LABORATORY 1

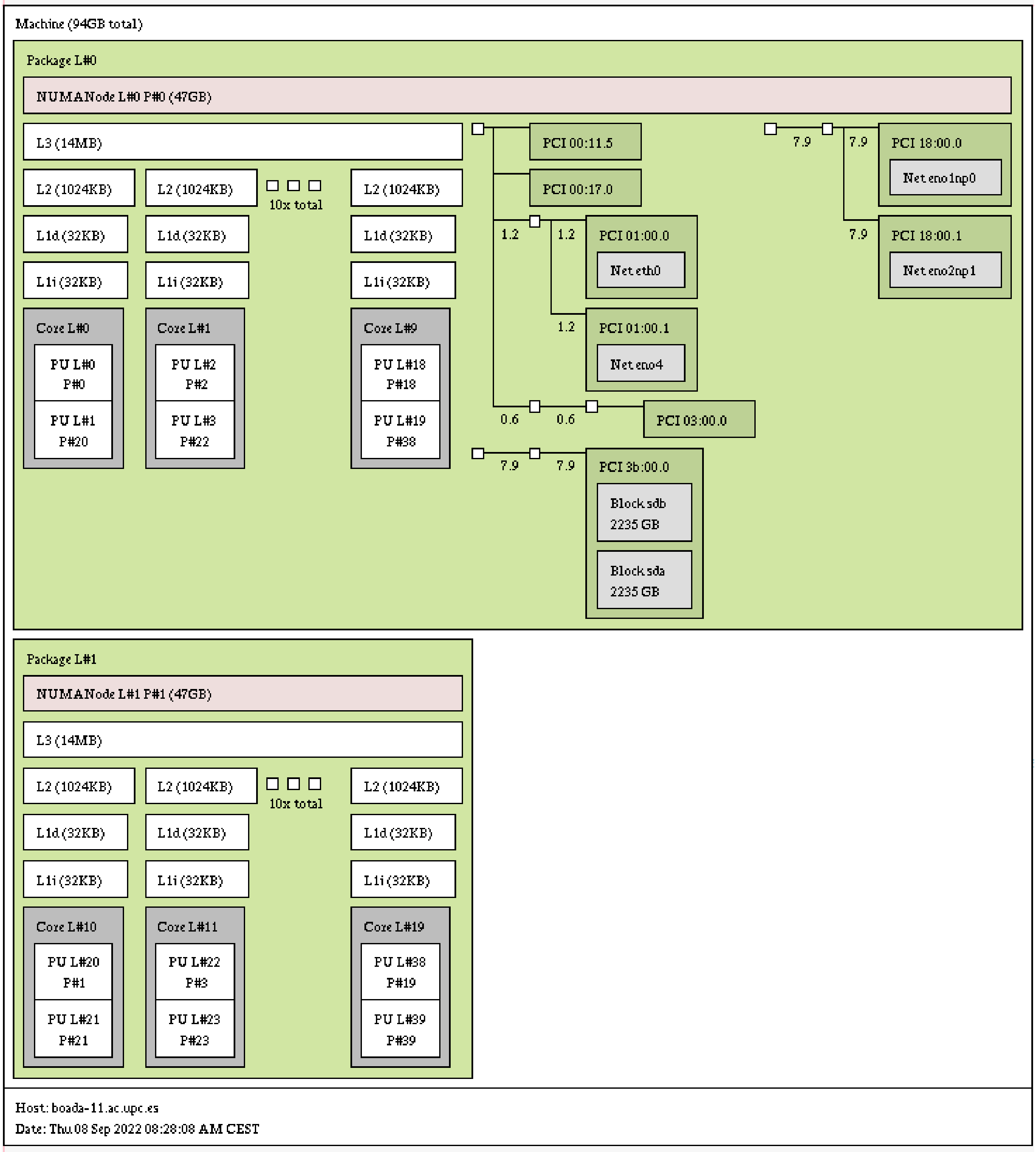
Group components: Albert Bausili Fernández, Noa-Yu Ventura Vila
Group identifier: par2315
29/09/2022

Index

1. Experimental setup

1.1 Node architecture

Boada server has 2 main operational parts, the boada 6-8 nodes which are used to run basic programs in "Interactive" mode with up to 2 parallel threads. The other cluster of nodes are the 11-14, which can run programs in "Execution" mode with a queue system and up to 20 parallel threads.

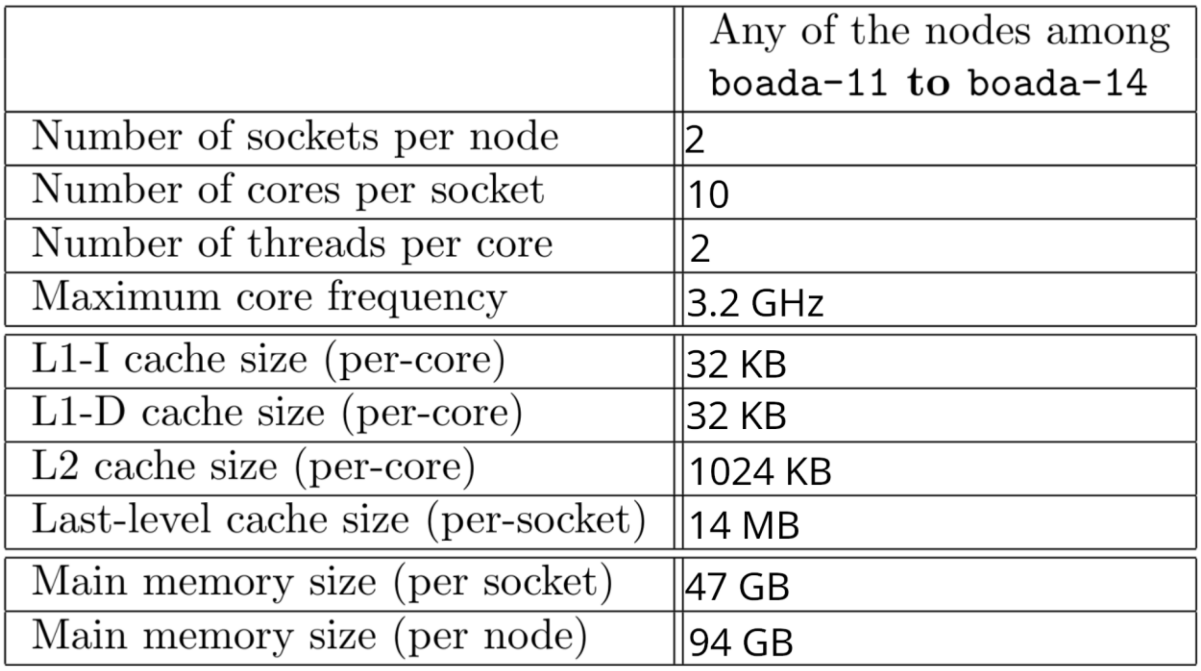


Node 11 scheme generated using xfig.

```
…
CPU max MHz:            3200.0000
CPU min MHz:            1000.0000
…
```

Abstract from lscpu output.

1.2 Compilation and execution of OpenMP programs



	Any of the nodes among boada-11 to boada-14
Number of sockets per node	2
Number of cores per socket	10
Number of threads per core	2
Maximum core frequency	3.2 GHz
L1-I cache size (per-core)	32 KB
L1-D cache size (per-core)	32 KB
L2 cache size (per-core)	1024 KB
Last-level cache size (per-socket)	14 MB
Main memory size (per socket)	47 GB
Main memory size (per node)	94 GB

Characteristics table of Boada 11 filled with the xfig and lscpu outputs.

1.3 Strong vs weak scalability for PI

# threads	Timing information (Interactive)				Timing information (Execution)			
	user	system	elapsed	% of CPU	user	system	elapsed	% of CPU
1	2.36s	0.00s	2.37s	99%	0.68s	0.00s	0.71s	97%
2	2.37s	0.00s	1.19s	199%	0.68s	0.00s	0.36s	190%
4	2.37s	0.00s	1.19s	199%	0.70s	0.00s	0.19s	367%
8	2.41s	0.03s	1.22s	199%	0.75s	0.00s	0.11s	661%
16	2.47s	0.11s	1.29s	198%	0.80s	0.00s	0.70s	1147%
20	2.47s	0.11s	1.30s	199%	0.83s	0.00s	0.60s	1331%

Timing table in Interactive (left) and Execution (right) of pi_omp program.

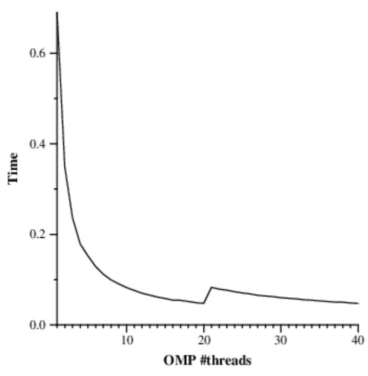
From the table we can deduce that the Interactive nodes had a maximum of 2 threads per program, because in spite of being the same code in both of them the Interactive one does not reduce the execution time past the threshold of 2 threads.

Another deduction that can be made is that the Interactive nodes are way less powerful than the Execution ones, because in the 1 thread test the elapsed time in the Execution node is approximately 3 times less than the other elapsed time.

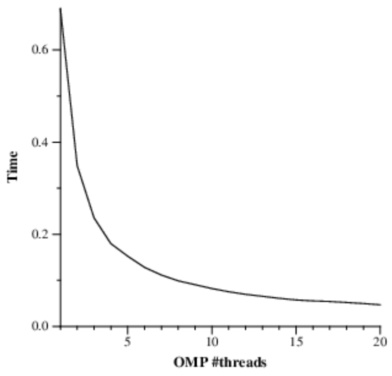
The last conclusion we observe is that in the Execution nodes the increment rate of the CPU usage reduces when the number of threads increases. We deduce that this is caused by the non-parallelizable code.

The scalability of a program is the speedup between the sequential and the parallel execution times. There are two types:

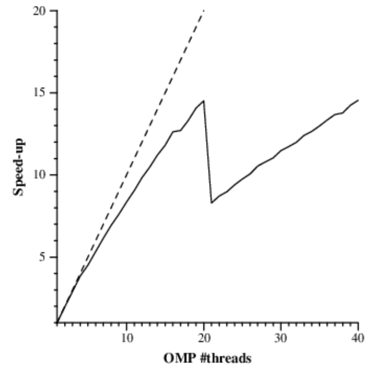
1. Strong scalability: we use parallelism to reduce the execution time of a program changing the number of threads to solve a problem of a fixed size (same number of iterations). In the minimum elapsed execution time graph of 20 cores we can see that the execution time of a problem rapidly decreases when we increase the number of threads.



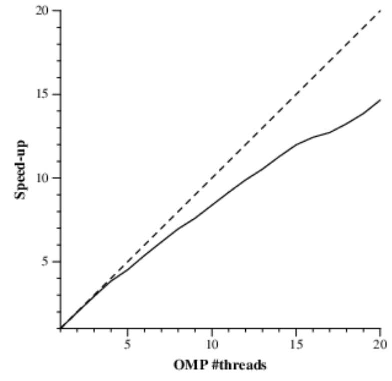
par2315
Min elapsed execution time
Thu Sep 8 10:21:20 AM CEST 2022



par2315
Min elapsed execution time
Thu Sep 8 09:50:25 AM CEST 2022

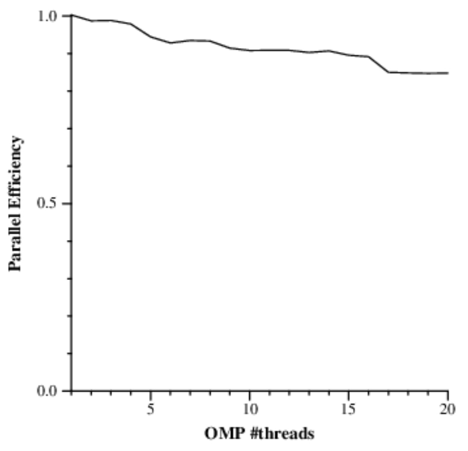


par2315
Speed-up wrt sequential time
Thu Sep 8 10:21:20 AM CEST 2022



par2315
Speed-up wrt sequential time
Thu Sep 8 09:50:25 AM CEST 2022

2. Weak scalability: in this case the size of the problem changes proportionally to the number of threads. In the parallel efficiency graph we can see that the line graph is almost horizontal, which means that the parallel efficiency doesn't improve as we increase the number of threads, unlike in the strong scalability case. That's mainly because of two reasons: there will always be non-parallelizable code we can't change, and the time needed for the synchronization of the threads also increases.

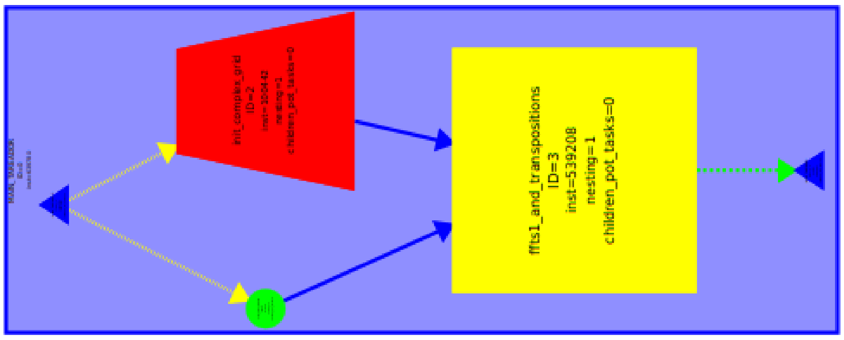


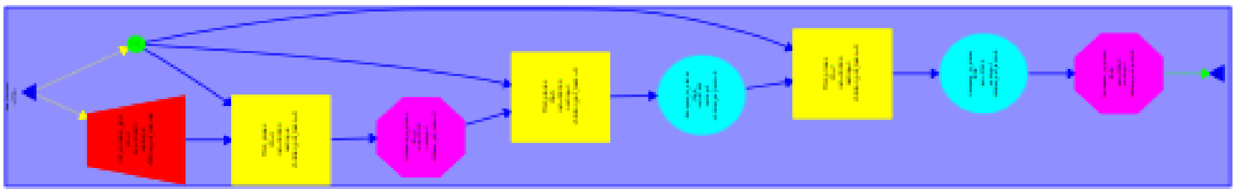
par2315
Parallel Efficiency w.r.t. one thread (weak scaling)
Thu Sep 8 10:28:28 AM CEST 2022

2. Systematically analyzing task decompositions with Tareador

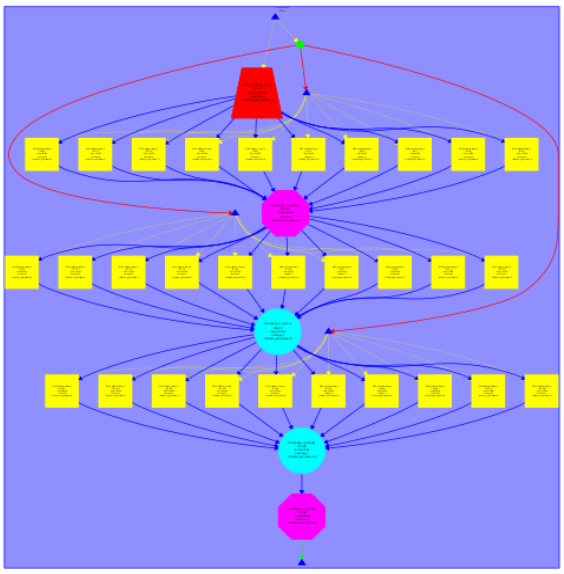
2.1 Exploring new task decompositions for 3DFFT

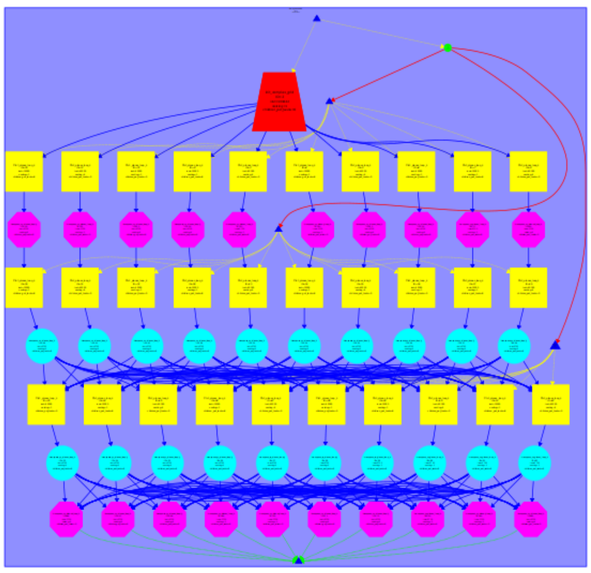
The first image corresponds to the Task Dependence Graph of the original code (3dfft_tarV0.c) and the second one corresponds to the version 1 (3dfft_tarV1.c), in which we split the function ffts1_and_transpositions into different tasks.



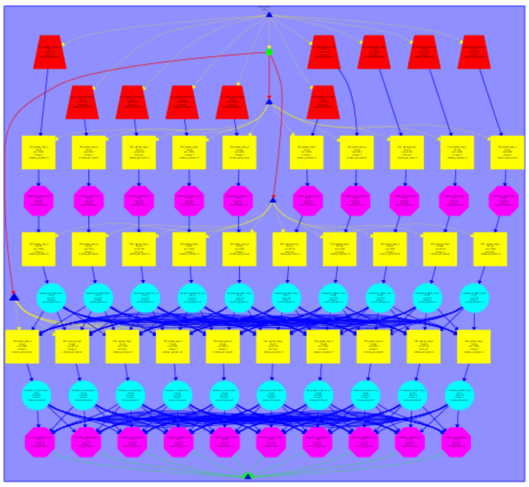


We keep splitting tasks and we get theTask Dependence Graph of version 2 (3dfft_tarV2.c) on the left and version 3 (3dfft_tarV3.c) on the right.

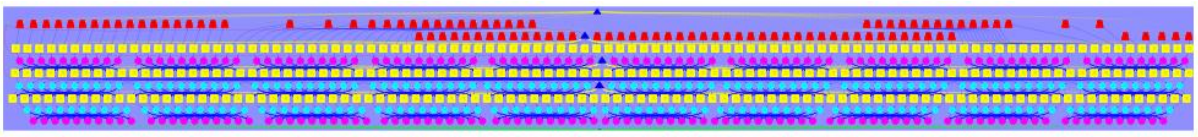




Task Dependence Graph of version 4 (3dfft_tarV4.c).



Task Dependence Graph of version 5 (3dfft_tarV5.c) with all functions parallelized.



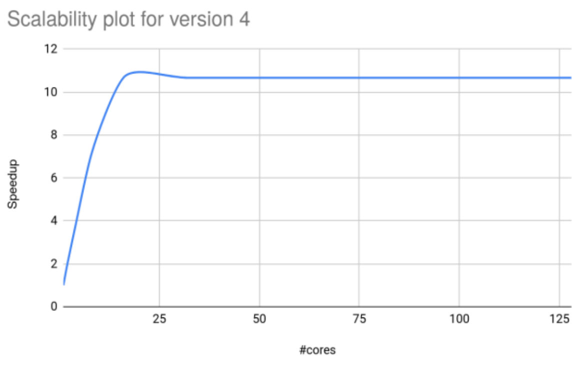
2.2 Execution table for all versions

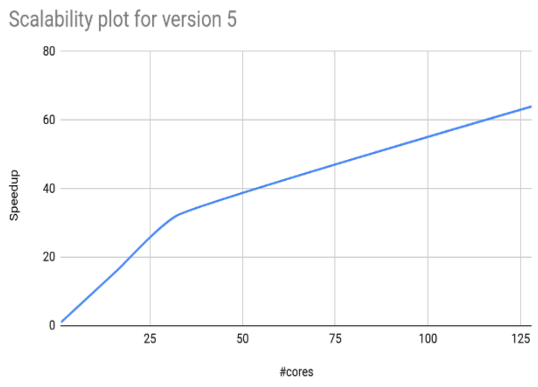
Version	T1	T∞	Parallelism
seq	0.64	0.64	1
v1	0.64	0.64	1
v2	0.64	0.36	1.78
v3	0.64	0.15	4.27
v4	0.64	0.06	10.67
v5	0.64	0.01	64

2.3 Timetable and scalability graphs for versions 4 and 5

Threads Num	V4	V4 Speed-up	V5	V5 Speed-up
1	0.64	1	0.64	1
2	0.32	2	0.32	2
4	0.17	3.76	0.16	4
8	0.09	7.11	0.08	8
16	0.06	10.67	0.04	16
32	0.06	10.67	0.02	32
128	0.06	10.67	0.01	64

We can see that version 4 has a weaker scalability than version 5. That's because in version 4 we tell the program that a task is equal to two loops, while in version 5 the task has only one loop, which makes the task faster. In the following sections we can see the difference in the respective codes.





2.4 Relevant code in version 4

```
…
        for (k=0; k<N; k++) {
            tareador_start_task("transpose_xy_planes_loop_k");
            for (j=0; j<N; j++) {
                for (i=0; i<N; i++) {
                    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_loop_k");
        }
…
```

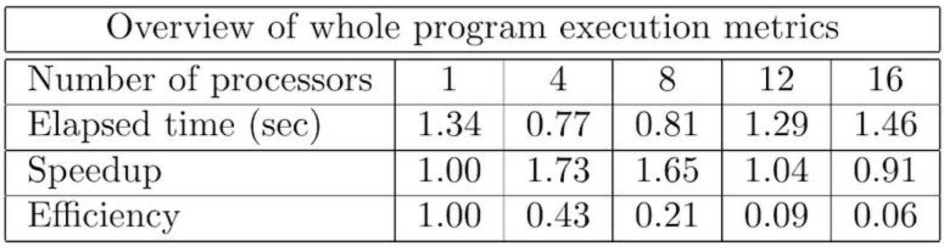
2.5 Relevant code in version 5

```
…
        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++) {
                        tareador_start_task("transpose_xy_planes_loop_j");
                        for (i=0; i<N; i++) {
                            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_loop_j");
                    }

                }

        }
…
```

3. Understanding the execution of OpenMP programs

3.1 Discovering modelfactors: Overall analysis

After a deep analysis of the modelfactors metrics we have reached some conclusions. The first one being that the scalability was really bad, as we can see in table 1, after the four threads execution the elapsed times just go worse and worse when adding more threads, even having a slower execution with sixteen threads than the sequential one.



Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	1.34	0.77	0.81	1.29	1.46
Speedup	1.00	1.73	1.65	1.04	0.91
Efficiency	1.00	0.43	0.21	0.09	0.06

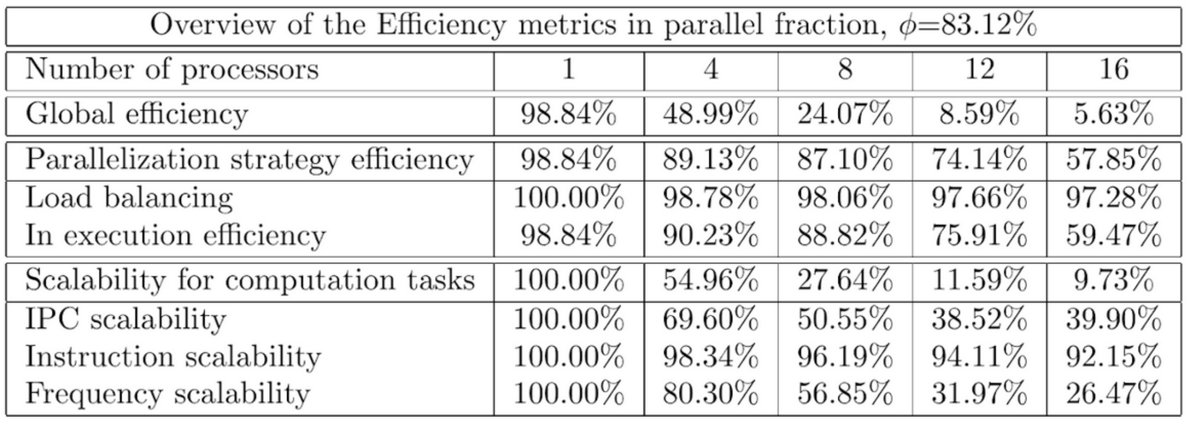
Table 1: Analysis done on Thu Sep 22 08:18:11 AM CEST 2022, par2315

The second conclusion was deduced from the first one, as we realized that the overheads due to synchronization are not negligible, as when the number of threads increased the synchronization time increased proportionally.

Then we computed the parallel fraction of the program using the formula:

$$S_4 = \frac{1.34}{0.77} = \frac{1.34}{(1-\phi)\cdot 1.34 + (\phi \cdot 1.34/4)} \rightarrow \phi = 0.54$$

Once we knew the parallel fraction of the program, we realized that the efficiency for the parallel regions is not quite right as we can see in the next figure:



Overview of the Efficiency metrics in parallel fraction, ϕ=83.12%					
Number of processors	1	4	8	12	16
Global efficiency	98.84%	48.99%	24.07%	8.59%	5.63%
Parallelization strategy efficiency	98.84%	89.13%	87.10%	74.14%	57.85%
Load balancing	100.00%	98.78%	98.06%	97.66%	97.28%
In execution efficiency	98.84%	90.23%	88.82%	75.91%	59.47%
Scalability for computation tasks	100.00%	54.96%	27.64%	11.59%	9.73%
IPC scalability	100.00%	69.60%	50.55%	38.52%	39.90%
Instruction scalability	100.00%	98.34%	96.19%	94.11%	92.15%
Frequency scalability	100.00%	80.30%	56.85%	31.97%	26.47%

Table 2: Analysis done on Thu Sep 22 08:18:11 AM CEST 2022, par2315

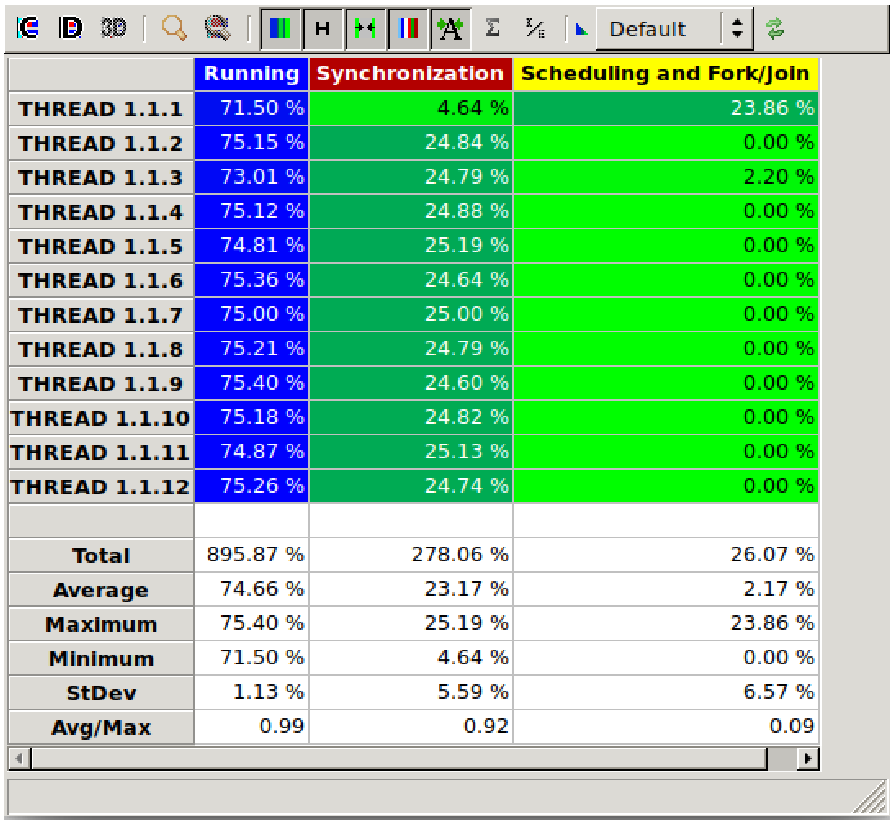
After seeing the huge difference between the expected efficiency and the real one we decided to find out the factor that was most important in terms of efficiency loss, and we concluded that that factor was the synchronization time, that represents, in the worst case a 67% of the whole execution time as we can see in the third figure:

Statistics about explicit tasks in parallel fraction					
Number of processors	1	4	8	12	16
Number of explicit tasks executed (total)	17920.0	71680.0	143360.0	215040.0	286720.0
LB (number of explicit tasks executed)	1.0	0.94	0.88	0.92	0.93
LB (time executing explicit tasks)	1.0	0.99	0.98	0.98	0.98
Time per explicit task (average us)	61.21	27.86	27.7	44.05	39.35
Overhead per explicit task (synch %)	0.16	10.64	12.78	31.36	67.52
Overhead per explicit task (sched %)	1.02	1.53	2.0	3.5	5.31
Number of taskwait/taskgroup (total)	1792.0	1792.0	1792.0	1792.0	1792.0

Table 3: Analysis done on Thu Sep 22 08:18:11 AM CEST 2022, par2315

3.2 Discovering Paraver (Part I): execution trace analysis

As we stated before, the overhead problem increases in function of the number of threads going from just a 0.16% with one thread to a 67% with sixteen threads. We find this gradual increase justified as the number of tasks increase proportionally to the number of threads but the amount of work done in each task is less and less, so in the end there are a huge number of little tasks that we need to synchronize while the work done by each of them is small despite having the same synchronization time.



	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	71.50 %	4.64 %	23.86 %
THREAD 1.1.2	75.15 %	24.84 %	0.00 %
THREAD 1.1.3	73.01 %	24.79 %	2.20 %
THREAD 1.1.4	75.12 %	24.88 %	0.00 %
THREAD 1.1.5	74.81 %	25.19 %	0.00 %
THREAD 1.1.6	75.36 %	24.64 %	0.00 %
THREAD 1.1.7	75.00 %	25.00 %	0.00 %
THREAD 1.1.8	75.21 %	24.79 %	0.00 %
THREAD 1.1.9	75.40 %	24.60 %	0.00 %
THREAD 1.1.10	75.18 %	24.82 %	0.00 %
THREAD 1.1.11	74.87 %	25.13 %	0.00 %
THREAD 1.1.12	75.26 %	24.74 %	0.00 %
Total	895.87 %	278.06 %	26.07 %
Average	74.66 %	23.17 %	2.17 %
Maximum	75.40 %	25.19 %	23.86 %
Minimum	71.50 %	4.64 %	0.00 %
StDev	1.13 %	5.59 %	6.57 %
Avg/Max	0.99	0.92	0.09

3.3 Discovering Paraver (Part II): understanding the parallel exec

3.3.1 Reducing Parallelisation Overheads and Analysis

After reviewing the histogram and timeline and as we were expecting from the previous results, we have concluded that the granularity of this program is really fine, or in other terms, there are a huge number of relatively small tasks.

Then we have analyzed the modelfactor output and from that we have drawn the following conclusions. The overall performance has improved in a highly sensible way, and can be seen especially in the speedup of table 1, which not only has not decreased as before, but now is more than three times higher than the sequential time. There are no slowdowns as there were before.

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	1.25	0.56	0.44	0.41	0.39
Speedup	1.00	2.23	2.83	3.05	3.25
Efficiency	1.00	0.56	0.35	0.25	0.20

Table 1: Analysis done on Wed Sep 28 10:35:48 PM CEST 2022, par2315

In the overheads is where we find the key of this efficiency improvement, as what before was a 67% and a 5% of synchronization and scheduling times respectively now these times have reduced to a 3% and a 0.05% each one of them as we can see in table 3.

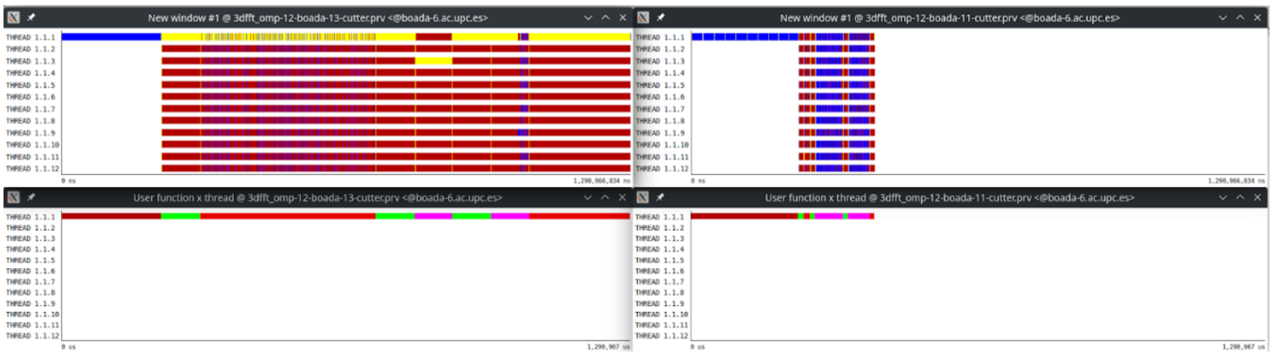
Statistics about explicit tasks in parallel fraction					
Number of processors	1	4	8	12	16
Number of explicit tasks executed (total)	80.0	290.0	570.0	850.0	1130.0
LB (number of explicit tasks executed)	1.0	0.85	0.84	0.82	0.86
LB (time executing explicit tasks)	1.0	0.7	0.55	0.45	0.41
Time per explicit task (average us)	15672.05	5259.85	3351.87	2585.27	2191.85
Overhead per explicit task (synch %)	0.0	2.82	2.59	3.98	3.04
Overhead per explicit task (sched %)	0.02	0.02	0.03	0.04	0.05
Number of taskwait/taskgroup (total)	8.0	8.0	8.0	8.0	8.0

Table 3: Analysis done on Wed Sep 28 10:35:48 PM CEST 2022, par2315

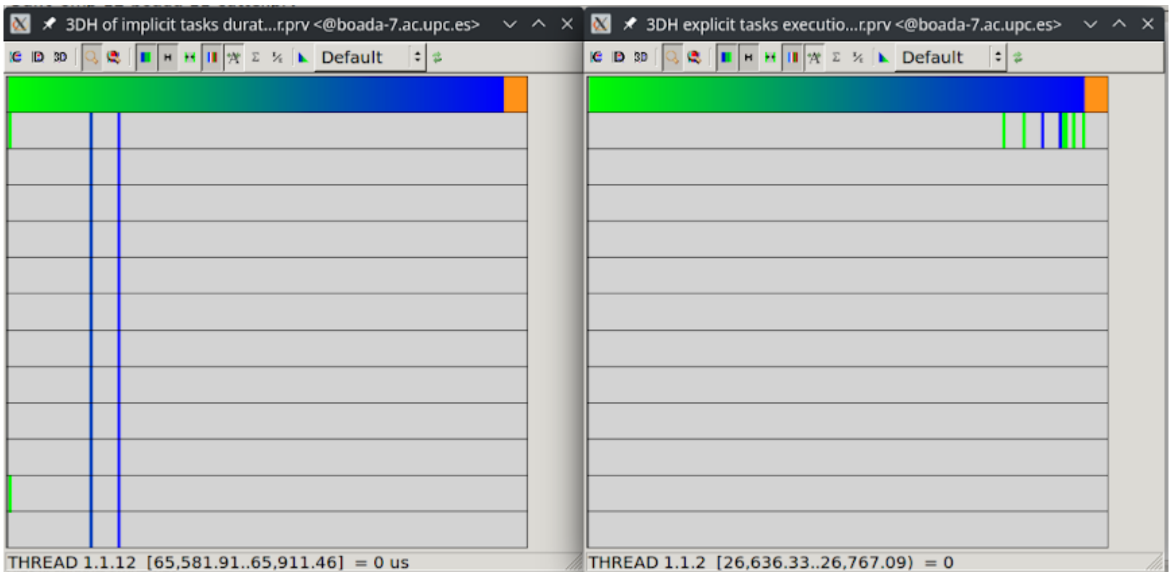
We have obtained that the parallel code section represents the 73.4% of the code, then aplying the amdahl law:

$$S_{max} = \frac{1}{1 - 0.734} = 3.76$$

Which tells us that the maximum speedup achievable with infinite processors would be of 3.76, and the one we have attained with 16 processors is 3.25 is quite close to that maximum, so as it has an asymptotically behavior, the amount of speedup gained while increasing the number of processors reduced proportionally.


New window #1 @ 3dfft_omp-12-boada-13-cutter.prv <@boada-6.ac.upc.es>
New window #1 @ 3dfft_omp-12-boada-11-cutter.prv <@boada-6.ac.upc.es>
User function x thread @ 3dfft_omp-12-boada-13-cutter.prv <@boada-6.ac.upc.es>
User function x thread @ 3dfft_omp-12-boada-11-cutter.prv <@boada-6.ac.upc.es>
THREAD 1.1.1 … THREAD 1.1.12

Single timeline windows of the original code (graph on the left) and the first-time optimized code (graph on the right).


3DH of implicit tasks durat...r.prv <@boada-7.ac.upc.es>
3DH explicit tasks executio...r.prv <@boada-7.ac.upc.es>
Default
THREAD 1.1.12 [65,581.91..65,911.46] = 0 us
THREAD 1.1.2 [26,636.33..26,767.09) = 0

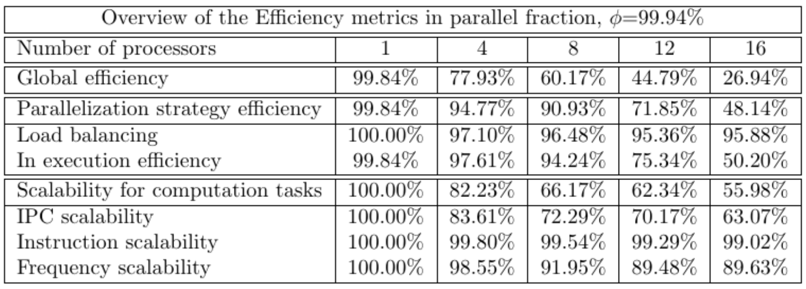
Histograms of implicit task duration for the first optimized version (Reducing Parallelisation Overheads and Analysis) and the second optimized version (Improving ϕ and Analysis).

3.3.2 Improving ϕ and Analysis

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	1.29	0.41	0.27	0.24	0.30
Speedup	1.00	3.12	4.82	5.37	4.31
Efficiency	1.00	0.78	0.60	0.45	0.27

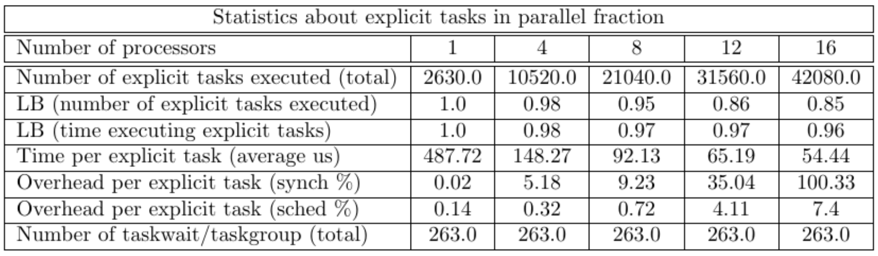
Table 1: Analysis done on Wed Sep 28 11:58:19 PM CEST 2022, par2315

Analyzing the last modelfactor output we can see that the function that limits the speedup the most is the synchronization overheads, as in the 16-core execution represents a 100.33% of the task time. All the functions are paralyzed, the only parts that are not paralyzed are the main code part which represents about 12.2% of the code.



Overview of the Efficiency metrics in parallel fraction, ϕ=99.94%					
Number of processors	1	4	8	12	16
Global efficiency	99.84%	77.93%	60.17%	44.79%	26.94%
Parallelization strategy efficiency	99.84%	94.77%	90.93%	71.85%	48.14%
Load balancing	100.00%	97.10%	96.48%	95.36%	95.88%
In execution efficiency	99.84%	97.61%	94.24%	75.34%	50.20%
Scalability for computation tasks	100.00%	82.23%	66.17%	62.34%	55.98%
IPC scalability	100.00%	83.61%	72.29%	70.17%	63.07%
Instruction scalability	100.00%	99.80%	99.54%	99.29%	99.02%
Frequency scalability	100.00%	98.55%	91.95%	89.48%	89.63%

Table 2: Analysis done on Wed Sep 28 11:58:19 PM CEST 2022, par2315

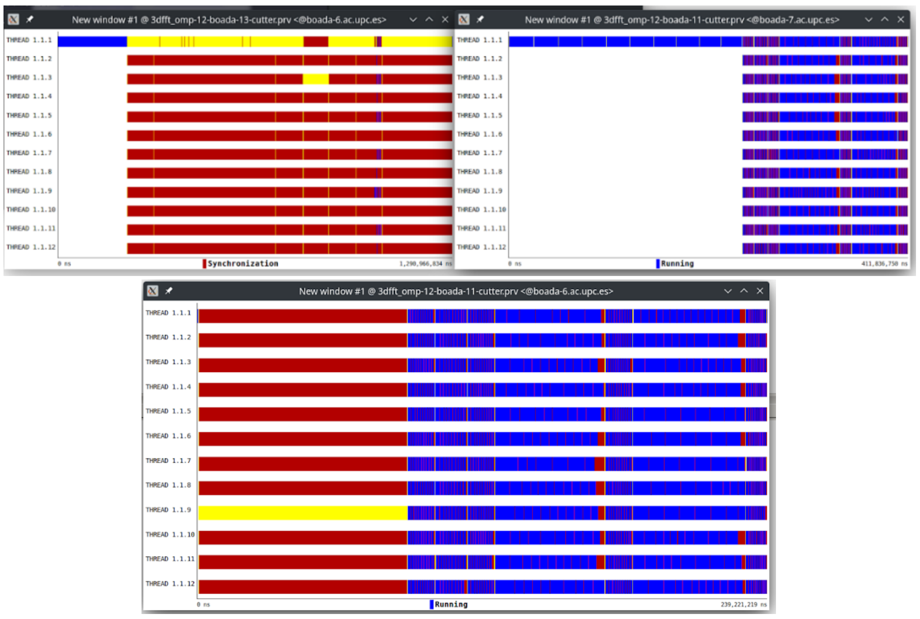


Statistics about explicit tasks in parallel fraction					
Number of processors	1	4	8	12	16
Number of explicit tasks executed (total)	2630.0	10520.0	21040.0	31560.0	42080.0
LB (number of explicit tasks executed)	1.0	0.98	0.95	0.86	0.85
LB (time executing explicit tasks)	1.0	0.98	0.97	0.97	0.96
Time per explicit task (average us)	487.72	148.27	92.13	65.19	54.44
Overhead per explicit task (synch %)	0.02	5.18	9.23	35.04	100.33
Overhead per explicit task (sched %)	0.14	0.32	0.72	4.11	7.4
Number of taskwait/taskgroup (total)	263.0	263.0	263.0	263.0	263.0

Table 3: Analysis done on Wed Sep 28 11:58:19 PM CEST 2022, par2315

Version	ϕ	ideal S12	T1	T12	real S12
initial version in 3dfft_omp.c	0.54	2.17	1.34	1.29	1.04
new version with reduced parallelisation overheads	0.73	3.7	1.25	0.41	3.05
final version with improved	0.88	8.33	1.29	0.24	5.37

As we can see in the two graphs on the top, there is a non-parallelized code at the beginning of the execution, which corresponds to the init_complex_grid function, but once we parallelize this function, we see a big improvement on the execution time of the last version (the graph at the bottom). In the original code we have a lot of small tasks because the tasks are defined on the inner loops, so there are a lot of context switches and that's why synchronization takes so much of the execution time. In the first and the second optimized versions the synchronization time greatly decreases because the granularity of the tasks is bigger and the number of tasks is reduced, which we achieved by uncommenting the outer loops of the other functions except the init_complex_grid function.



Timeline window of the code for the original version and the first and second optimized versions.