POLYTECHNIC UNIVERSITY OF CATALONIA

Lab 1

Experimental setup and tools

DELIVERABLE

PAR

Authors

LIANGWEI DONG (PAR 4202)
DAVID LATORRE ROMERO (PAR 4211)

2022-23 Q1

Experimental Setup

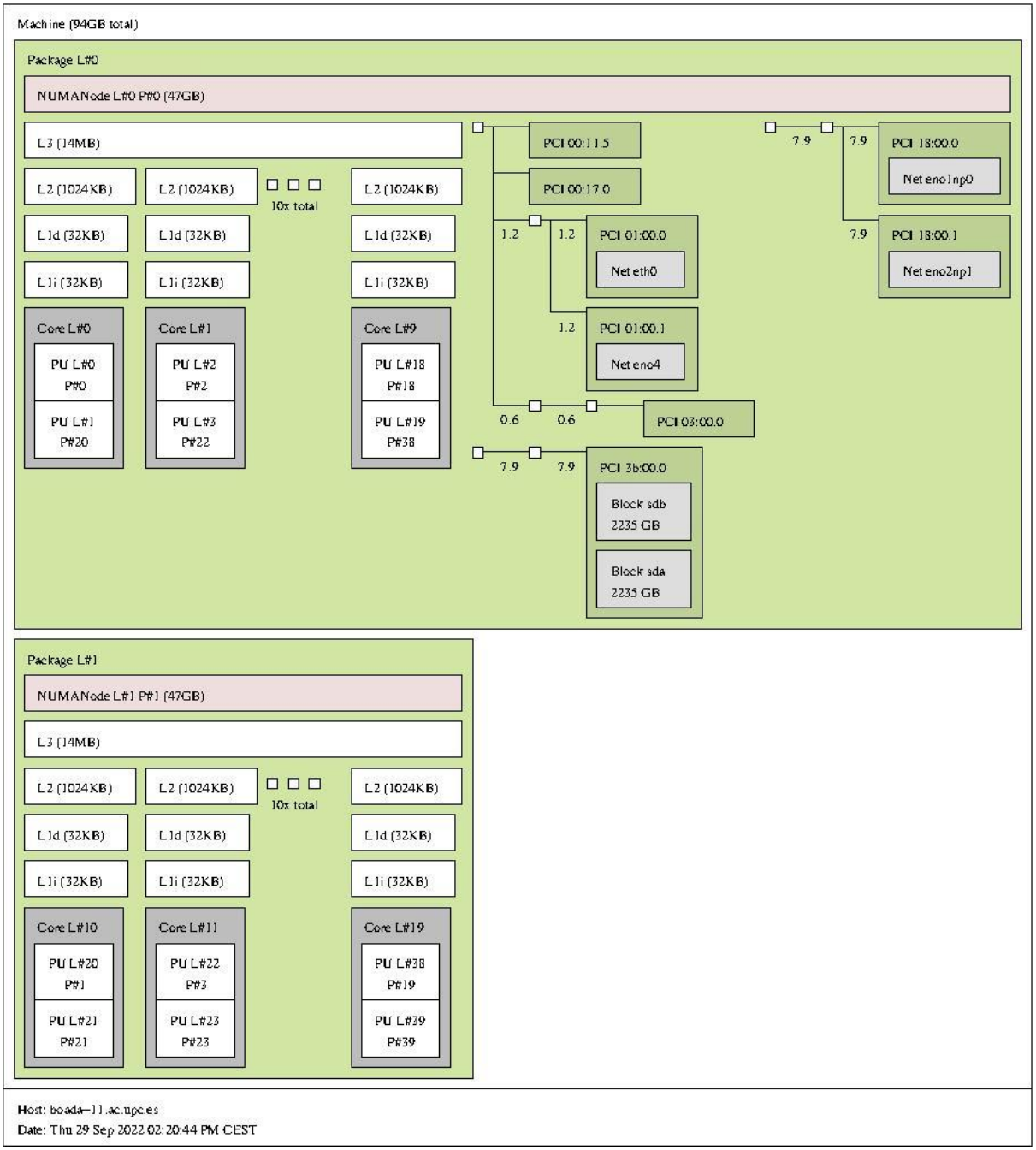
The objective of this laboratory session is to be familiarized with the hardware and software environment that will be used during the semester to do all laboratory assignments in PAR.

We will be using *boada*, a multiprocessor server located at the Computer Architecture Department. In this course we will use only nodes *boada-6* to *boada-8* interactively and nodes *boada-11* to *boada-14* through the execution queue.

Node architecture and memory

We ran sbatch submit-arch.sh command which executes the lscpu and lstopo commands in order to obtain information about the hardware in one of the node of execution queue (*boada-11 to 14*):

	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	3200 MHz
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)	47GB
Main memory size (per node)	94GB



Compilation and execution of *OpenMP* programs

We executed the program provided (*pi_omp.c*) in two ways, interactively and via a queueing system, with 1, 2, 4, 8, 16 and 20 threads and 1.000.000.000 iterations for each execution.

We obtained this table:

#threads	Timing information (interactive)				Timing information (queued)			
	user	system	elapsed	% of CPU	user	system	elapsed	% of CPU
1	2.36s	0.0s	2.37s	99%	0.68s	0.0s	0.7s	98%
2	2.37s	0.0s	1.19s	199%	0.69s	0.0s	0.36s	191%
4	2.37s	0.02s	1.20s	199%	0.71s	0.0s	0.19s	358%
8	2.38s	0.03s	1.21s	199%	0.76s	0.0s	0.11s	649%
16	2.43s	0.09s	1.27s	199%	0.78s	0.0s	0.07s	1119%
20	2.46s	0.13s	1.3s	199%	0.83s	0.0s	0.06s	1310%

We can see that in interactive mode, the user time is greater. That's because in queued mode, the execution is isolated and in interactive mode, the resources are shared with other programs.

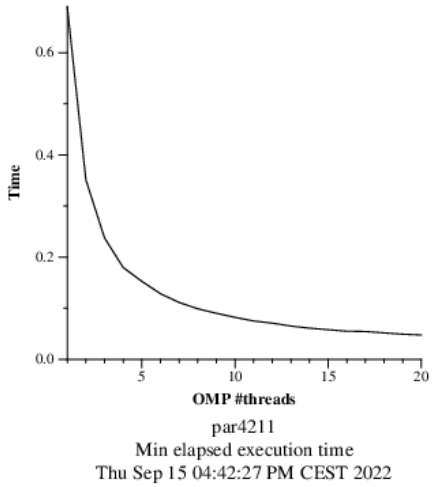
We also observe that the elapsed time reduces when the number of threads increases in queued mode but in interactive mode, it doesn't reduce from 2 threads. The reason why is that the system limits the number of cores to be used in parallel executions to two in interactive mode. This is also the reason why the percentage of CPU does not increase either.

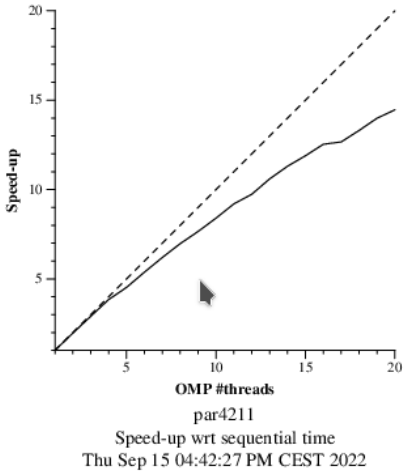
Strong vs. weak scalability

We are going to explore the scalability of the parallel version in pi omp.c when varying the number of threads used to execute the parallel code.

In strong scalability the number of threads is changed with a fixed problem size. In this case parallelism is used to reduce the execution time of the program. The problem size for strong scalability is 1.000.000.000 iterations.

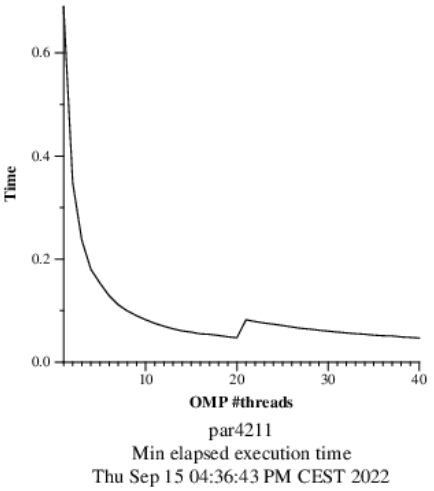
We get these two plots when we set *np_MAX* to 20:

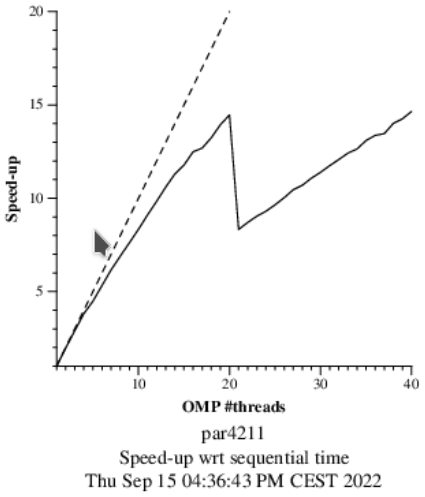




The speed-up increases really close to the diagonal but is not perfect due to overheads.

We get these two plots when we set *np_MAX* to 40:

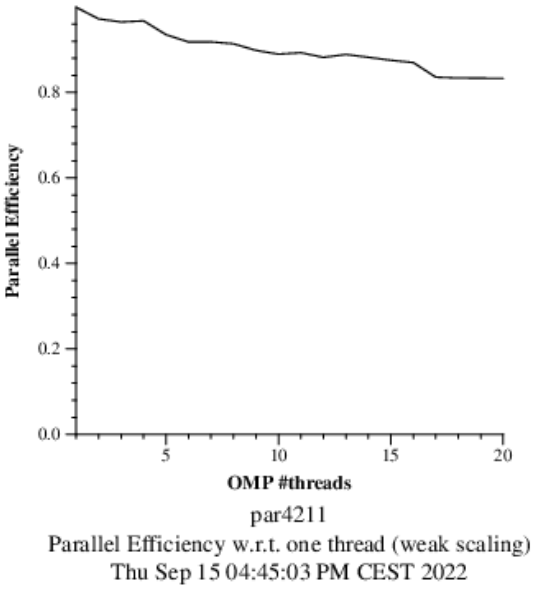




The speed-up falls with 21 threads because we only have 20 cores in a node. So this causes us to have 20 threads in parallel and when these have finished, the one thread left will begin, causing the fall in the speed-up.

In weak scalability the problem size is proportional to the number of threads. In this case parallelism is used to increase the problem size for which the program is executed. The initial problem size is 100.000.000 which grows proportionally with the number of threads.

We get this plot in weak scalability:



We observe that the parallel efficiency is always close to 1, that's because the number of threads is fitted to problem size, making it more efficient.
In strong scalability we have to set the number of threads respecting to the number of cores.

3 Systematically analysing task decompositions with Tareador

The purpose of this session is to get acquainted with the use and information that the tool "Tareador" gives to us.

With Tareador, we can analyze the parallelization of the execution of a program in a more visual way. Also, with it we can also simulate the execution with any number of threads we want, and produce timelines of the execution with x threads.

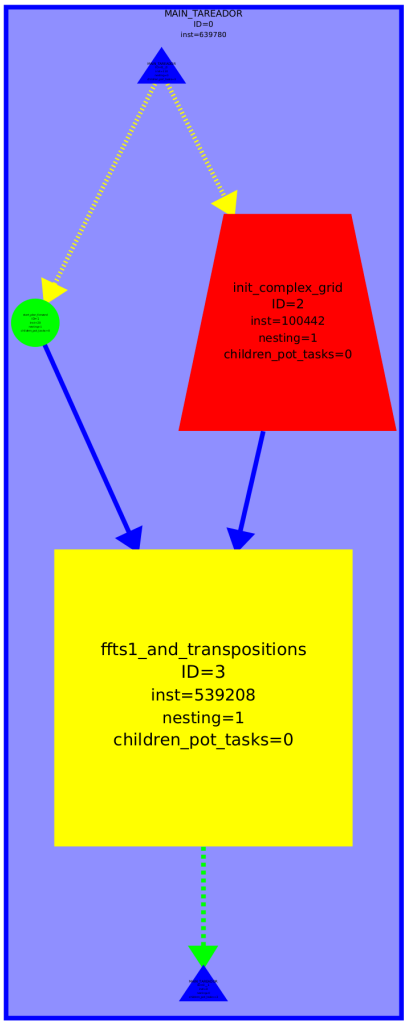
To get used to this tool, we are going to work with the file 3dfft_tar.c, which contains a program that executes different functions that have loops inside. From an initial version, we are going to make multiple improvements to it, increasing the parallelism respectively, proving this with the data that Tareador gives us.

The table below shows the different execution times and parallelization of the different versions:

Version	**T1**	**T∞**	**Parallelism**
seq	639,780,001 ns	639,707,001 ns	1.000114115
v1	639,780,001 ns	639,707,001 ns	1.000114115
v2	639,780,001 ns	361,190,001 ns	1.771311496
v3	639,780,001 ns	154,939,001 ns	4.129237938
v4	639,780,001 ns	64,614,001 ns	9.901569182
v5	639,780,001 ns	35.140.001 ns	18.206601673

Initial version:

Let's take a look at the graph that Tareador gives us with the initial version:



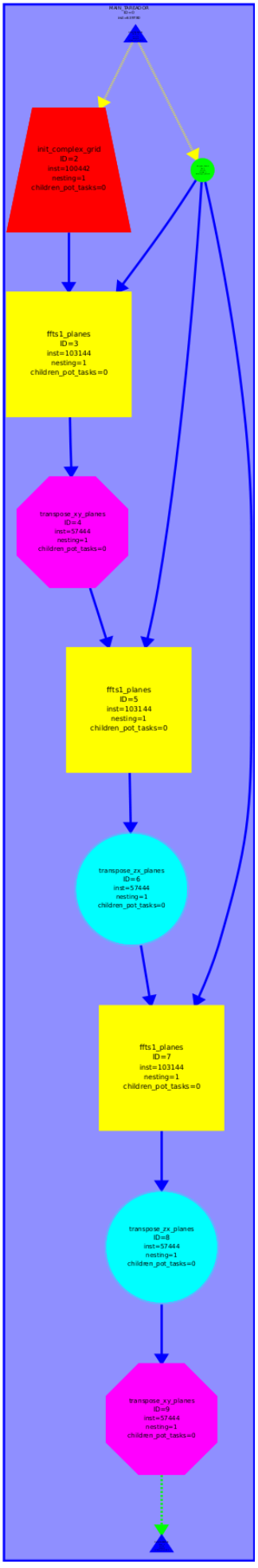
With this version the Tareador doesn't give us any useful information, mainly because we only have 1 “Tareador” task, where inside it there is the execution of the majority of the functions (yellow box).

Therefore, this is the cause why there is no parallelism (we can see this on the table above), because the functions are executed one after another, sequentially.

Version 1.0

In this version, we have replaced the only task we had with more fine-grained tasks. This is, now every task has the execution of a single function.

Tareador graph:

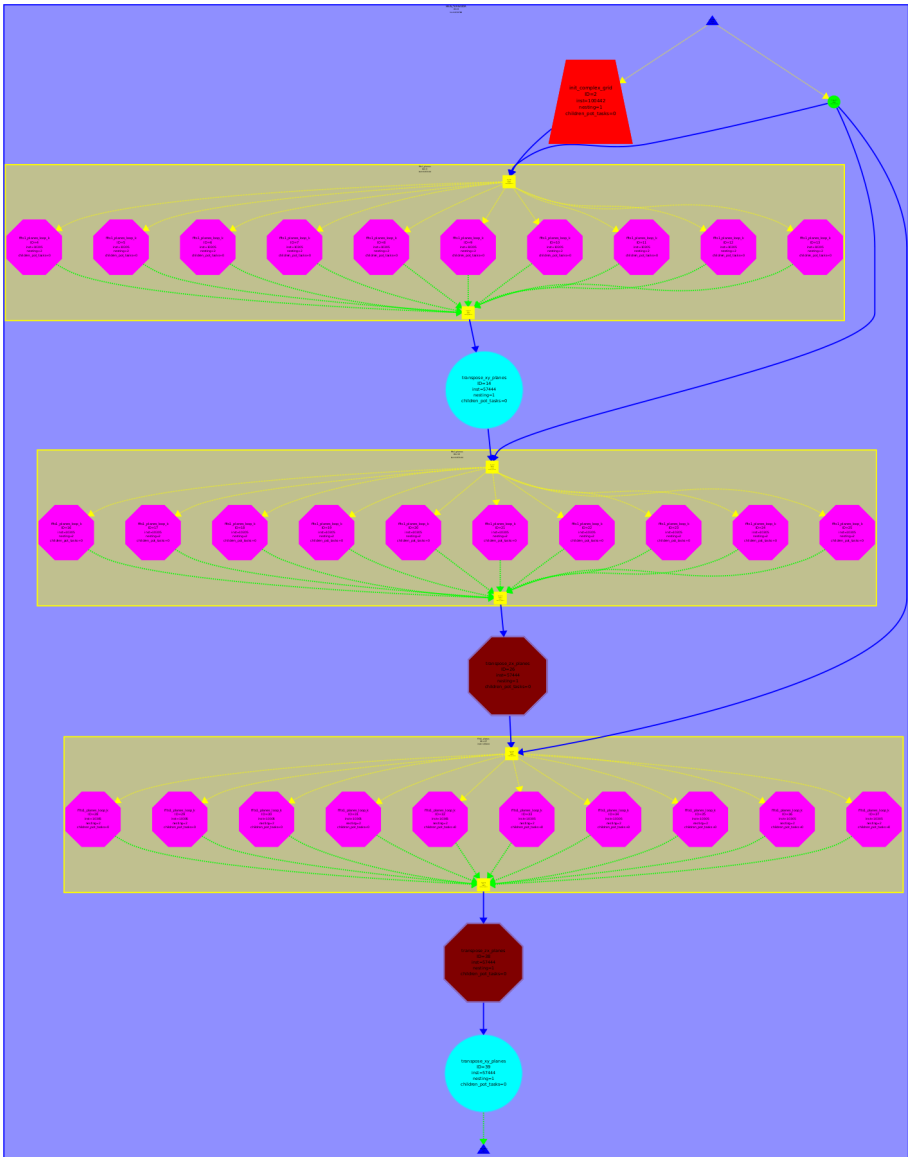

init_complex_grid
ID=2
inst=100442
nesting=1
children_pot_tasks=0
ffts1_planes
ID=3
inst=103144
nesting=1
children_pot_tasks=0
transpose_xy_planes
ID=4
inst=57444
nesting=1
children_pot_tasks=0
ffts1_planes
ID=5
inst=103144
nesting=1
children_pot_tasks=0
transpose_zx_planes
ID=6
inst=57444
nesting=1
children_pot_tasks=0
ffts1_planes
ID=7
inst=103144
nesting=1
children_pot_tasks=0
transpose_zx_planes
ID=8
inst=57444
nesting=1
children_pot_tasks=0
transpose_xy_planes
ID=9
inst=57444
nesting=1
children_pot_tasks=0

Now, at least the graph gives us more information, but we can still see that there is no parallelization, the functions are still executed one after another, sequentially.

Version 2.0:

Now we are going to modify the ffts1_planes function, creating new tasks on every iteration of the outer loop.

Tareador graph:



As we can see in the graph, doing this we now have different tasks being executed in parallel (the boxes that are aligned horizontally (pink boxes) are the ones that are being executed at

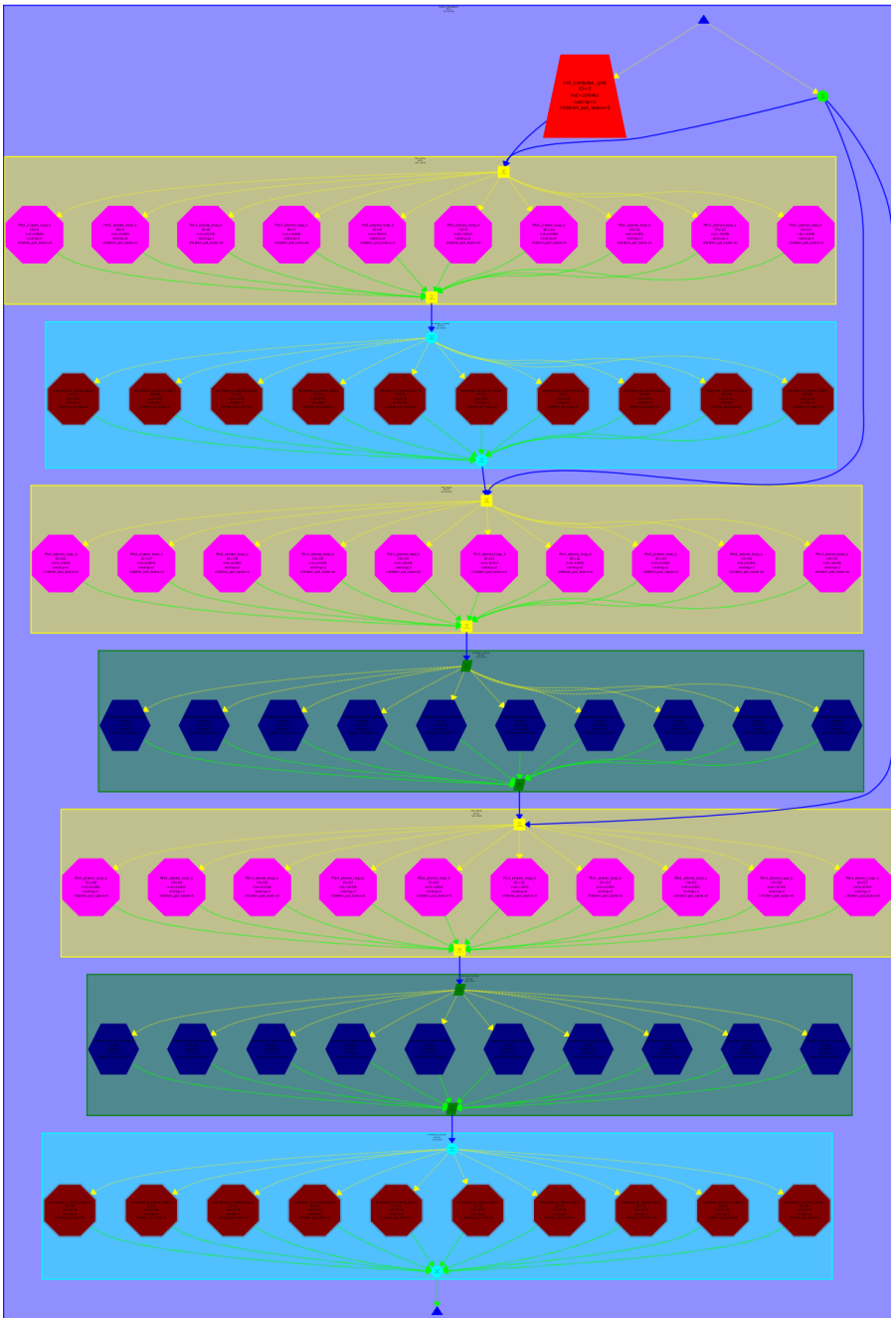
the same time). These boxes represent the tasks that execute the inner loop of the ffts1_planes function.

This is why, we can now start to get slower times with more than one thread (as seen in the table above).

Version 3.0

With the same logic as we did on the previous version, now let's do the same for the iterations of the outter loops of the functions transposexyplanes and transposezxplanes.

Tareador graph:

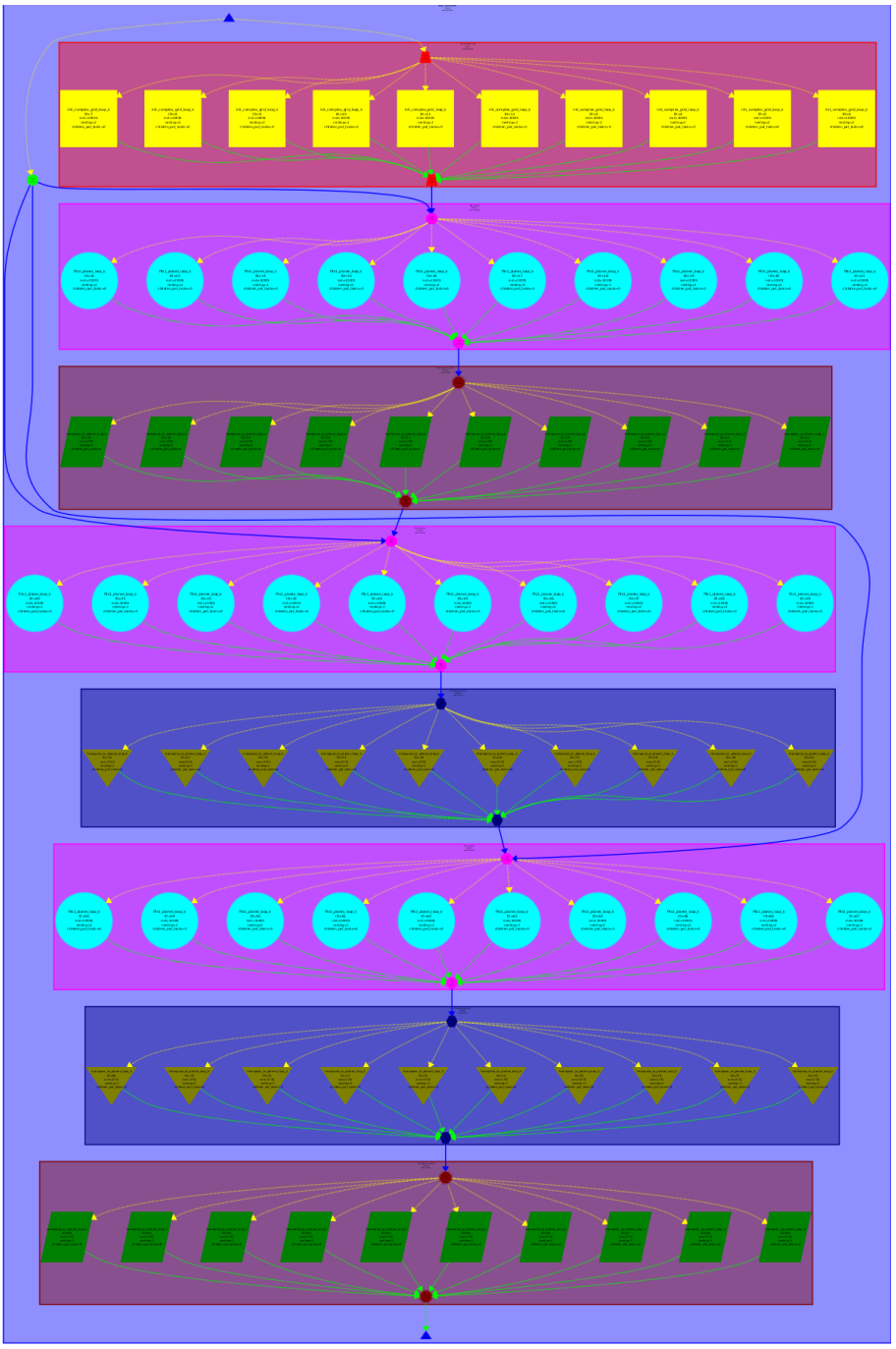


As expected, doing this we now have even more parallelization, because the parallel fraction of the program has increased. Therefore, lower times with more than 1 thread.

Version 4.0

Next, we do the same for the outer loop of the function init_complex_grid.

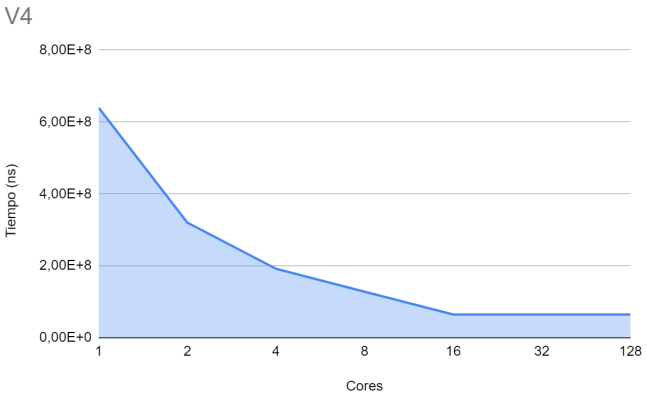
Paraver graph:

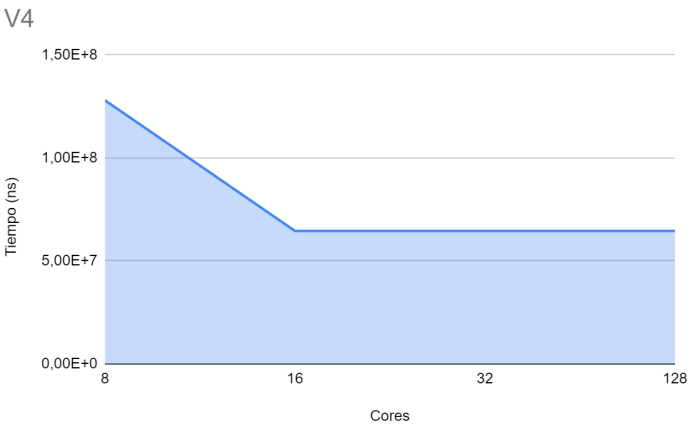


As expected, more parallel fraction leads to lower times. But let's take a look again at the graph, and the times of V4 on the table below.

Cores	Tiempo (ns)(V4)	Tiempo (ns)(V5)
1	639780001	639780001
2	320257001	301851987
4	191882001	160290001
8	127992001	82490001
16	64614001	51710001
32	64614001	41927001
128	64614001	35140001

Taking a look at this table we can see there is a bottleneck starting at 16 threads, in which the execution time does not decrease. That's why we do not have enough granularity to take advantage of more threads.



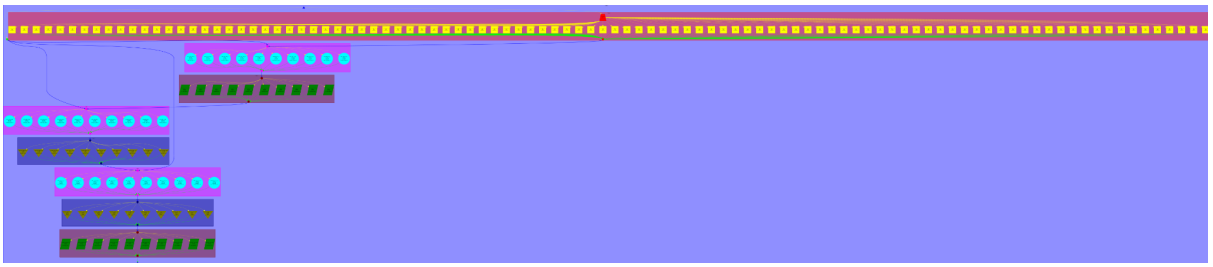


Version 5.0

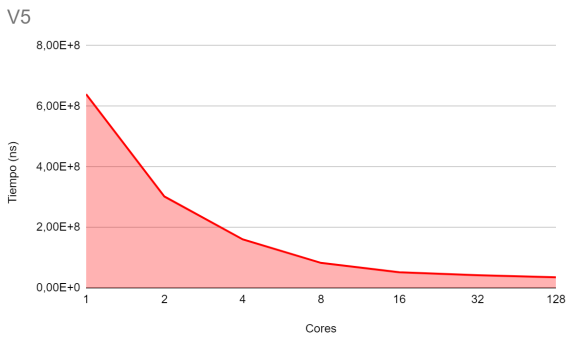
In this version we try to get rid of the bottleneck problem trying to increase the granularity. To do this, we change the creation of the tasks to the inner loops. Let's only to this for the

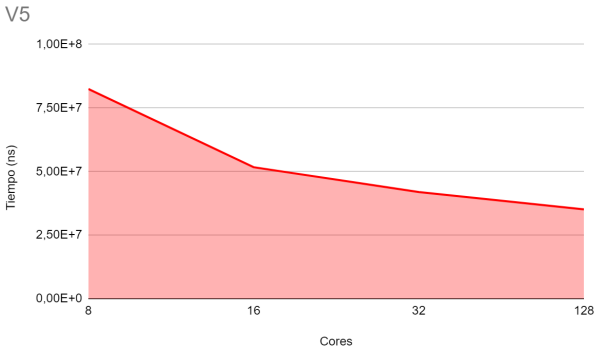
function init_complex_grid, otherwise Paraver could spend too much time creating the analysis.

Paraver graph:



As we can see, with more granularity we can now get rid of that bottleneck (table above) and get more parallelization, slower execution times.





4 Understanding the parallel execution of 3DFFT

The purpose of this session is to get acquainted with the information that the tools “paraver” and “modelfactors” give to us in terms of parallelization.

We are gonna analyze the execution of the program inside the file “3dfft_omp.c” with these tools, and use them to see what really happens in the parallel part and how we can improve it to be more efficient and, therebefore, faster. First, we will improve the initial program, and then we are going to implement a second improvement on top of this one.

Although we are gonna collect information for the execution of the 3 versions with 1,4,8,12 and 16 threads, we will specially focus on what happens on the execution with 12 threads, showing, at the end, a comparison table of the times and speedups for the 3 different versions.

- **3DFFT, initial version: Obtaining parallelisation metrics using modelfactors**

Running modelfactors, we have created 3 different tables, showing diverse metrics and information. Let’s take a look at them, and analyze how good is this initial version of 3DFFT:

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	1.31	0.78	0.80	1.27	1.48
Speedup	1.00	1.68	1.63	1.03	0.88
Efficiency	1.00	0.42	0.20	0.09	0.05

Table 1: Analysis done on Tue Sep 27 05:50:11 PM CEST 2022, par4211

Overview of the Efficiency metrics in parallel fraction, ϕ=83.85%					
Number of processors	1	4	8	12	16
Global efficiency	98.81%	49.32%	24.05%	8.60%	5.45%
Parallelization strategy efficiency	98.81%	88.97%	86.65%	71.13%	55.47%
Load balancing	100.00%	98.62%	97.68%	97.48%	96.93%
In execution efficiency	98.81%	90.22%	88.70%	72.97%	57.23%
Scalability for computation tasks	100.00%	55.43%	27.76%	12.10%	9.82%
IPC scalability	100.00%	68.67%	50.18%	37.79%	39.22%
Instruction scalability	100.00%	98.33%	96.20%	94.14%	92.15%
Frequency scalability	100.00%	82.09%	57.50%	34.00%	27.18%

Table 2: Analysis done on Tue Sep 27 05:50:11 PM CEST 2022, par4211

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	0.93	0.79	0.83	0.74
LB (time executing explicit tasks)	1.0	0.99	0.98	0.98	0.97
Time per explicit task (average us)	60.28	27.2	27.17	41.57	38.39
Overhead per explicit task (synch %)	0.16	10.83	13.26	36.51	74.44
Overhead per explicit task (sched %)	1.03	1.54	2.12	4.05	5.82
Number of taskwait/taskgroup (total)	1792.0	1792.0	1792.0	1792.0	1792.0

Table 3: Analysis done on Tue Sep 27 05:50:11 PM CEST 2022, par4211

Looking at table 1, we already can see that the efficiency of the executable running with more than 1 thread is not good, getting worse every time there are more threads. Efficiency is so poor that with 16 threads it reaches the point where it even gets slower than running on serial with 1 thread.

To find the reason why this happens, we can look at tables 2 and 3. The first thing to note is that table 2 tells us the parallel fraction of the program is 83%, meaning that the 17% of the executable runs in serial and we cannot make it faster. This is a factor that impacts on the global efficiency of the program.

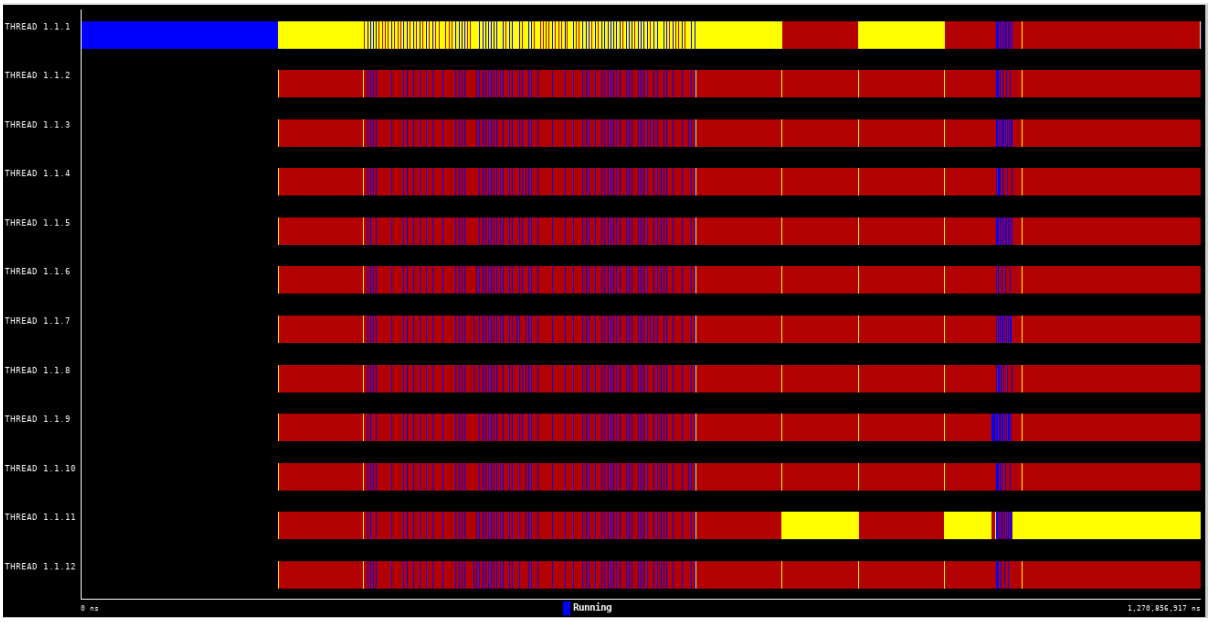
But, apart from this factor, there are others that affect the efficiency. From table 2, we can also see that on the parallel part, the scalability for computation tasks decreases considerably as the number of threads increases, especially the frequency scalability.

This is mainly because of the increment of the overhead per explicit task due to synchronization, as the number of threads is larger, as we can see in table 3. Overhead due to scheduling also takes part in the worsening of the efficiency, but the main factor is the overhead due to synchronization.

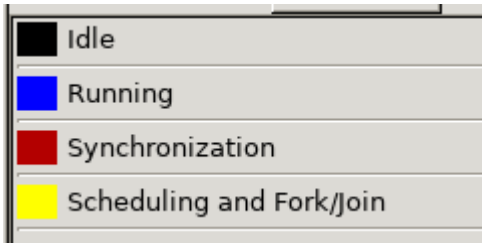
- **3DFFT, initial version: Paraver (Part 1)**

Now let's get deeper into the analysis with Paraver. Paraver is very useful because not only gives us a lot of information but also it gives it in a visual way.

Let's open the visual tray with 12 threads of the initial version, and take a look at the main timeline window:



The colors represent:



It's clear that looking at the timeline window, the thread 1.1.1 is the one that does the scheduling on the explicit tasks. But the first thing that catches our attention is this large period at the start of the execution where only the first thread is running.

With the next timeline (that displays the execution of the function on the timeline) we can see that this is due to the execution of the function init_complex_grid, a function that we are not parallelizing in this initial version:

Apart from this, it is also very important to note from the main timeline window that all the threads (except from the first one (that does scheduling) and the thread 11 on the final part (because it also does scheduling)) spend a lot of time on synchronization, waiting for the other threads to end. Let's confirm this expressing the data in percentages, since the timeline window when is very zoomed out could mislead us:

Default

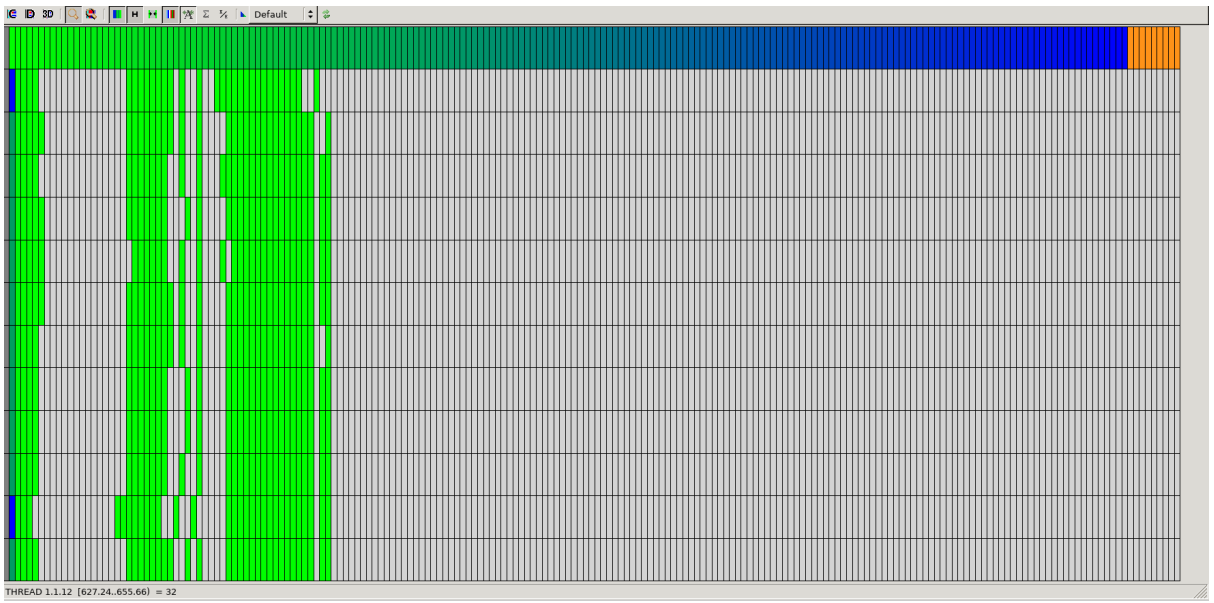
	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	71.52 %	5.03 %	23.45 %
THREAD 1.1.2	72.18 %	27.82 %	0.00 %
THREAD 1.1.3	71.78 %	28.22 %	0.00 %
THREAD 1.1.4	72.35 %	27.65 %	0.00 %
THREAD 1.1.5	71.89 %	28.11 %	0.00 %
THREAD 1.1.6	72.36 %	27.64 %	0.00 %
THREAD 1.1.7	71.85 %	28.15 %	0.00 %
THREAD 1.1.8	72.45 %	27.55 %	0.00 %
THREAD 1.1.9	72.24 %	27.76 %	0.00 %
THREAD 1.1.10	72.28 %	27.72 %	0.00 %
THREAD 1.1.11	66.64 %	27.28 %	6.08 %
THREAD 1.1.12	72.31 %	27.69 %	0.00 %
Total	859.86 %	310.60 %	29.54 %
Average	71.66 %	25.88 %	2.46 %
Maximum	72.45 %	28.22 %	23.45 %
Minimum	66.64 %	5.03 %	0.00 %
StDev	1.54 %	6.29 %	6.55 %
Avg/Max	0.99	0.92	0.10

On average, we spend 25% of the time on synchronization. Although from the timeline it seemed more, it is still a large percentage that produces this big overhead that lowers drastically the parallel efficiency of the program.

Also, from the table, we can see that the percentage that every thread (from thread 2 to 12) spends on synchronization is almost the same, meaning that this overhead problem doesn't seem due to the number of threads we are using but a constant problem.

- **3DFFT, initial version: Paraver (Part 2)**

Let's take a look now at the granularity of the explicit tasks for every thread, to see if the scheduling overhead problem affects it. Let's visualize this granularity in a histogram that indicates for each thread the time that every one of them spends for explicit task:



As we can see on the image, this overhead seems to affect the granularity, producing a very fine granularity in which every thread uses the same amount of time for type of explicit task, because for every explicit task there is a wait time for synchronization.

- **3DFFT: Reducing Parallelisation Overheads and Analysis**

Based on the problem we have just seen, let's see if we can increase this granularity by parallelizing the outer loops, and not the inner loops, trying to reduce the overhead due to synchronization.

Let's make the changes, and observe first the tables that modelfactors produce:

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	1.28	0.57	0.46	0.40	0.38
Speedup	1.00	2.26	2.80	3.21	3.32
Efficiency	1.00	0.56	0.35	0.27	0.21

Table 1: Analysis done on Wed Sep 28 05:22:37 PM CEST 2022, par4211

Overview of the Efficiency metrics in parallel fraction, ϕ=82.27%					
Number of processors	1	4	8	12	16
Global efficiency	99.96%	77.63%	60.06%	52.07%	45.69%
Parallelization strategy efficiency	99.96%	96.62%	96.76%	95.36%	97.21%
Load balancing	100.00%	97.96%	97.69%	97.54%	98.07%
In execution efficiency	99.96%	98.63%	99.05%	97.76%	99.13%
Scalability for computation tasks	100.00%	80.35%	62.07%	54.60%	47.00%
IPC scalability	100.00%	81.51%	66.05%	60.16%	52.03%
Instruction scalability	100.00%	99.99%	99.98%	99.97%	99.96%
Instruction scalability	100.00%	99.99%	99.98%	99.97%	99.96%
Frequency scalability	100.00%	98.59%	94.00%	90.80%	90.37%

Table 2: Analysis done on Wed Sep 28 05:22:37 PM CEST 2022, par4211

Statistics about explicit tasks in parallel fraction					
Number of processors	1	4	8	12	16
Number of explicit tasks executed (total)	70.0	280.0	560.0	840.0	1120.0
LB (number of explicit tasks executed)	1.0	0.97	0.9	0.95	0.81
LB (time executing explicit tasks)	1.0	0.98	0.98	0.98	0.98
Time per explicit task (average us)	15003.29	4667.78	3020.91	2289.27	1994.63
Overhead per explicit task (synch %)	0.0	3.43	3.22	4.68	2.65
Overhead per explicit task (sched %)	0.04	0.03	0.03	0.05	0.05
Number of taskwait/taskgroup (total)	7.0	7.0	7.0	7.0	7.0

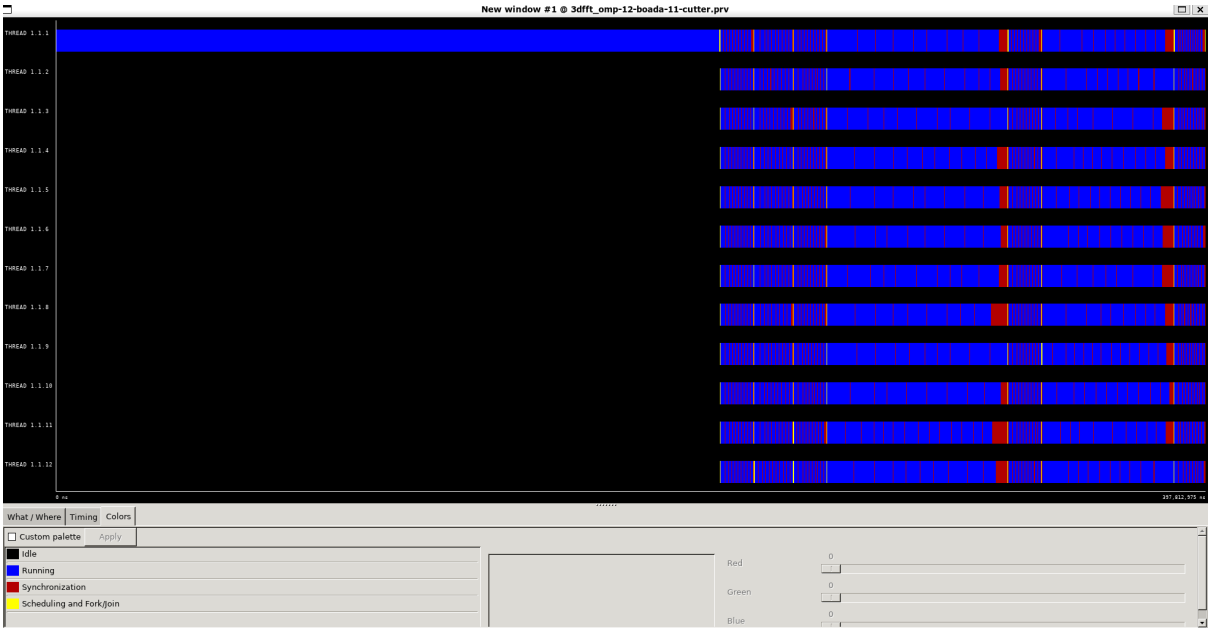
Table 3: Analysis done on Wed Sep 28 05:22:37 PM CEST 2022, par4211

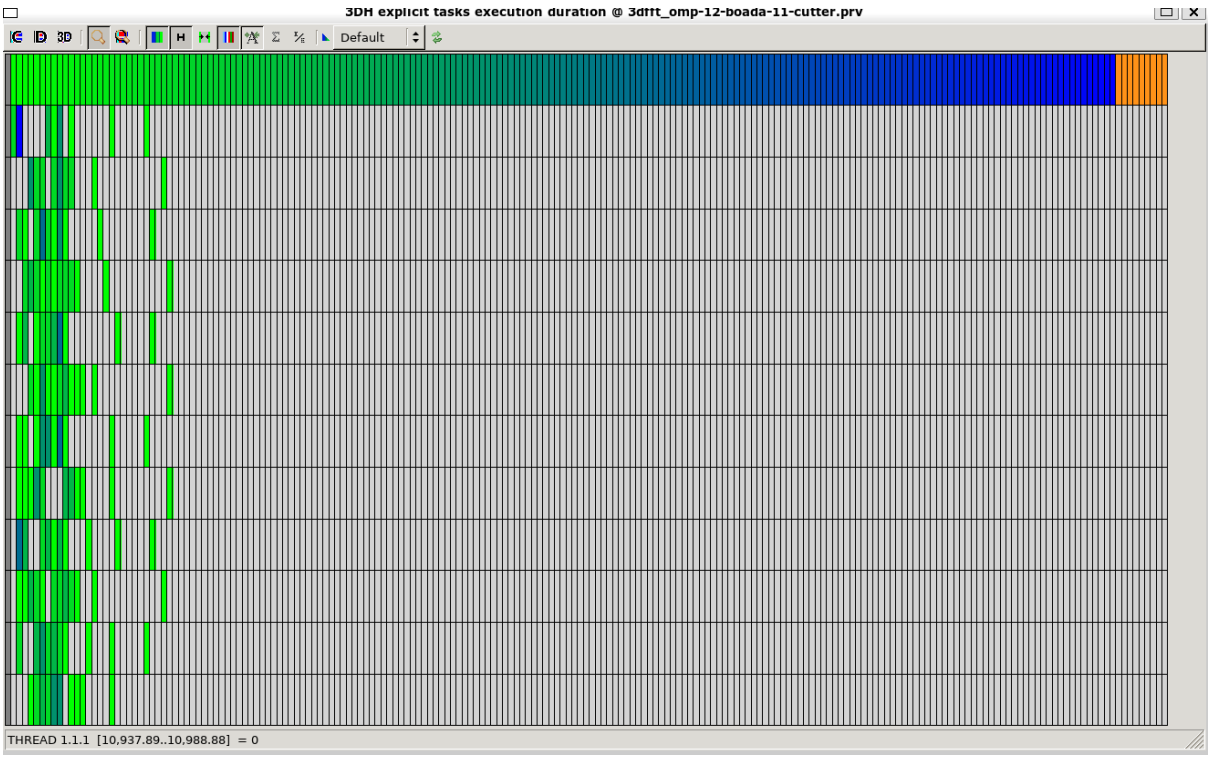
Observing table 1 we can already see that we have improved a lot with these simple modifications.

Although the efficiency for a lot of threads is still poor (which is logical because there is a function of the program we are not parallelizing), at least now the speedup is always higher as the number of threads is higher (at least up to 16 threads). The reason why the speedup is still less noticeable with a lot of threads is because as we increase the number of threads, the time per explicit task is being reduced but not as the overhead per explicit task, resulting in larger overheads per explicit task.

What's more, taking a look at table 3, we can now see that we have very little overhead per explicit task, resulting in a very good parallel efficiency.

Let’s now observe the timeline window and the granularity of the explicit tasks for this improved version with 12 threads:





Taking a look at the timeline window we can now see two main things: the first one, that now parallel part of the program takes a lot less in comparison to the serial part (which takes the

same time), and the second, that now our threads spend much more time running and therefore being useful, and not waiting in synchronization mode.

With the histogram we can see that we now have much more and diverse granularity, because we have no such large overheads in comparison to the time that an explicit task uses to take. Is important to note that the scale of the histogram has changed, now it is much larger simply because making the parallelization on the outer loops makes the explicit tasks take more time.

- **3DFFT: Improving φ and Analysis**

Let's now make the final improvement, which is gonna parallelize the function init_complex_grid, that function that had our first thread running alone for a long time at the beginning of the execution of the program.

As we did with the two other versions, we are going to first analyze the tables of modelfactors, and then the graphs of paraver:

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	1.28	0.41	0.25	0.23	0.29
Speedup	1.00	3.10	5.06	5.47	4.36
Efficiency	1.00	0.77	0.63	0.46	0.27

Table 1: Analysis done on Wed Sep 28 05:56:26 PM CEST 2022, par4211

Overview of the Efficiency metrics in parallel fraction, ϕ=99.94%					
Number of processors	1	4	8	12	16
Global efficiency	99.84%	77.37%	63.28%	45.61%	27.21%
Parallelization strategy efficiency	99.84%	95.39%	93.11%	73.10%	47.83%
Load balancing	100.00%	98.17%	97.77%	94.49%	95.12%
In execution efficiency	99.84%	97.17%	95.24%	77.36%	50.28%
Scalability for computation tasks	100.00%	81.10%	67.96%	62.40%	56.90%
IPC scalability	100.00%	83.56%	74.11%	70.11%	64.08%
Instruction scalability	100.00%	99.80%	99.54%	99.28%	99.03%
Frequency scalability	100.00%	97.25%	92.13%	89.65%	89.66%

Table 2: Analysis done on Wed Sep 28 05:56:26 PM CEST 2022, par4211

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	0.91	0.95	0.88	0.84
LB (time executing explicit tasks)	1.0	0.99	0.98	0.97	0.97
Time per explicit task (average us)	487.35	150.22	89.63	65.08	53.52
Overhead per explicit task (synch %)	0.02	4.49	6.65	32.88	101.62
Overhead per explicit task (sched %)	0.14	0.33	0.71	3.89	7.45
Number of taskwait/taskgroup (total)	263.0	263.0	263.0	263.0	263.0

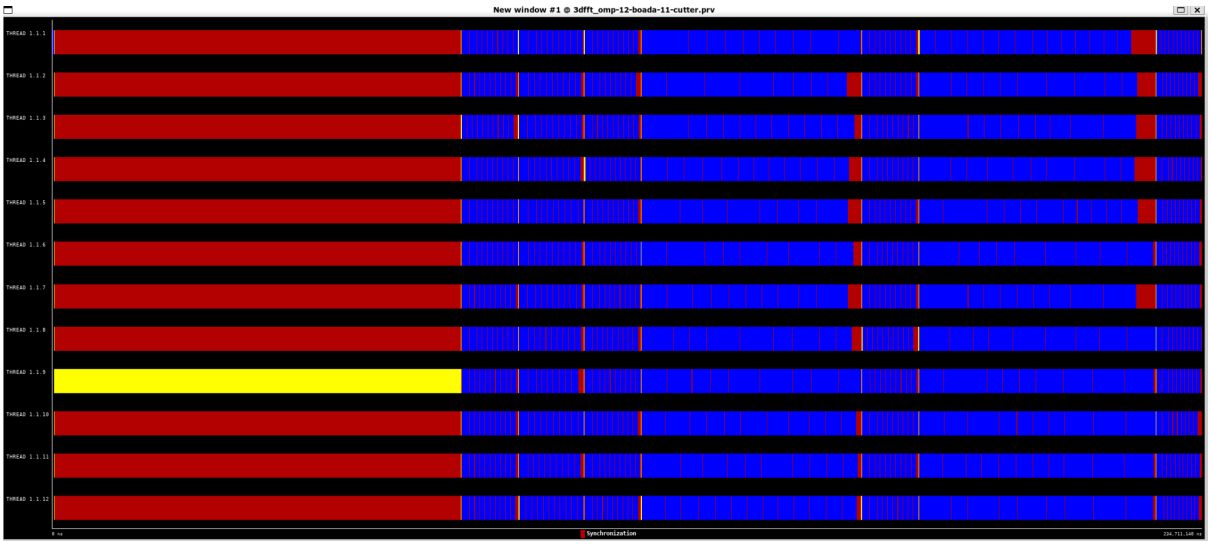
Table 3: Analysis done on Wed Sep 28 05:56:26 PM CEST 2022, par4211

Observing the first table, once again we obtain larger speedups.

It is important to note that this time we are getting slower times for 16 threads than for 12 threads mainly because for the function that we have just parallelized, we did it like in the first version, that is, parallelizing the inner loop.

This is why, if we take a look at tables 2 and 3, we have again worse parallel efficiency and larger overheads per explicit task.

Timeline of Paraver:



We can now see that we use all the threads for the whole execution of the program.

What is important to note about the timeline is that we now spend a lot of time in synch in the function init_complex_grid, because as we have just explained, for this function we parallel the inner loops and not the outer ones.

- **Conclusions:**

Let's now make the comparison table between the three versions, showing the increasing speedups and parallel fraction for the 2 improved versions we've just made. Let's also include some plots about the time and speed-up and scalability for each version:

Version	φ (%)	ideal S12	T1(s)	T12(s)	real S12
initial version in 3dfftomp.c	83,85	1,083	1,31	1,27	1,0315
new version with reduced parallelisation overheads	82,27		1,28	0,4	3,2
final version with improved φ	99,94		1,28	0,23	5,56

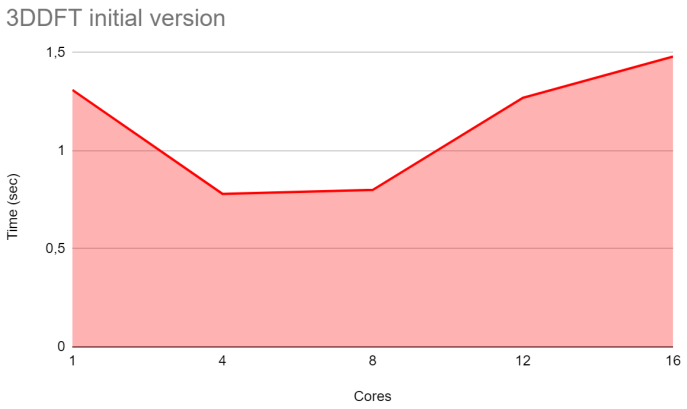
Ideal speedup can be computed with the following formula:

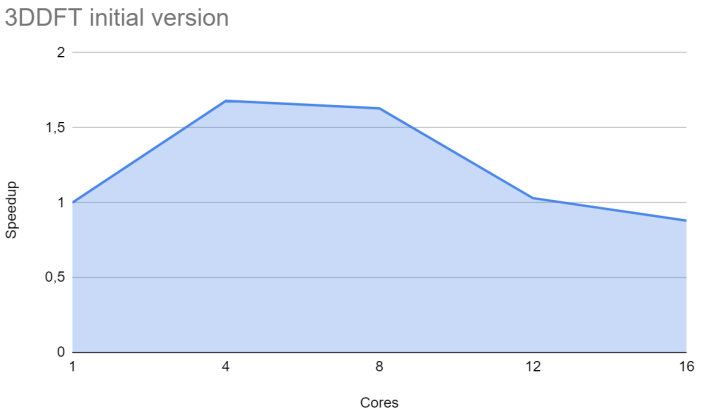
$$S_{overall} = \frac{1}{(1-f) + \frac{f}{S_{part}}}$$

Where Soverall is Sp, f is the fraction of the algorithm that can be parallelized (φ) and Spart is p (the corresponding speedup, number of cores).

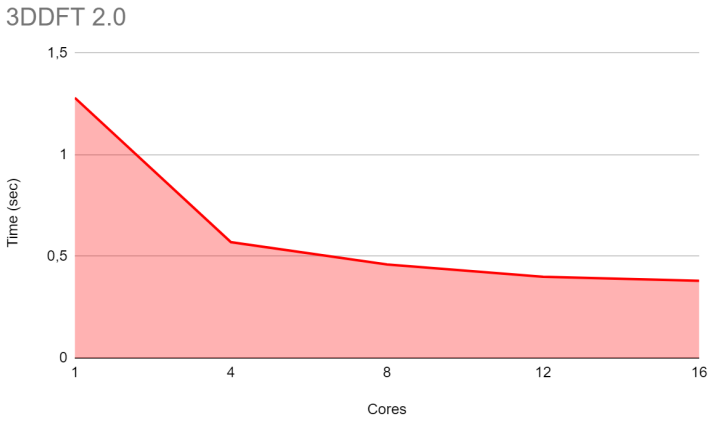
Real speedup (and T1, T12) are computed with the values given from modelfactors.

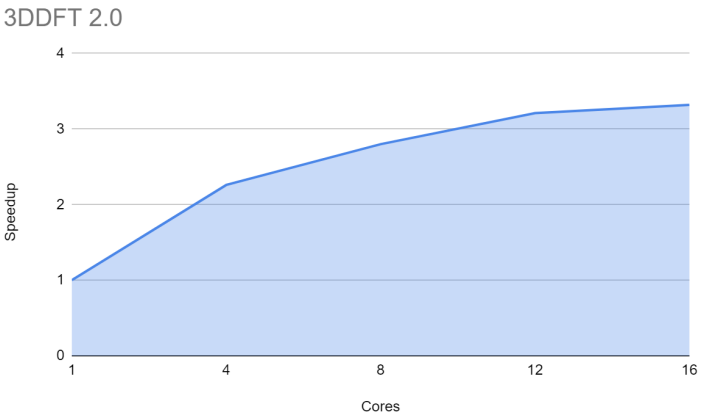
Plots initial version:





Plots version 2.0:





Plots version 3.0: