Laboratory Assignment 1

PAR - Q2 2019/2020

Marc Monfort Grau
Albert Mercadé Plasencia
Grup par2302

March 4th 2020

INDEX

Node architecture and memory

Describe the architecture of the boada server. To accompany your description, you should refer to the following table summarising the relevant architectural characteristics of the different node types available:

The Boada server, it's used to put in practice parallelism. It's composed of multiple nodes with a total of 3 kinds of architectures. Boada-1 is the only interactive node. To use the rest of the nodes you can submit execution jobs via a queuing system. The following table shows the specifications of each type:

	boada-1 to boada-4	boada-5	boada-6 to boada-8
Number of sockets per node	2	2	2
Number of cores per socket	6	6	8
Number of threads per core	2	2	1
Maximum core frequency	2395 MHz	2600 MHz	1700 MHz
L1-I cache size (per-core)	32 KB	32 KB	32 KB
L1-D cache size (per-core)	32 KB	32 KB	32 KB
L2 cache size (per-core)	256 KB	256 KB	256 KB
Last-level cache size (per-socket)	12 MB	15 MB	20 MB
Main memory size (per socket)	12 GB	31 GB	16 GB
Main memory size (per node)	23 GB	63 GB	31 GB

Table 1: Boada architectural characteristics.

Also include in the description the architectural diagram for one of the nodes boada-1 to boada-4 as obtained when using the lstopo command. Appropriately comment whatever you consider appropriate.

Figure 1: Boada - 3 architecture diagram.

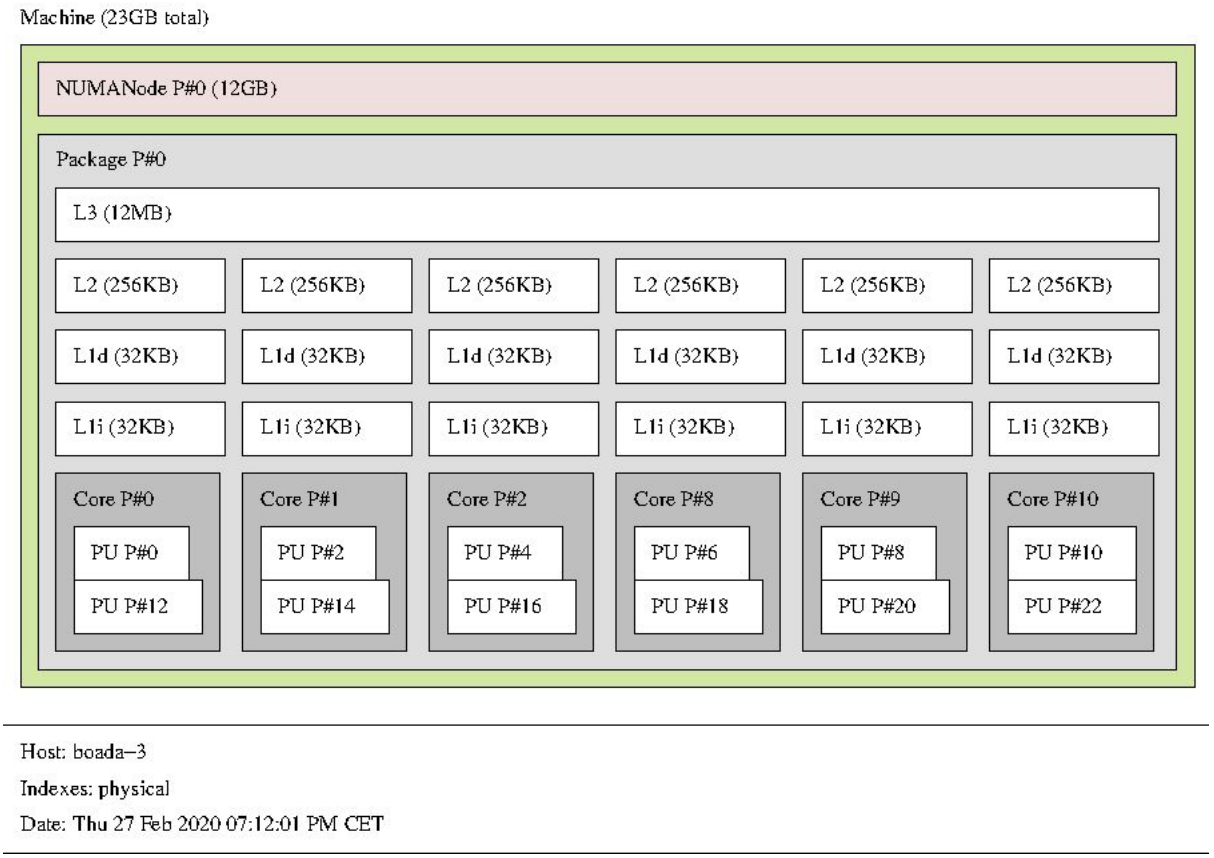

Figure 2: Amplified socket 1 (boada -3) architecture diagram

We can see the architecture diagram of boada-3 (the same for boada 1, 2, and 4). It has 3 levels of cache: L1 divided in data and instruction memory (each with 32 KB), L2 (256 KB), and a last level cache L3 (12 MB). L3 is shared between the 6 cores of each socket.

Strong vs weak scalability

Briefly explain what strong and weak scalability refer to. Exemplify your explanation using the execution time and speed–up plots that you obtained for pi omp.c. Reason about the results obtained.

Strong scalability refers to increasing the number of threads used but leaving the problem size unchanged, thus reducing the execution time of the program.

On the other hand, weak scalability consists of having the number of threads used proportional to the size of the problem. Hence, the problem size per processor stays constant and we can solve larger problems.

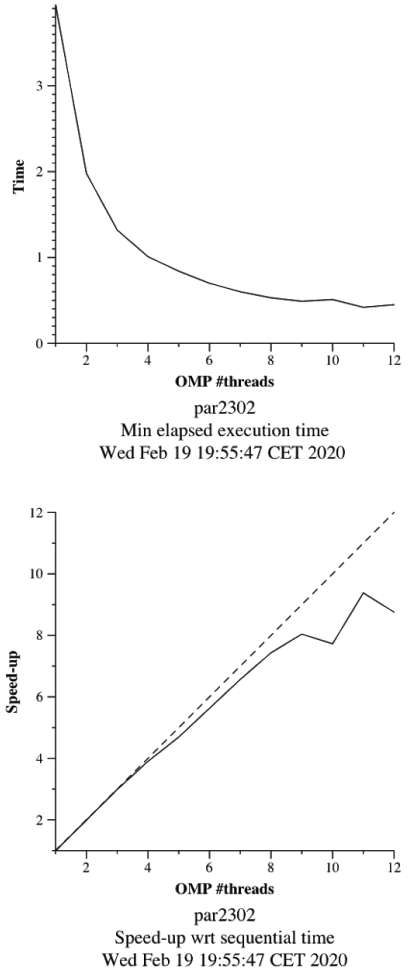

Figure 3: Strong scalability time and speed-up for pi_omp.c

On the *Time v OMP #threads* chart of Figure 3, we can clearly appreciate the strong scalability of pi_omp.c because as we increase the number of threads, we observe an almost proportional reduction in execution time. Furthermore, we can see this effect more clearly in the *Speed-up v OMP #threads* chart in Figure 3. The chart follows the 45º line almost

perfectly up to 4 threads, after that and up until 8 threads we see the curve start to moderately move away from the identity line and after 8 threads the speed-up is visibly no longer proportional to the number of threads used. With the help of these two graphs we evidently see the effects of strong scalability and also its limits.

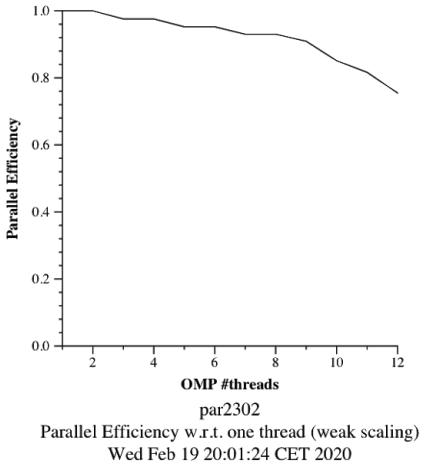

Figure 4: Weak scalability parallel efficiency for pi_omp.c

In this second execution, we tested the weak scalability of pi_omp.c, that is increasing the problem size proportionally to the number of threads used. As such, we should theoretically observe that the parallel efficiency stays at 1.0, because the problem size per processor is constant and so is the execution time. However, we can appreciate on Figure 4 that wasn't the case for this execution. After increasing the number of cores employed to 2, we see a continuous decrease in parallel efficiency as we keep increasing the threads used, sinking below 0.8 for 12 threads. Hence, we have seen that pi_omp.c isn't fully weak scalable as the execution time and therefore the parallel efficiency haven't stayed constant throughout.

Analysis of task decompositions for 3DFFT

In this part of the report you should summarise the main conclusions from the analysis of task decompositions for the 3DFFT program. Backup your conclusions with the following table properly filled in with the information obtained in the laboratory session for the initial and different versions generated for 3dfft tar.c, briefly commenting the evolution of the metrics.

The 3DFFT program shows us the benefits in terms of computation time and parallelism that we obtain with a finer-grained task decomposition. In the following table, we see the speed-up that we achieve for each version, and how it increase in the measure that we increase the decomposition in task of the code.

Version	T1 (ms)	T∞ (ms)	**Parallelism**
Seq	639,780	639,780	1.0
V1	639,780	639,707	1.0001
V2	639,780	361,190	1.7713
V3	639,780	154,354	4.1449
V4	639,780	64,018	9.9938
V5	639,780	13,781	46.4247

Table 2: 3DFFT task decompositions.

For versions v4 and v5 of 3dfft_tar.c perform an analysis of the potential strong scalability that is expected. For that include a plot with the execution time and/or speed-up when using 1, 2, 4, 8, 16 and 32 processors, as reported by the simulation module inside Tareador. You should also include the relevant(s) part(s) of the code that help the reader to understand why v5 is able to scale to a higher number of processors compared to v4, capturing the task dependence graphs that are obtained with Tareador.

```
for (k = 0; k < N; k++) {
    tareador_start_task("...v4...");        V4 CODE
    for (j = 0; j < N; j++) {
        tareador_start_task("...v5...");        V5 CODE
        for (i = 0; i < N; i++) {
            [...] //code
        }
        tareador_start_task("...v5...");        V5 CODE
    }
    tareador_start_task("...v4...");        V4 CODE
}
```

Code 1: Code of the implementation in V4 and V5.

In Code 1 we see the differences between the version 4 and version 5 code for 3DFFT.c. In version 5 there is finer-grained task decomposition, since the call to generate tasks it's inside the second loop, while in version 4 the call it's just inside the first loop.

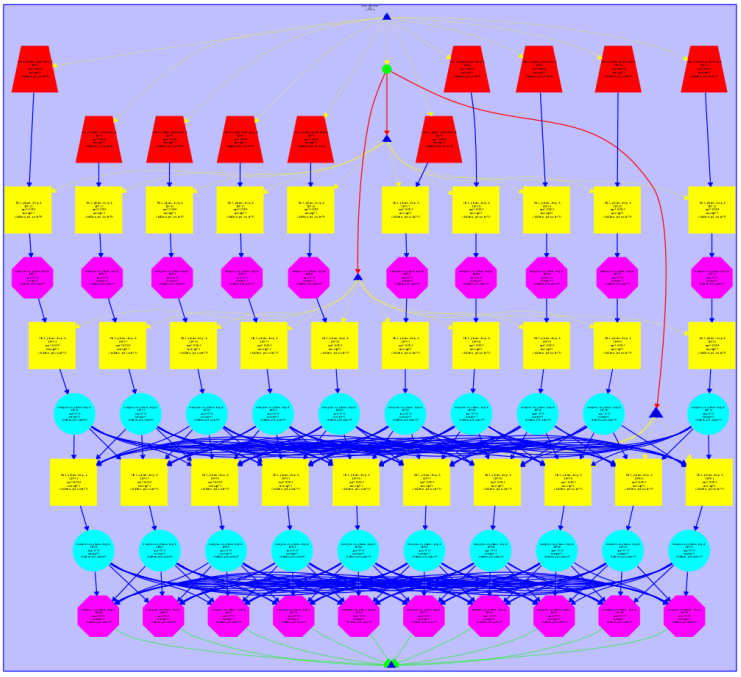

Figure 5: V4 Task dependence graph.

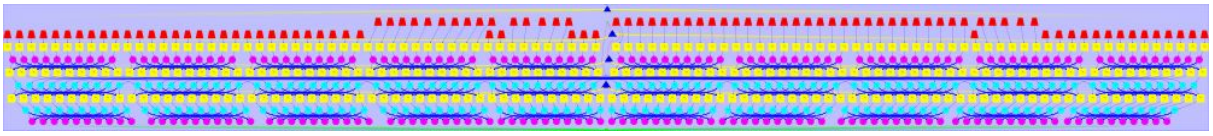

Figure 6: V5 Task dependence graph.

In Figure 5 and Figure 6 we see the task dependence graph from the version 4 and 5 respectively. We can appreciate how the V5 graph is much wider than V4, meaning we have more possibilities to parallelise its execution.

Processors	1	2	4	8	16	32
time (ms)	639,780	320,310	165,389	91,496	64,018	64,018

Table 3: V4 Time obtain for each number of processors.

Processors	1	2	4	8	16	32	64	128
time (ms)	639,780	320,091	160,092	80,179	40,264	20,592	10,978	8,155

Table 4: V5 Time obtain for each number of processors.

These two tables show the execution time of both version, while using from 1 to 128 processors. In the version 4 we stop at 32 processors because the execution time it's not improving, and won't improve anymore. In version 5 we keep testing the code until 128 processors (the maximum allowed in Tareador) without a stop improving.

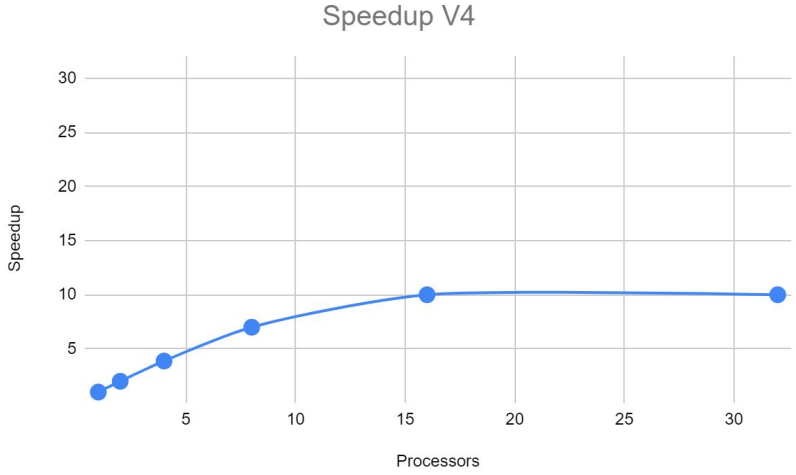

Figure 7: V4 Speed-up

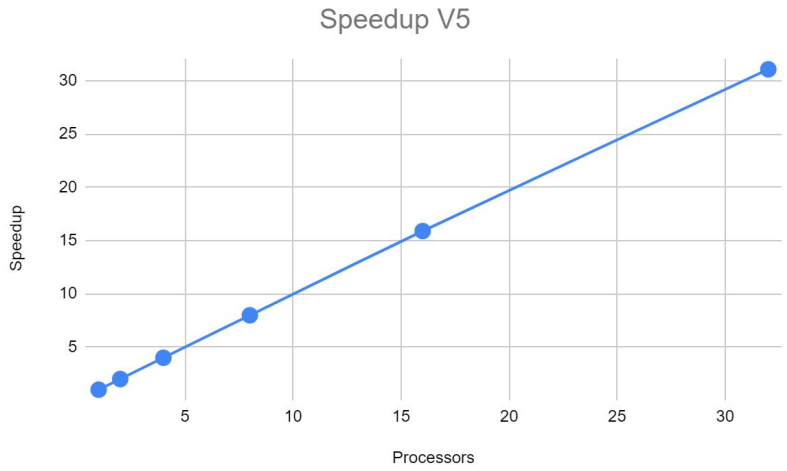

Figure 8: V5 Speed-up

Figure 7 and Figure 8 show the speed-up of both version. Again, it's seen how version 5 allow more parallelism almost doubling the speed-up while doubling the number of processors. In version 4 we see a maximum speed-up of 10 with 15 processors.

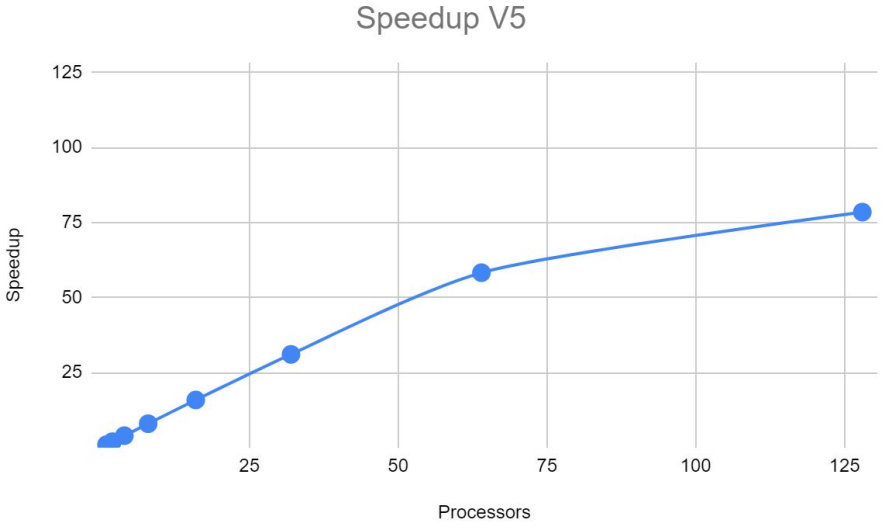

Figure 9: V5 Speed-up (range from 1 to 128)

However, this increase of speed-up in V5 it's not infinitely doubling with the number of processors. As we see in Figure 9, the speed-up slowly shrinking after 64 processors.

Understanding the parallel execution of 3DFFT

In this final section of your report you should comment about how did you observed with Paraver the parallel performance evolution for the OpenMP parallel versions of 3DFFT. Support your explanations with the results reported in the following table which you obtained during the laboratory session. It is very important that you include the relevant Paraver captures (timelines and profiles of the % of time spent in the different OpenMP states) to support your explanations too.

Version	ϕ	S_∞	T_1	T_8	S_8
initial version in 3dfft omp.c	0.648	2.84	2.28s	1.41s	1.62
new version with improved ϕ	0.826	5.75	2.48s	0.91s	2.73
final version with reduced parallelisation overheads	0.904	10.36	2.34s	0.64s	3.66

Table 5. Parallel fraction and speed-ups for the 3 versions of 3dfft_omp.c

Table 5 above shows us the evolution of the parallel fraction, the potential speed up and the speed up for 8 threads over the three versions of 3dfft_omp.c that we saw in session 3 of this assignment. As we can see, the parallel fraction increases in each iteration of the program, that is because in the second version we parallelize *init_complex_grid*, and it increases even further for the third version because we increase the granularity and therefore we reduce overheads.

Execution time with one processor (T_1) is lowest with the initial version because we had yet to introduce further task creation and its accompanying overhead in *init_complex_grid*, thus the execution time is lower. However, the final version improves T_1 with respect to the second version because of the reduction in overhead time due to the granularity increase. In T_8 we do see the benefits of parallelism, where each execution has improved T_8 with respect to the previous iteration. Once again, that is explained by parallelising *init_complex_grid* and the later rise in granularity.

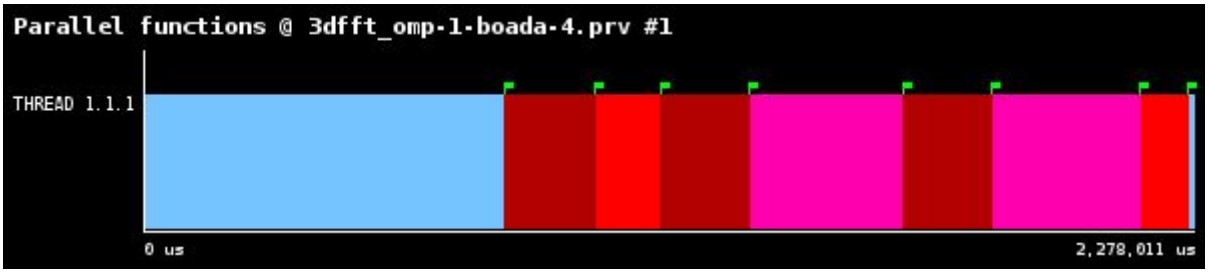

Figure 10. OMP parallel functions for initial version

	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	94.11 %	-	5.80 %	0.08 %	0.02 %	0.00 %
THREAD 1.1.2	30.68 %	64.38 %	4.92 %	-	0.01 %	-
THREAD 1.1.3	33.74 %	64.38 %	1.87 %	-	0.01 %	-
THREAD 1.1.4	34.35 %	64.37 %	1.26 %	-	0.02 %	-
THREAD 1.1.5	30.52 %	64.37 %	5.10 %	-	0.01 %	-
THREAD 1.1.6	34.74 %	64.38 %	0.86 %	-	0.02 %	-
THREAD 1.1.7	34.50 %	64.38 %	1.11 %	-	0.01 %	-
THREAD 1.1.8	34.29 %	64.39 %	1.31 %	-	0.01 %	-
Total	326.93 %	450.64 %	22.22 %	0.08 %	0.12 %	0.00 %
Average	40.87 %	64.38 %	2.78 %	0.08 %	0.01 %	0.00 %
Maximum	94.11 %	64.39 %	5.80 %	0.08 %	0.02 %	0.00 %
Minimum	30.52 %	64.37 %	0.86 %	0.08 %	0.01 %	0.00 %
StDev	20.19 %	0.01 %	1.96 %	0 %	0.00 %	0 %
Avg/Max	0.43	1.00	0.48	1	0.96	1

Figure 11. OMP state profile for initial version

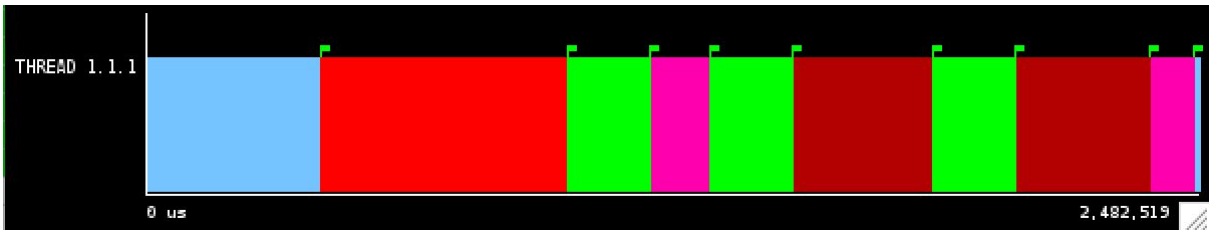

Figure 12. OMP parallel functions for second version

	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	91.24 %	-	8.55 %	0.17 %	0.03 %	0.00 %
THREAD 1.1.2	70.71 %	25.40 %	3.86 %	-	0.03 %	-
THREAD 1.1.3	66.87 %	25.40 %	7.69 %	-	0.03 %	-
THREAD 1.1.4	62.34 %	25.40 %	12.22 %	-	0.03 %	-
THREAD 1.1.5	69.00 %	25.40 %	4.56 %	-	0.03 %	-
THREAD 1.1.6	65.54 %	25.42 %	9.01 %	-	0.03 %	-
THREAD 1.1.7	66.57 %	25.40 %	7.99 %	-	0.03 %	-
THREAD 1.1.8	64.06 %	25.42 %	10.49 %	-	0.03 %	-
Total	557.34 %	177.85 %	64.38 %	0.17 %	0.26 %	0.00 %
Average	69.67 %	25.41 %	8.05 %	0.17 %	0.03 %	0.00 %
Maximum	91.24 %	25.42 %	12.22 %	0.17 %	0.03 %	0.00 %
Minimum	62.34 %	25.40 %	3.86 %	0.17 %	0.03 %	0.00 %
StDev	8.56 %	0.01 %	2.61 %	0 %	0.00 %	0 %
Avg/Max	0.76	1.00	0.66	1	0.93	1

Figure 13. OMP state profile for second version

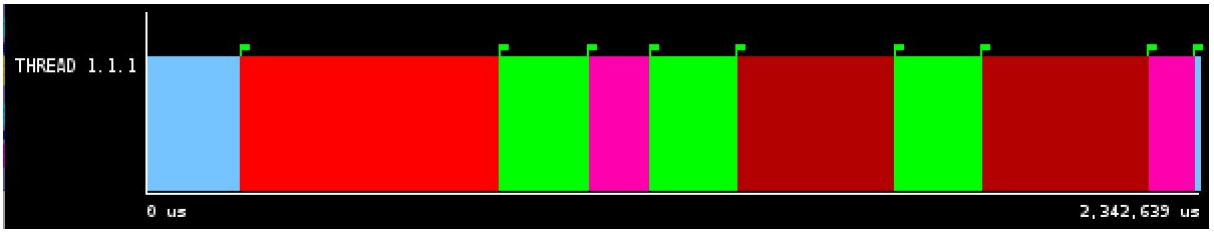

Figure 14. OMP parallel functions for final version

	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	99.54 %	-	0.16 %	0.30 %	0.00 %	0.00 %
THREAD 1.1.2	58.74 %	40.64 %	0.61 %	-	0.00 %	-
THREAD 1.1.3	54.46 %	40.64 %	4.89 %	-	0.00 %	-
THREAD 1.1.4	55.88 %	40.64 %	3.48 %	-	0.00 %	-
THREAD 1.1.5	58.52 %	40.64 %	0.83 %	-	0.00 %	-
THREAD 1.1.6	53.74 %	40.66 %	5.61 %	-	0.00 %	-
THREAD 1.1.7	58.54 %	40.65 %	0.81 %	-	0.00 %	-
THREAD 1.1.8	58.54 %	40.66 %	0.80 %	-	0.00 %	-
Total	497.97 %	284.52 %	17.20 %	0.30 %	0.01 %	0.00 %
Average	62.25 %	40.65 %	2.15 %	0.30 %	0.00 %	0.00 %
Maximum	99.54 %	40.66 %	5.61 %	0.30 %	0.00 %	0.00 %
Minimum	53.74 %	40.64 %	0.16 %	0.30 %	0.00 %	0.00 %
StDev	14.22 %	0.01 %	2.03 %	0 %	0.00 %	0 %
Avg/Max	0.63	1.00	0.38	1	0.42	1

Figure 15. OMP state profile for final version

Figures 10, 12 and 14 show the *OMP_parallel _functions* timeline for each version of the program, which we used to calculate ϕ and T_1, and consequently S_∞ as well.We just had to add the times of all the shown parallel sections of the execution, which gave us T_{PAR} and allowed us to calculate ϕ by dividing T_{PAR} by T_1. To calculate S_∞ we used Amdahl's law that tells us that $S_\infty = \frac{1}{1-}$.

The profiles showed in figures 11, 13 and 15 show the *OMP_state_profile* for all three executions. What is very clear at first sight is that by parallelizing *init_complex_grid* we see a considerable decrease in *Not Created* time that translates into and increase in *Running* time and to a lesser extent on *Synchronisation* time, that is because a greater part of the execution can be done in parallel and hence threads 2 to 8 can be running for more time. However, that also means that there will be more tasks that need to synchronise and hence *Synchronisation* represents a bigger percentage as well.

From the second version to the final one we see that the *Running* and *Synchronisation* percentages decrease and *Not Created* increases. That is a consequence of the increase in granularity and thus the reduction in overhead creation, which reduces the number of parallel tasks and hence overhead time and so the parallel region of the program executes faster due to having less tasks to wait for in *Synchronisation* mode. Therefore, as the parallel version of the program executes faster and the sequential part obviously stays the same, it is normal that the parallel threads are running for a smaller percentage of the total execution time.

Finally you should comment about the (strong) scalability plots (execution time and speed–up) that are obtained when varying the number of threads for the three parallel versions that you have analysed.

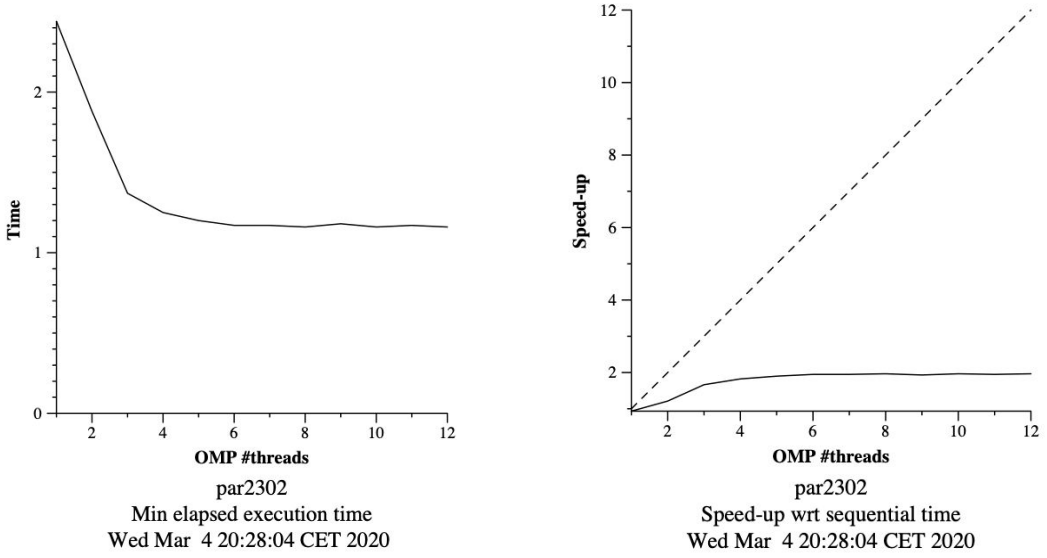

Figure 16. Strong scalability of initial version

Above in figure 16 we see the strong scalability of the initial version. It shows us that approximately using 4 threads or more gives very little or no visible improvement in execution time and therefore neither in speed-up.

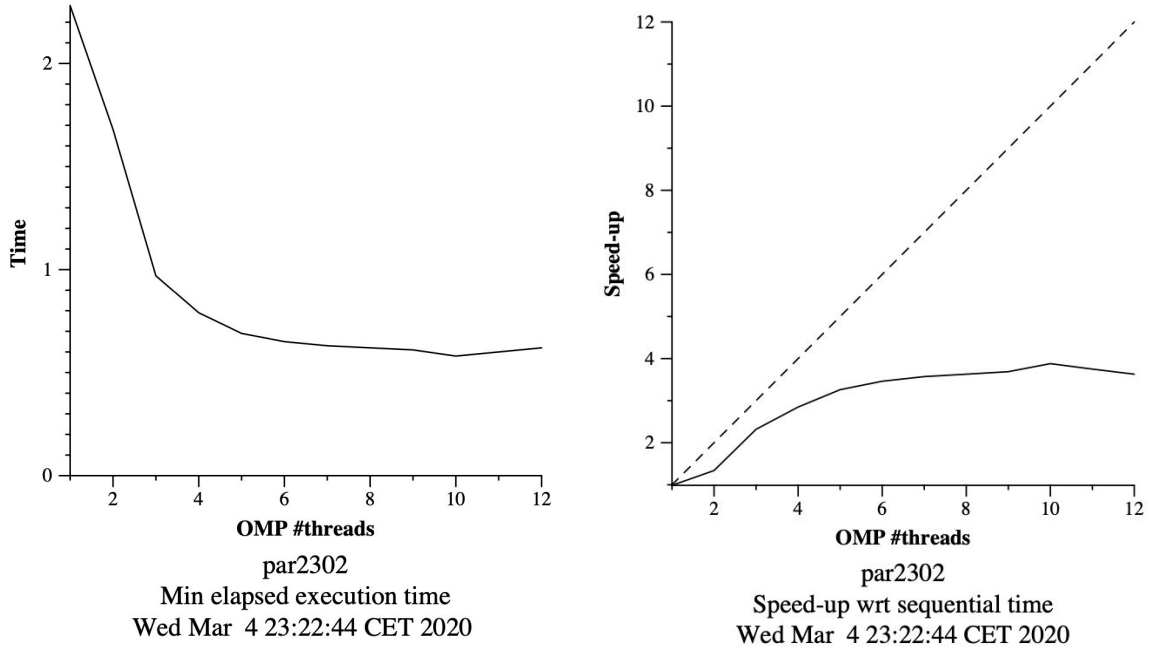

Figure 17. Strong scalability of initial version

In this second version of the program, where we parallelised *init_complex_grid*, we see a better strong scalability. The execution time seems to decrease up until the use of 10 thread

and then increases when using more. That could be explained by having to synchronise more threads and the overhead time that it incurs. Thus we see the best execution time when using 10 threads. When we look at the speed-up chart, we see that while it does increase all the way to 10 threads as we would expect from the previous graphic, it is far from the 45º line. However, we do see an improvement with respect to the previous version.

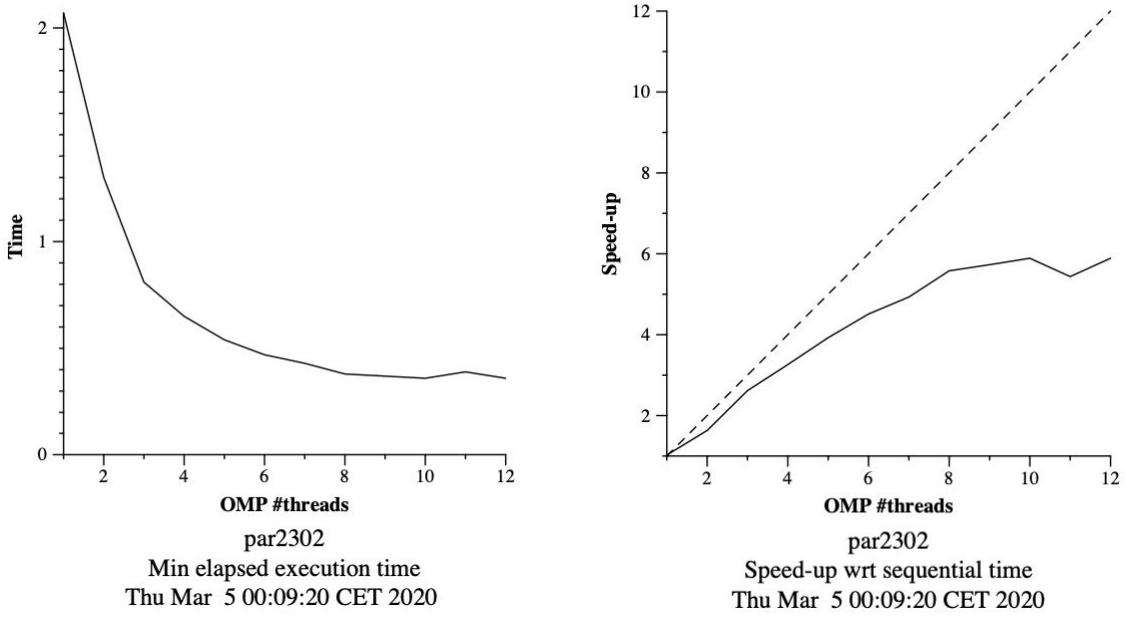

Figure 18. Strong scalability of initial version

Lastly, in this final version of 3dfft_omp.c we see an improvement over the second version as well, we see execution time decrease even more as we use more threads and this time we see a minimum at 10 and 12 threads. As we have seen above in a previous section of the report, that is given by a reduction in overhead time. On the speed-up graphic we see that the curve is much closer to the identity line than previous versions but as we use more threads the curve starts to flatten. Hence, we see that even though the granularity of this final version is increased and we would think that makes it less prone to parallelization, that effect is offset by the reduction in overhead time due to task creation and other aspects of the execution.