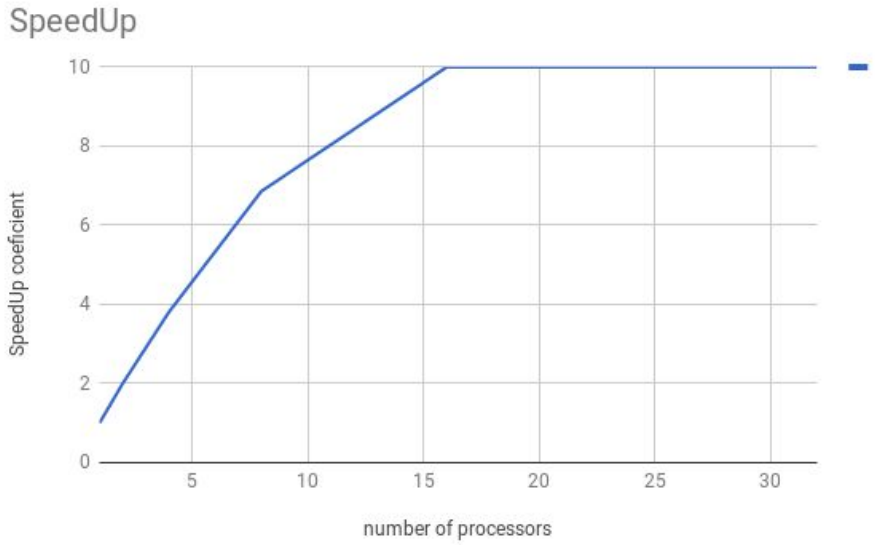
Comparison of the activity of every core when simulating with 8 vs 32 processors.

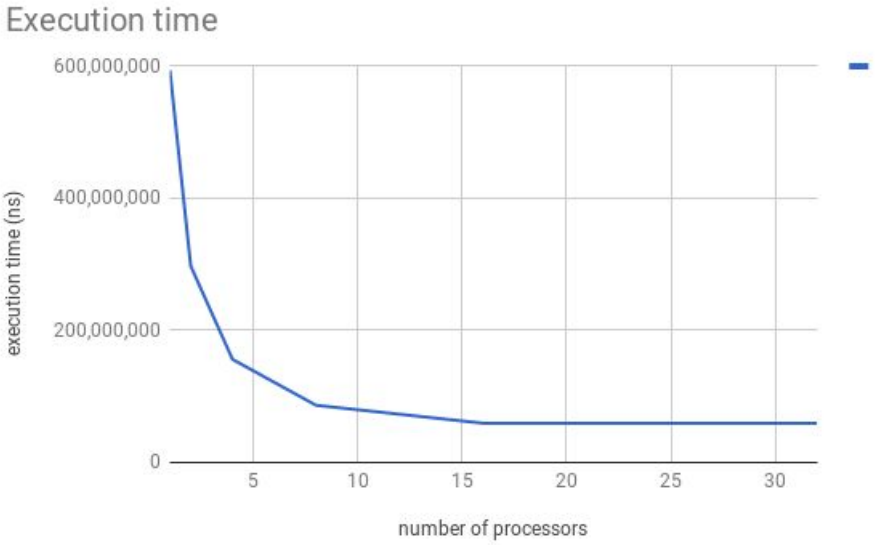
The sections in grey-blue indicate that a processor is idle.

Both the plot of the execution time and speedup show that we get a better result with more processors, but with diminished returns. After a certain point, with more than 16 processors, we stop seeing any improvement.

This is because the way our program is written, it can only be parallelized to a certain point, after that it does not matter how many processors we have because there are not enough tasks that could be run in parallel, not even in theory.

The extra processors just run idle most of the time. This can be observed in the timelines that we have added, the sections in grey-blue indicate that a CPU is idle in that part of the timeline.

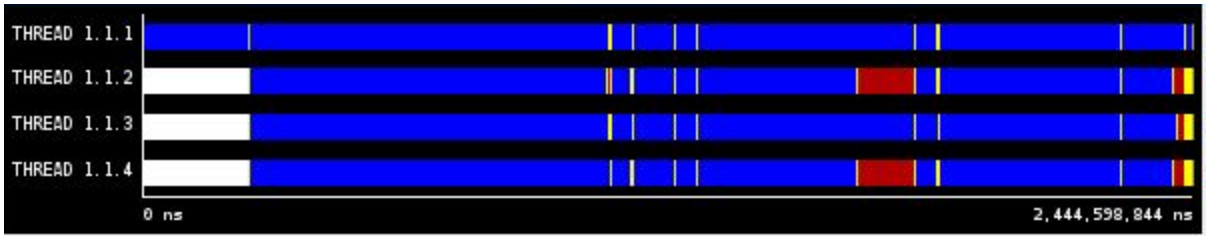




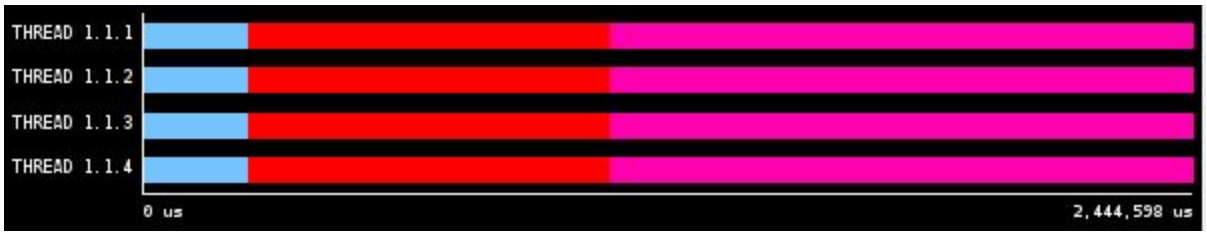
Session 3

Tracing the execution of parallel programs

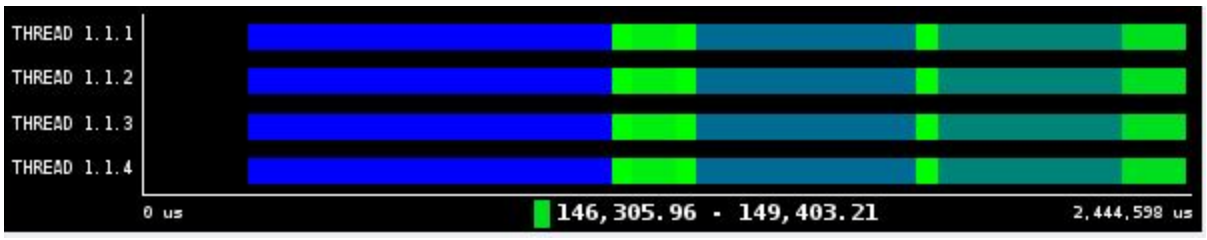
7. From the analysis with Paraver that you have done for the complete parallelization of 3dfft omp.c, explain how have you computed the value for φ, the parallel fraction of the application. Please, include any Paraver timeline that may help to understand how you have performed the computation of φ.



Timeline with Thread State (Default) configuration



Timeline with OMP_parallel_function configuration



Timeline with OMP_worksharings configuration

The above timelines show the behaviour of our program. It can be observed that there is an initial non-parallelizable part which is executed only in one thread. With the tools provided by Paraver, we have determined that this initial part runs for 243,041,702 ns. The rest is almost equally shared among the other processors, this will be the part that our program is running in parallel.

Φ is the fraction of time our program runs in parallel. Therefore, Φ is equal to the execution time of the parallelizable part of the program, executed sequentially, divided by T1; the total time of the program when run in a single processor. This means we can compute it as follows:

Total execution time minus the initial sequential part times four(the number of threads), divided by T1.

T $_{par}$ = (2,444,598,844 ns - 243,041,702 ns)*4 = **8,806,228,568 ns.**

T $_1$ = (8,806,228,568 + 243,041,702) = **9,049,270,270 ns**

Φ = (8,806,228,568 / 9,049,270,270) = **0.973142**

8. Show and comment the profile of the % of time spent in the different OpenMP states for the complete parallelization of 3dfft omp.c on 4 threads.

	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	2,430,234,736 ns	-	10,444,847 ns	3,880,132 ns	36,739 ns	2,390 ns
THREAD 1.1.2	2,003,666,568 ns	246,630,305 ns	172,992,737 ns	21,296,391 ns	12,843 ns	-
THREAD 1.1.3	2,150,568,677 ns	246,621,335 ns	25,541,730 ns	21,856,327 ns	10,775 ns	-
THREAD 1.1.4	2,007,717,757 ns	246,616,423 ns	168,912,499 ns	21,341,587 ns	10,578 ns	-
Total	8,592,187,738 ns	739,868,063 ns	377,891,813 ns	68,374,437 ns	70,935 ns	2,390 ns
Average	2,148,046,934.50 ns	246,622,687.67 ns	94,472,953.25 ns	17,093,609.25 ns	17,733.75 ns	2,390 ns
Maximum	2,430,234,736 ns	246,630,305 ns	172,992,737 ns	21,856,327 ns	36,739 ns	2,390 ns
Minimum	2,003,666,568 ns	246,616,423 ns	10,444,847 ns	3,880,132 ns	10,578 ns	2,390 ns
StDev	173,330,812.89 ns	5,747.45 ns	76,679,264.78 ns	7,631,974.69 ns	11,008.50 ns	0 ns
Avg/Max	0.88	1.00	0.55	0.78	0.48	1

When the work done in different threads is over and is needed for other tasks, it may happen that the aforementioned threads are slightly out of sync, therefore we need to spent time to resync them.

A similar thing happens with the time spent for scheduling and Fork/Join. The work done in different threads may not be useful unless combined, meaning we need to spend time to “join”. We also need time to plan the needed forks, that is the step required before a task can be run in parallel. All this tasks require some time, even if they do not inherently belong to our program.

As it can be observed by the table, the time spent in synchronization, scheduling and Fork/Join is shared among all threads, but not equally.