# PAR Laboratory Assignment Lab 1:
# *Experimental setup and tools*

Nil Quera and Sergio Arcos

**PAR4104**

Fall *2019/2020*

# Index

# 1. Introduction

In this deliverable, we are going to use several analysis tools in order to learn how parallelism works. We will use boada, a multiprocessor server located at the Computer Architecture Department, here at UPC. To actually get all the information we need, we will use specialised data analysis software: Paraver and Tareador.

The main objective is to see how tasks are distributed when we have several processors working and what's the difference between sequential and parallel programs. Hence, we will compare different versions of the same program (3dfft) to see how the execution time and task distribution change.

# 2. Node architecture and memory

To start, we must describe the architecture of the boada server, since every analysis made hereafter will take place in such computers.
Boada is a multiprocessor server with 8 nodes, each based in a different processor generation. As described in the table below, we have three set of nodes that share common features.

| | 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 | 2.4GHz | 2.6GHz | 1.7GHz |
| L1-I cache size (per-core) | 32KB | 32KB | 32KB |
| L1-D cache size (per-core) | 32KB | 32KB | 32KB |
| L2 cache size (per-core) | 256KB | 256KB | 256KB |
| Last-level cache size (per-socket) | 12MB | 15MB | 20MB |
| Main memory size (per socket) | 12GB | 31GB | 16GB |
| Main memory size (per node) | 24GB | 62GB | 32GB |

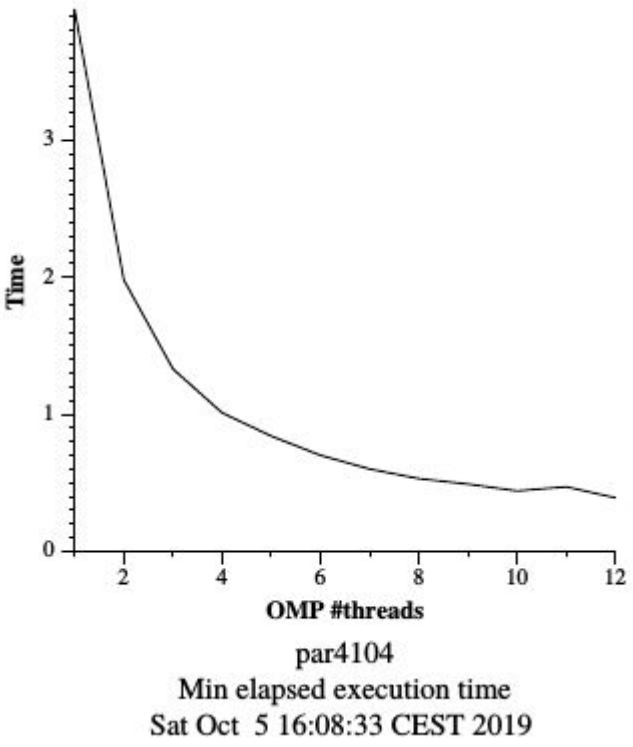
The reason we have eight separate nodes instead of only one is that we want to have the resources spread so that many users can make use of them at the same time. From boada-1, tasks can be queued on any of the other nodes. In this way, they will run in a shared environment. To ensure independent execution and real representations, boada-1 has its resources isolated. Thus, we can also run processes in boada-1 to make sure the results are realistic.

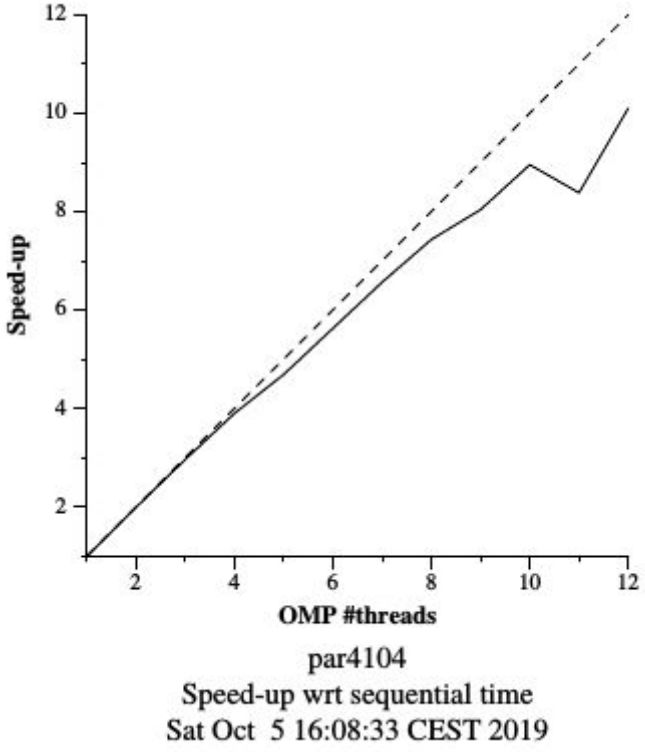
Next, we present a basic architectural diagram of boada-1 (which is identical to boada-2 to boada-4). Click on the link below to see the full size diagram.
Architectural Diagram

# 3. Strong vs. weak scalability

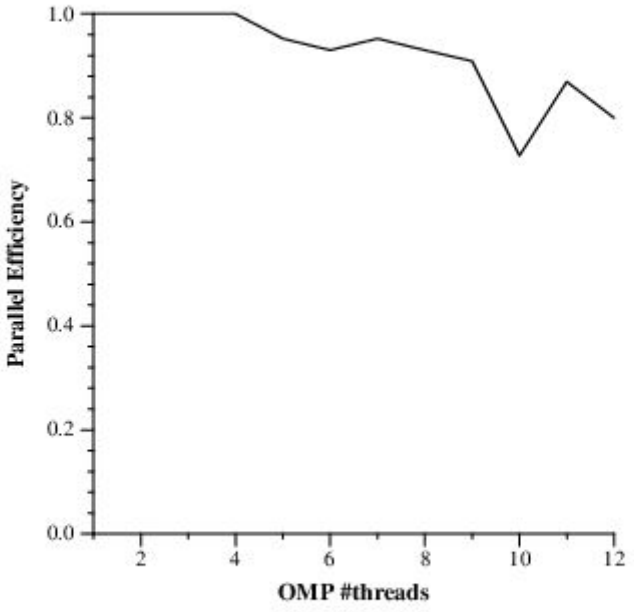
When trying to increase a program's efficiency, we must differentiate between two different approaches. First, we can increase the number of threads while fixing the problem size. This gives place to the concept of strong scalability. How large will the scalability value be depends on the capacity of the program and the programmer to reduce the execution time by parallelising the code. On the other side, if the problem size is increased at the same time as the number of threads, we would be talking about weak scalability. The main goal here would be to make the program deal with bigger input while keeping its execution time as low as possible.

Below we can see an example of strong scalability. On the left plot, execution time decreases when we put more cores into work. While time is shortened very quickly at the beginning, there's a point from which increasing threads makes no reasonable decrease on time. What's more, we can see that execution takes longer with eleven threads than with ten. The plot on the right shows the same effect. The speed-up seems to grow slowlier with the more threads we have.




Executing the program while increasing the input size, the execution time keeps at the same level (0.4 seconds). However, it starts to lose efficiency when we have more processors. This probably happens because of the overhead of creating more tasks. Consequently, the time slightly increases with more processors. At the same time, the speedup is also lower. We can see how the efficiency decreases in the next plot obtained from the execution of the test program.




par4104
Parallel Efficiency w.r.t. one thread (weak scaling)
Sun Oct 6 17:40:06 CEST 2019

# 4. Analysis of task decompositions for 3DFFT

Here we will use Tareador to have a general view of the task decomposition that will be implemented in every version. Also, we will use Paraver to see more in-depth information such as timestamps and task distribution between processors.

We start with a simple and sequential program. The objective of this part is to modify some aspects of the code, slowly transforming the sequential code into a parallel one. At every version we make, we will analyze the program with Paraver and Tareador in order to compare it to the previous version. It is important to say that Tareador does not actually parallelize the code, but runs a simulation of what we would get in such case.

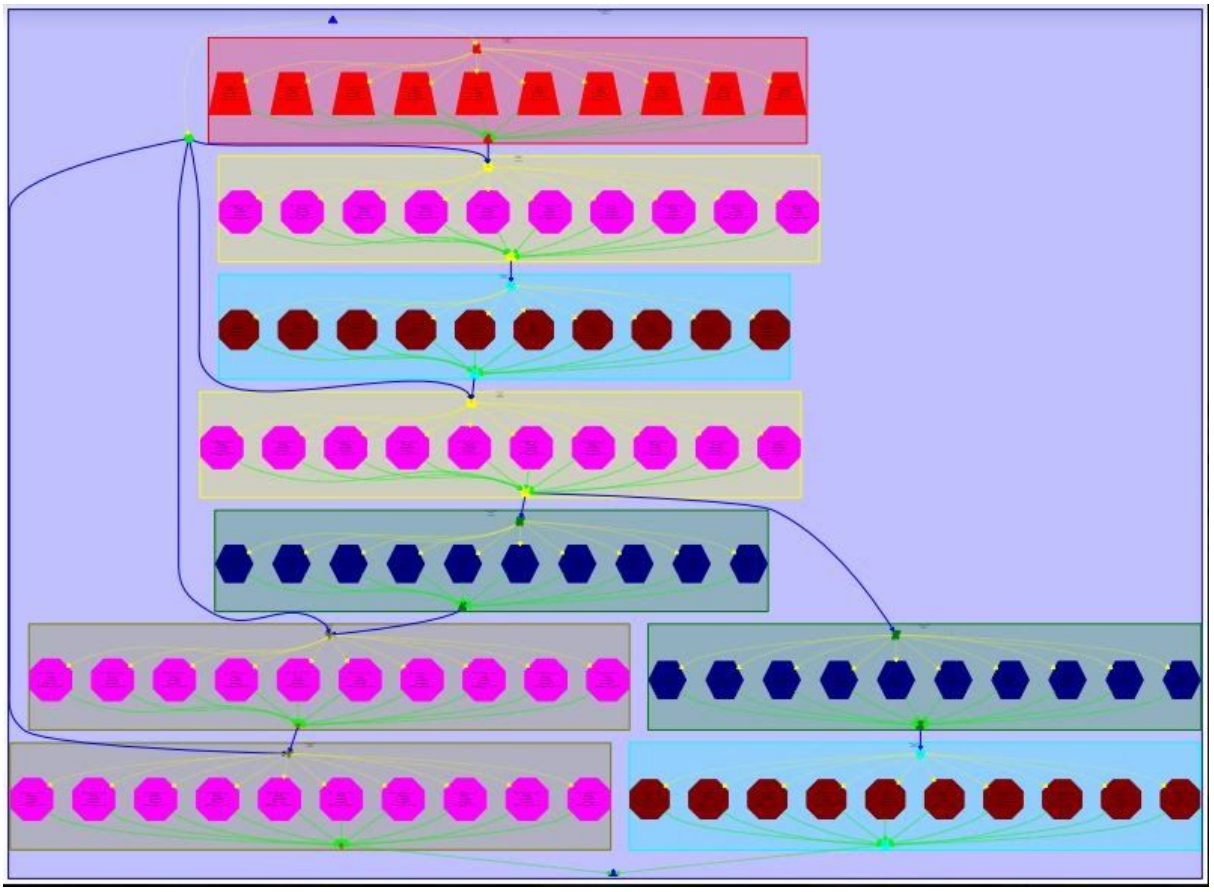
On the first version (*v1*), we encapsulated each function call in a different task. Tareador, however, found dependencies between tasks, not being able to make any more parallelisation. That is why, as can be seen in the table, *seq* and *v1* have the same execution time. In *v2* we declared every k loop in ffts1_planes as a single task. In this case, they don't have dependencies between each other, so they can be executed in parallel. In *v3* we did the same with the functions transpose_xy_planes and transpose_zx_planes.

| Version | T1 | T∞ | Parallelism |
|---|---|---|---|
| seq | 639.76 ms | 639.71 ms | 1 |
| v1 | 639.76 ms | 639.71 ms | 1 |
| v2 | 639.76 ms | 361.44 ms | 1.8 |
| v3 | 639.78 ms | 154.94 ms | 4.1 |

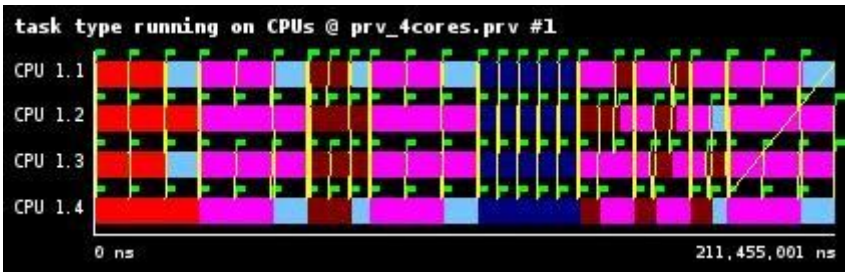
From version *seq* to *v3* we made little changes to the code, but the last two versions are actually where things get interesting. At this point (*v4*) we already have a parallel program, but we want to see if having too much granularity makes the execution time faster or not.

## Version v4

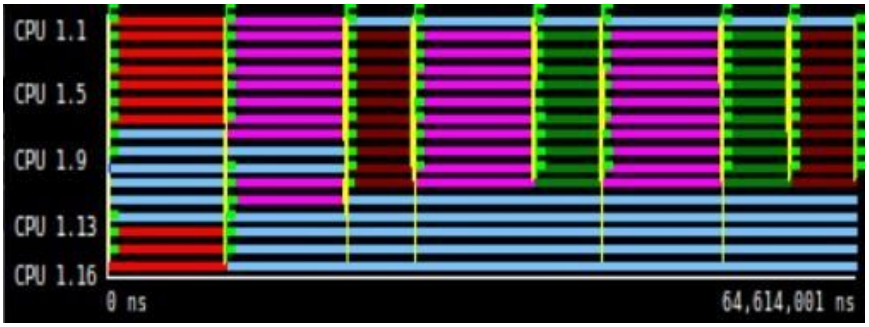
On version 4, we redefined the function init_complex_grid with fine-grained tasks. Again, we achieved a better execution time. In this case, we tried executing the code with different number of processors.
In the following diagram, we can see how much parallelisation we have achieved. However, notice that there are inevitable dependencies between tasks. After that we have attached pictures of the execution timeline with 4, 16 and 32 processors.



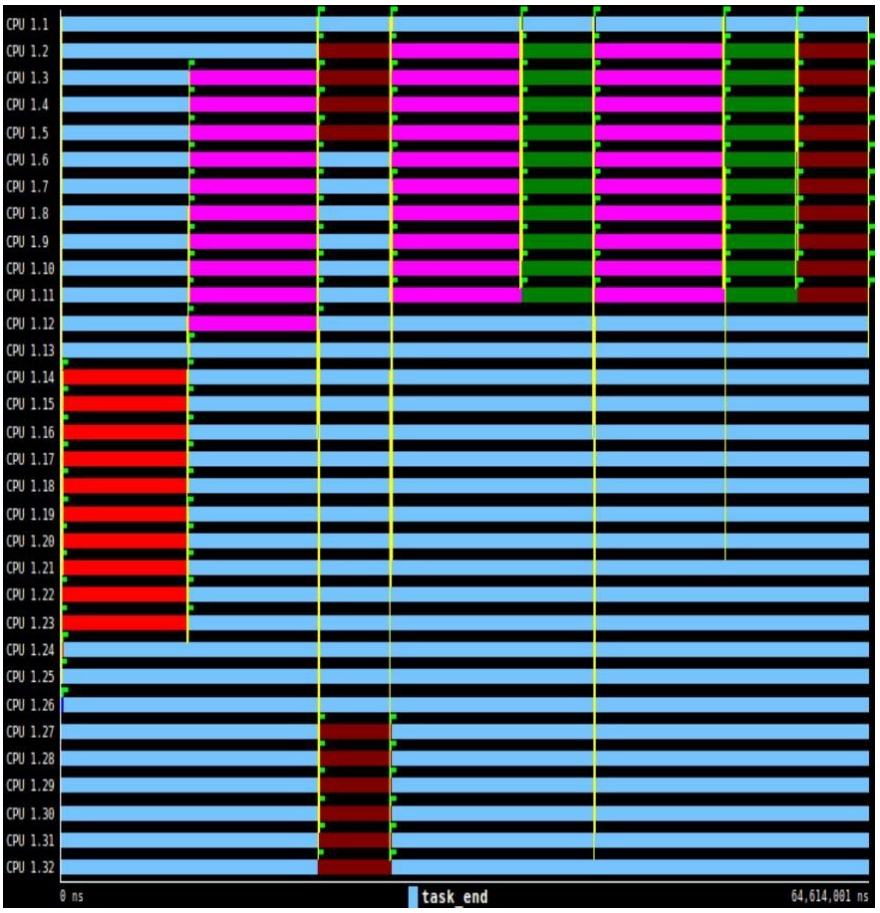
*Dependency graph of v4*




*Task time  on 4 CPUs*




T*ask time on 16 CPUs*

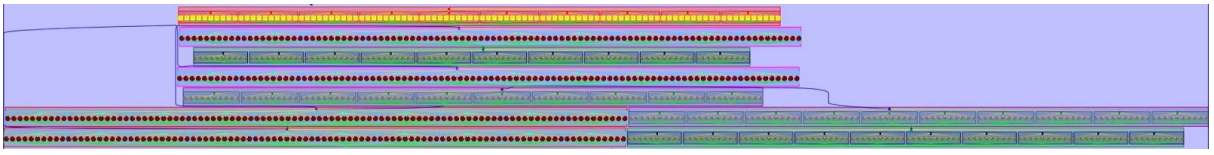



*Task time on 32 CPUs*

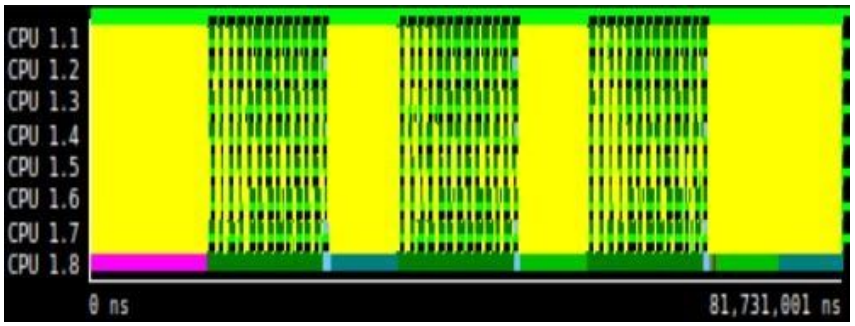
As we can see, we get to a point where it does not make much sense to have more processors. This happens because the tasks cannot be more decomposed.

# Version v5

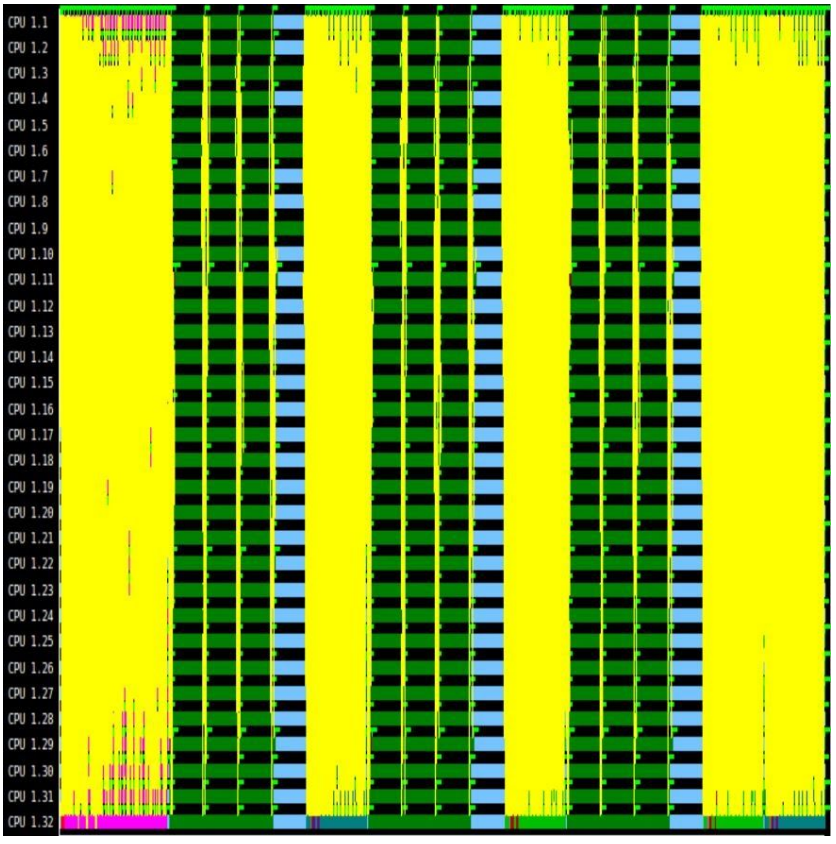
Finally, in version v5 we will see whether it is worth to decompose as much as possible in different tasks. To accomplish this, we changed the code to create even finer-grained tasks. The following task dependency graph shows the huge amount of different tasks. Of course, this will allow us to take advantage of the use of many processors. But the problem will be the overhead of the cpu to generate and keep interrelated each and every task.



*Dependency graph of v5*




*Task time on 8 CPUs*




*Task time on 32 CPUs*

As we can see, the timeline is pretty similar with 8 and 32 CPUs. Since there are many tasks, increasing the number of processors just redistributes them and runs them in parallel. Notice here the big amount of yellow lines that go from task to task. Every single line is a dependency between two tasks.

| Version | T1 | T2 | T4 | T8 | T16 | T32 | T∞ | Parallelism *T1/T∞* |
|---|---|---|---|---|---|---|---|---|
| v4 | 742.92 ms | 371.69 ms | 211.46 ms | 137.1 ms | 74.72 ms | 63.36 ms | 63.36 ms | 11.72 |
| v5 | 742.92 ms | 371.74 ms | 186.7 ms | 95.85 ms | 50.34 ms | 28.07 ms | 8.77 ms | 84.71 |

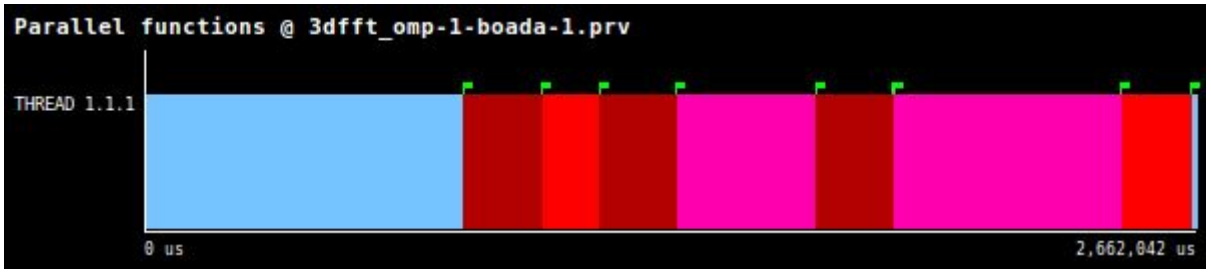
As a conclusion, we can see an enormous difference between version v4 and v5. There is a point where there is too much fine-grain tasks to do, limiting the potential scalability (due to overheads). All those tasks require a lot of CPUs to be executed. Hence, v5 would only make sense when we really want to reduce a lot the time and we have enough hardware to run such parallelisation which usually means that the economic cost will be high.

# 5. Understanding the parallel execution of 3DFFT

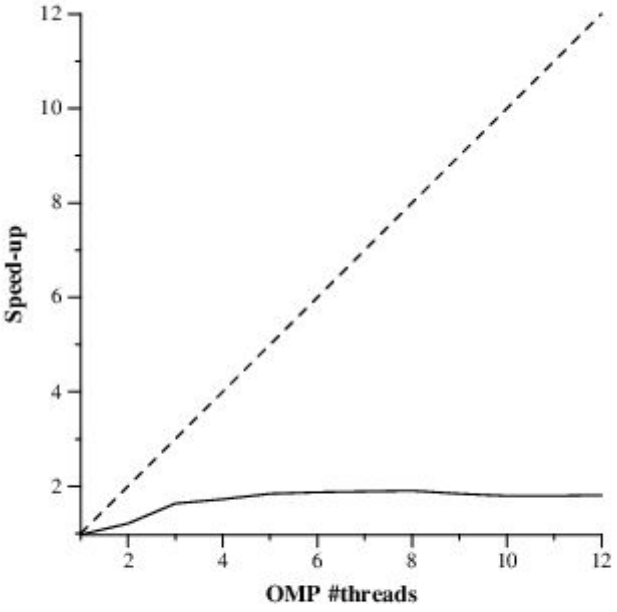
OpenMP is a software that provides a set of tools to produce real parallelisation. In this section, the 3dfft code will include several openMP instructions. To analyse the results, we will use Paraver, a program that offers full statistical measures and detailed timelines. As we did in the previous section, we will make different improvements on the parallelised code and comment any change that occurs.

## Initial version

First of all, it's important to know the percentage of the program that can be parallelised. In the following picture obtained in Paraver after the execution of 3dfft on 1 thread, light blue shows the sequential part of the program. Everything else is potentially parallelizable. From now on we recommend you take a look at the table below for a summary of the statistics obtained.



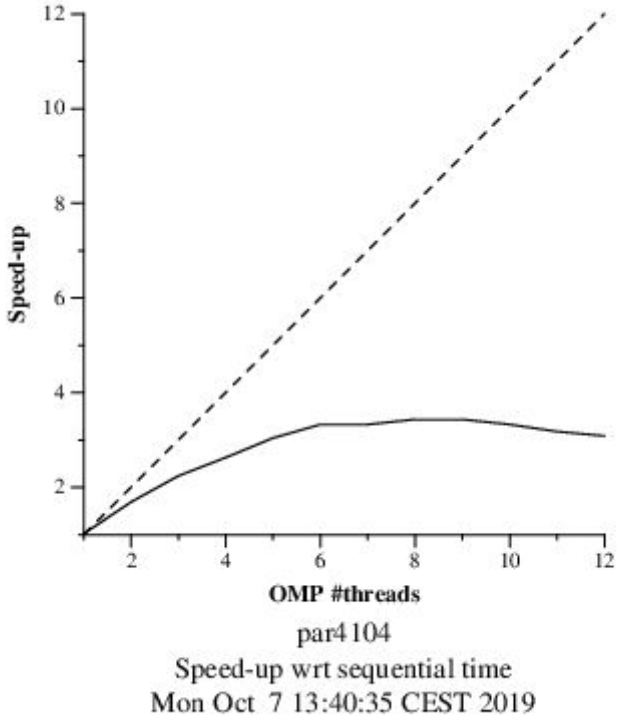

Now we proceed to analyse the strong scalability of the first version of the program. As we can see in the plot below, it is very noticeable that the speed-up remains almost constant independently of the number of threads. This clearly means that our first version is still poorly parallelised. The ideal speed up we would obtain with infinite processors is 3.24. We see that we are not that far from this ideal speedup. Overall, this happens because the parallelised fraction of the program is still very small.



## New version with improved $\phi$

A program parallelisation is poorly scalable when there is a time consuming sequential part. So, we must strive for reducing the sequential part. In this section, we have parallelised what seemed to be the longest function of our program. We will see the effects on the execution time.

Taking a look at the same Paraver timeline as before with the improved version of the program, we now see that the parallel fraction ($\phi$) is of 90%. Now we have parallelised nearly all the program, so we should see improvements on the scalability. Even though the strong scalability is far from being linear, it is still better than with the initial version. However, there's still a big gap between the ideal S∞ and the scalability achieved with 8 processors. We will see in the next section that this gap can be reduced by minimizing the parallelisation overheads.




## Version with reduced parallelisation overheads

In this final version of our program, we have reduced the granularity of tasks. Now, instead of defining a task as every execution of the k loop, we will define the whole k loop as a single task. This seems contradictory, since we are getting less parallelism. However, doing this we reduce the overhead of creating k different tasks. Now, we will see how this changes effect on our program execution time.

| **Version** | $\phi$ | $S_\infty$ | $T_1$ | $T_8$ | $S_8$ |
|---|---|---|---|---|---|
| initial version in 3dfft_omp.c | 69% | 3.2 | 2.32 s | 1.39 s | 1.67 |
| new version with improved $\phi$ | 90% | 10 | 2.38 s | 0.82 s | 2.89 |
| final version with reduced parallelisation overheads | 92% | 12.9 | 3.31 s | 0.61 s | 5.41 |