PAR Laboratory Assignment
Lab 1: Experimental setup and tools

Alvaro Moreno Ribot
Albert Escoté Alvarez

PAR 22

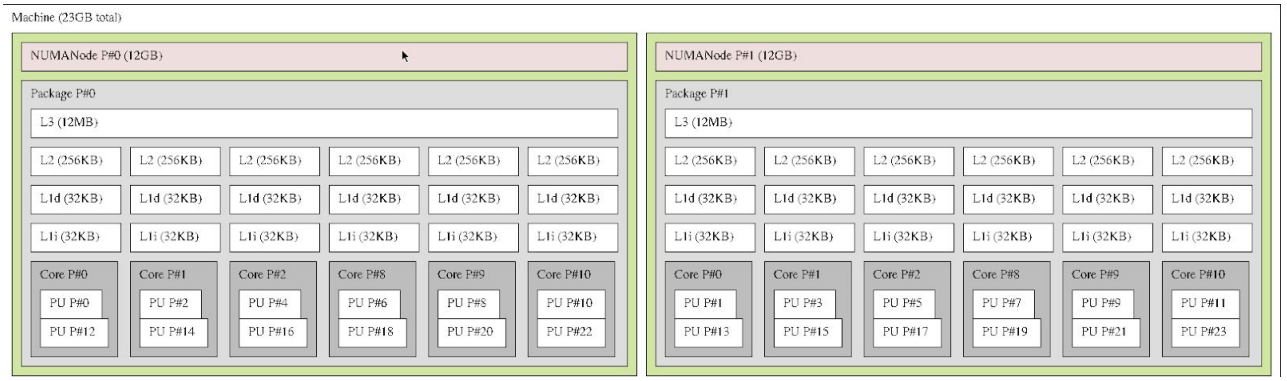
Node architecture and memory

As the provided document states, the architecture for nodes boada-1 to boada-4 is identical, so executing the command **lscpu** on boada-1 is enough to get the necessary information.

	boada-1 to boada-4
Number of sockets per node	2
Number of cores per socket	6
Number of threads per core	2
Maximum core frequency	2395 MHz
L1-I cache size (per-core)	32K
L1-D cache size (per-core)	32K
L2 cache size (per-core)	256K
Last-level cache size (per-socket)	12288K
Main memory size (per socket)	12 GB
Main memory size (per node)	24 GB

```
par2214@boada-1:~/lab1$ lscpu
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              24
On-line CPU(s) list: 0-23
Thread(s) per core:  2
Core(s) per socket:  6
Socket(s):           2
NUMA node(s):        2
Vendor ID:           GenuineIntel
CPU family:          6
Model:               44
Model name:          Intel(R) Xeon(R) CPU           E5645  @ 2.40GHz
Stepping:            2
CPU MHz:             1975.837
CPU max MHz:         2395.0000
CPU min MHz:         1596.0000
BogoMIPS:            4800.13
Virtualization:      VT-x
L1d cache:           32K
L1i cache:           32K
L2 cache:            256K
L3 cache:            12288K
NUMA node0 CPU(s):   0,2,4,6,8,10,12,14,16,18,20,22
NUMA node1 CPU(s):   1,3,5,7,9,11,13,15,17,19,21,23
Flags:               fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr
 sse sse2 ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_ts
c cpuid aperfmperf pni dtes64 monitor ds_cpl vmx smx est tm2 ssse3 cx16 xtpr pdcm pcid dca sse4_1 sse4_2 popcnt lahf_l
m pti ssbd ibrs ibpb stibp tpr_shadow vnmi flexpriority ept vpid dtherm ida arat flush_l1d
```

result execution of `lscpu`



As we can see we have, inside each node, two sockets of 12GB each and each of the sockets has 6 processing units.

Interactive vs. queued

In the next table we show the results of the execution of **pi_omp.c** program on interactive and queued execution to analyze their differences. We executed the scripts with 1, 2, 4 and 8 threads with the same number of iterations: 1.000.000.000.

#threads	Interactive				Queued			
	user	system	elapsed	%CPU	user	system	elapsed	%CPU
1	3.97	0.00	0:03.98	99%	3.94	0.00	0:03.97	99%
2	7.96	0.00	0:03.99	199%	3.94	0.01	0:02.00	197%
4	7.95	0.06	0:04.01	199%	3.99	0.00	0:01.01	392%
8	8.00	0.03	0:04.02	199%	4.17	0.00	0:00.54	772%

The result of the scripts was the user time, the system time, the elapsed time and the % of CPU used by the program. In the case of the queued execution, the result was a file named **time-pi_omp-X-boada-Y**, but the content was the same as the interactive execution. We organize that results between the different number of threads and between interactive and queued.

We can observe that in the interactive execution as more threads we have, more time is spent in user time (time related to the execution). Also the elapsed time and % of CPU used increases. For the queued execution we can observe that the user and system time remains the same approximately and the elapsed time decreases considerably. As a result of this difference, the % of CPU increases a lot.

Strong vs. weak scalability

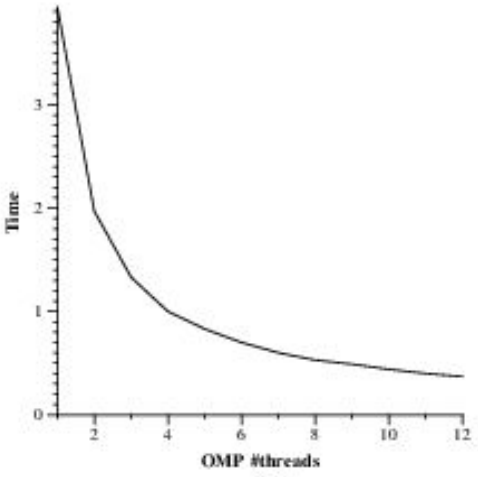
In the two following sections we are going to discuss the two different approaches to parallelism: strong and weak and we will be applying them to the pi omp.c program.

Strong Scalability

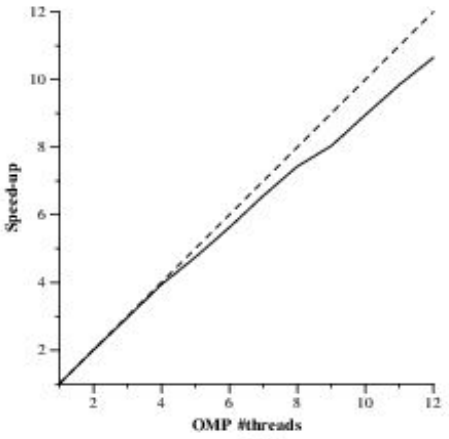
We refer to **strong scalability** when we increase the number of threads but keep a fixed problem size reducing the work that each processor has to do. The fact that programs are never infinitely parallelizable because of overheads from task creation and synchronization. This means we could reach a point where adding more processors would not change the execution time at all, as the maximum parallelization has been achieved and would result in a worse performance because of the overheads dominance.

The image below shows the plots from executing the **pi_omp.c** program on **boada-2** to **4** using the strong scalability method and we submitted it using the **submit-strong-omp.sh** script we are given.

The problem size is 1.000.000.000 iterations. I can see that, as I increase the number of threads, the time is reduced logarithmically. If I put a close eye on the last parts (10 to 12 threads) I can see that the time reduction starts to transform to a more horizontal line.



par2214
Min elapsed execution time
Tue Feb 23 17:44:23 CET 2021

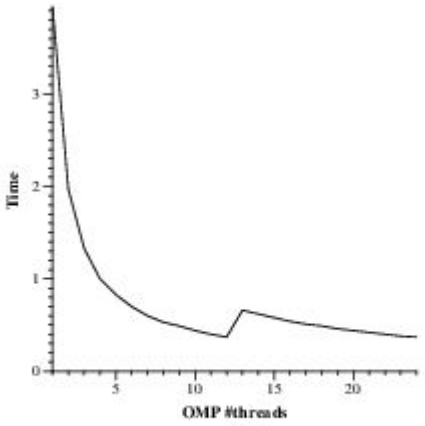


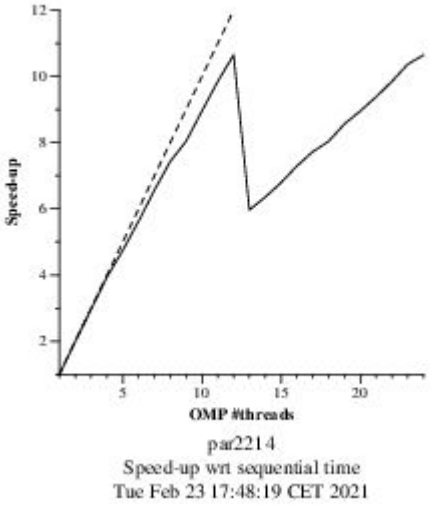
par2214
Speed-up wrt sequential time
Tue Feb 23 17:44:23 CET 2021

Now we are asked to change the **np_MAX** value from the **submit-strong-omp.sh** script from 12 threads to 24 threads.

At first glance, we can see a somewhat strange behavior. The execution time is acting the same as before from 1 to 12, but when we increase from 12 to 13 we can see that the execution time steeply jumps up.

This strange behavior is due to something called hyper-threading. With 13, 14… threads the cost of doing **hyper-threading** is higher than the benefit of it. But **if we keep adding cores**, we finally get a time reduction again.

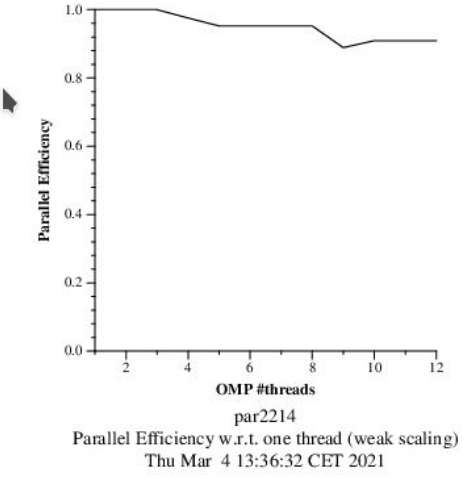




Weak Scalability

In weak scalability we will do the opposite. Now we will take advantage of the additional free computation resources we gain by parallelization to increase the problem size in a proportional way to the number of threads. We should see a more or less constant speed-up while the amount of work that is being done increases.

As we can see in the image on the right, it shows the plot for the parallel efficiency of executing **pi_omp.c** using the weak scaling strategy. We can see that, as we increase the problem size the speedup does not vary in a significant way from 1 to 4 threads. Once we reach 4 threads it starts to decrease slowly and when we reach 7 threads it starts decreasing faster.

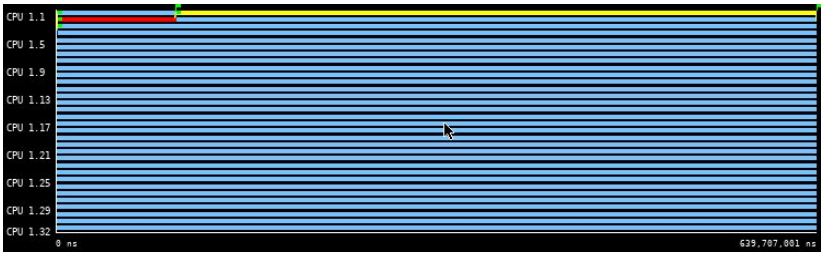


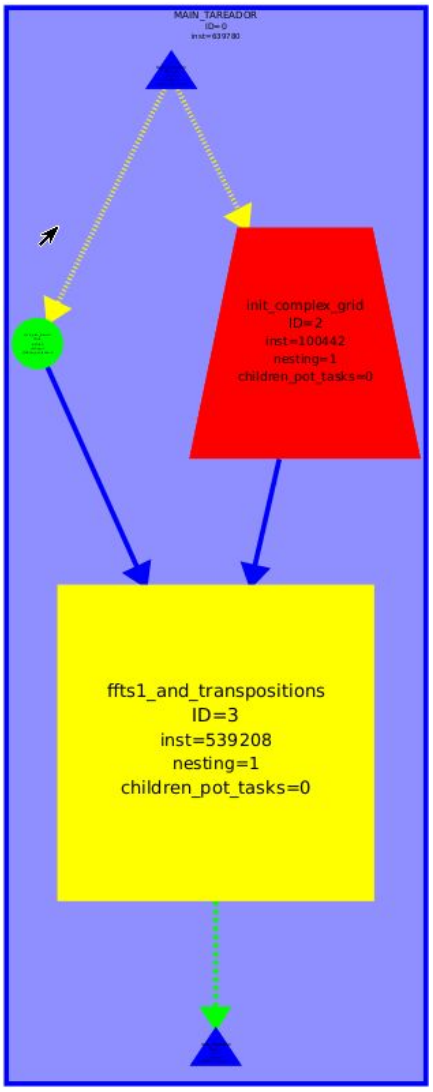
Session 2: Systematically analysing task decompositions with Tareador

In this lab session we will be exploring what Tareador can offer us. Tareador is an environment that we will be using to analyse the potential parallelism and visualize dependencies between tasks.

We are asked to modify the code we are given from the file 3dftt_tar.c to visualize and analyze different parallelization techniques.

On the image on the right, we can see the result of the initial sequential 3dftt_tar.c version we are given. We can see there are 3 tasks (apart from the creation and ending tasks) and the sizes are very different between them. This gives us a hint that we may want to parallelize the bigger tasks.





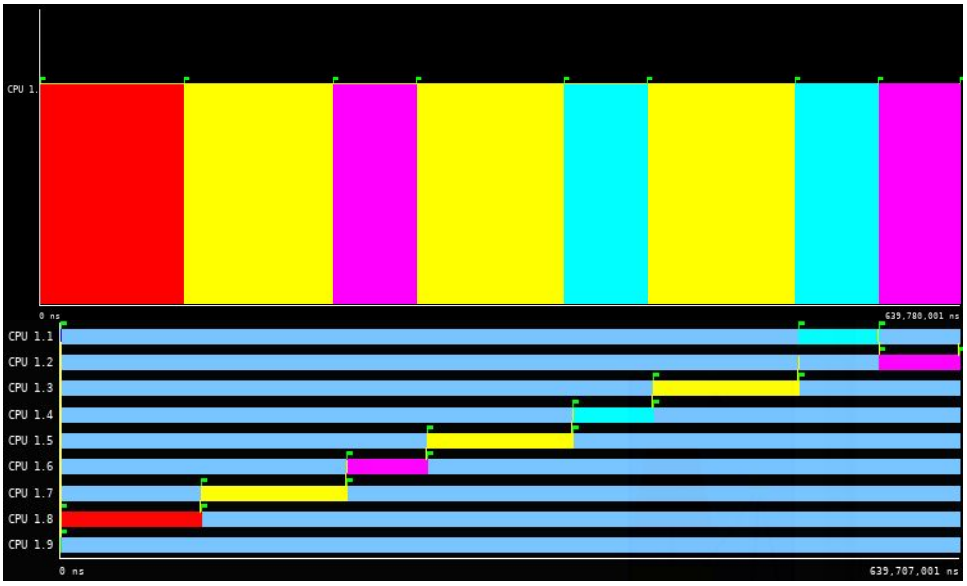
Version 1

The first version consists in replacing the task named *ffts1_and transpositions* with a sequence of finer grained tasks, one for each function invocation inside it. The modified code is the following:

```
...

tareador_start_task("ffts1_and_transpositions");
ffts1_planes(p1d, in_fftw);
tareador_end_task("ffts1_and_transpositions");

tareador_start_task("ffts1_and_transpositions_xy");
transpose_xy_planes(tmp_fftw, in_fftw);
tareador_end_task("ffts1_and_transpositions_xy");

tareador_start_task("ffts1_and_transpositions");
ffts1_planes(p1d, tmp_fftw);
tareador_end_task("ffts1_and_transpositions");

tareador_start_task("ffts1_and_transpositions_zx");
transpose_zx_planes(in_fftw, tmp_fftw);
tareador_end_task("ffts1_and_transpositions_zx");

tareador_start_task("ffts1_and_transpositions");
ffts1_planes(p1d, in_fftw);
tareador_end_task("ffts1_and_transpositions");

tareador_start_task("ffts1_and_transpositions_zx");
transpose_zx_planes(tmp_fftw, in_fftw);
tareador_end_task("ffts1_and_transpositions_zx");

tareador_start_task("ffts1_and_transpositions_xy");
transpose_xy_planes(in_fftw, tmp_fftw);
tareador_end_task("ffts1_and_transpositions_xy");

...
```

Once we have edited the code as the one on the image, we need to execute the script **`./run-tareador.sh 3dfft_tar`** and we will be able to visualize the task dependence graph like the one in the image on the right. As we can see, comparing it to the original graph; the shape that was associated with *ffts1_and_transpositions* in the initial version has now been divided into several other shapes which represent more granularity.

The two images below represent the timelines for the execution of the code above. We can see that the execution is still sequential as we haven't applied any parallelism yet.

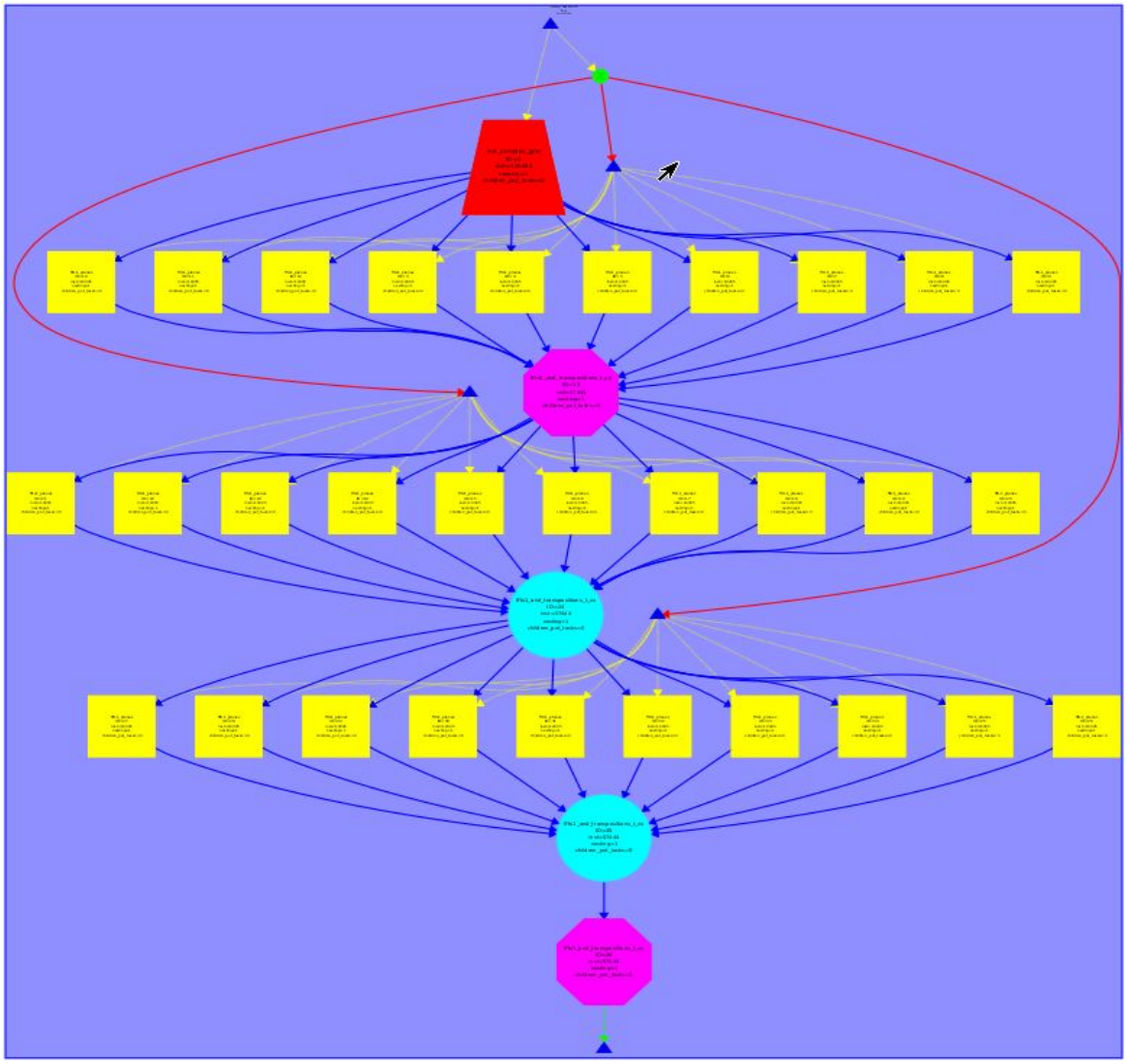


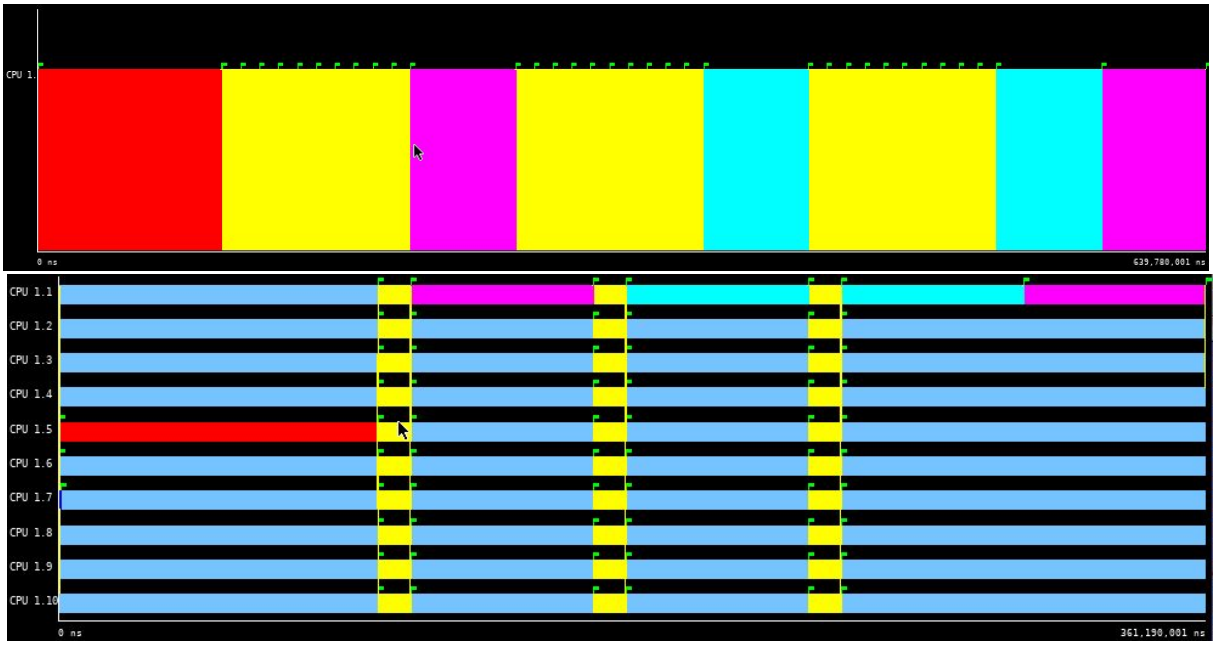
Version 2

The second version **keeps the changes made in version one** and we also will be replacing the definition of tasks associated with function invocations of *ffts1_planes* with fine-grained tasks associated with individual iterations of the k loop defined inside the function body . The changes that have been made in the code for this version are the following ones:

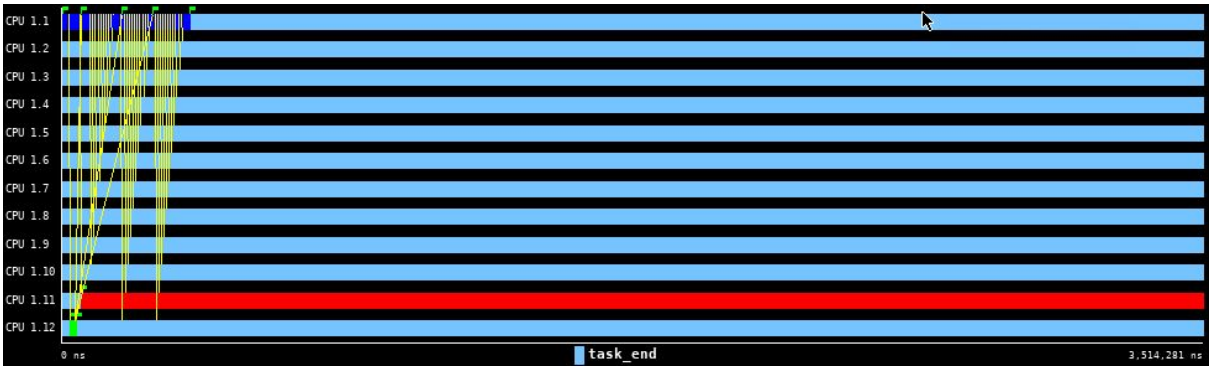
```
void ffts1_planes(fftwf_plan p1d, fftwf_complex in_fftw[][N][N]) {
    int k,j;

    for (k=0; k<N; k++) {
     tareador_start_task("ffts1_planes");
     for (j=0; j<N; j++) {
       fftwf_execute_dft( p1d, (fftwf_complex *)in_fftw[k][j][0], (fftwf_complex *)in_fftw[k][j][0]);
     }
     tareador_end_task("ffts1_planes");
    }
}
```

As we can see in the outputed dependency graph given by the Tareador, it has changed and there are several more shapes. The task that previously was formed by a single big shape tagged as *ffts1_planes* has now been divided into several more and can be executed in parallel as there are no dependencies on the calculations inside the function.





The first timeline is the execution of this same program with only one thread. The second one is the execution of this program with 10 threads, which we can call Tinf as there is where we reach the maximum parallelism. We can see that all the tasks created from the *ffst_planes* (yellow) loop are executed in parallel.



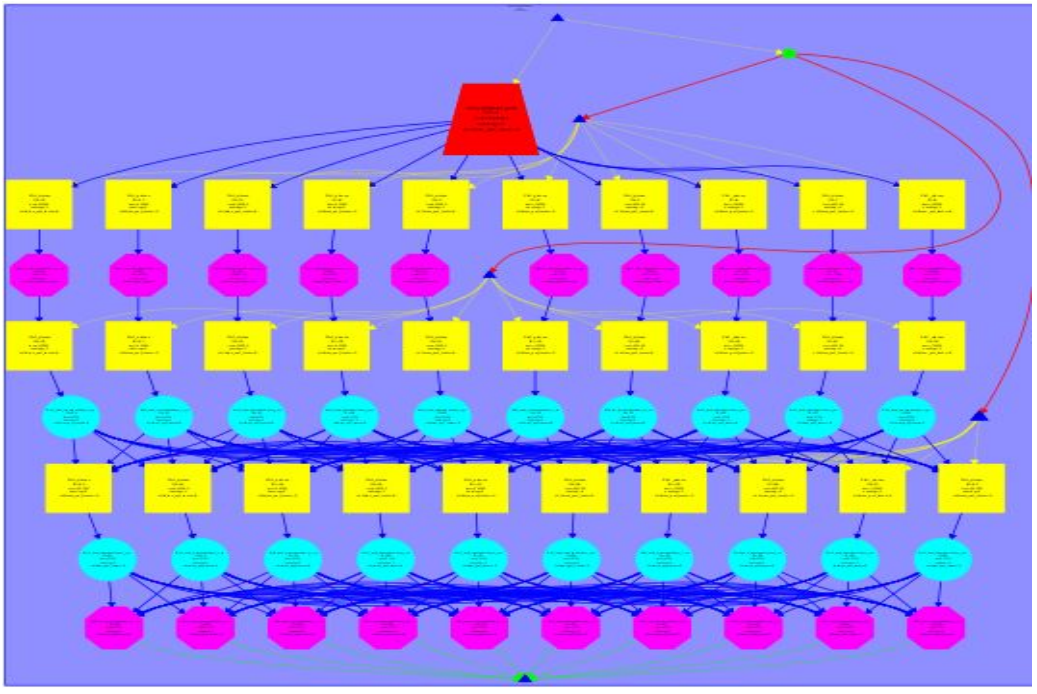
If we zoom in, we will be able to see that there is some work that is also done in parallel on the program start. We think that these small tasks are related to task creation.

Version 3

The third version also keeps the changes made in the second one and consists in replacing the definition of tasks associated to function invocations of *transpose_xy_planes* and *transpose_zx_planes* with fine-grained tasks inside the corresponding body functions and associated to individual iterations of the k loop, similarly as it was made in the second version.

```
1 void transpose_xy_planes(fftwf_complex  tmp_fftw[][N][N], fftwf_complex in_fftw[][N][N]) {
2     int k,j,i;
3
4     for (k=0; k<N; k++) {
5      tareador_start_task("ffts1_and_transpositions_t_xy");
6      for (j=0; j<N; j++) {
7        for (i=0; i<N; i++)
8        {
9          tmp_fftw[k][i][j][0] = in_fftw[k][j][i][0];
10         tmp_fftw[k][i][j][1] = in_fftw[k][j][i][1];
11       }
12     }
13    tareador_end_task("ffts1_and_transpositions_t_xy");
14    }
15 }
16
17 void transpose_zx_planes(fftwf_complex in_fftw[][N][N], fftwf_complex tmp_fftw[][N][N]) {
18    int k, j, i;
19
20    for (k=0; k<N; k++) {
21     tareador_start_task("ffts1_and_transpositions_t_zx");
22     for (j=0; j<N; j++) {
23       for (i=0; i<N; i++)
24       {
25         in_fftw[i][j][k][0] = tmp_fftw[k][j][i][0];
26         in_fftw[i][j][k][1] = tmp_fftw[k][j][i][1];
27       }
28     }
29     tareador_end_task("ffts1_and_transpositions_t_zx");
30    }
31 }
```

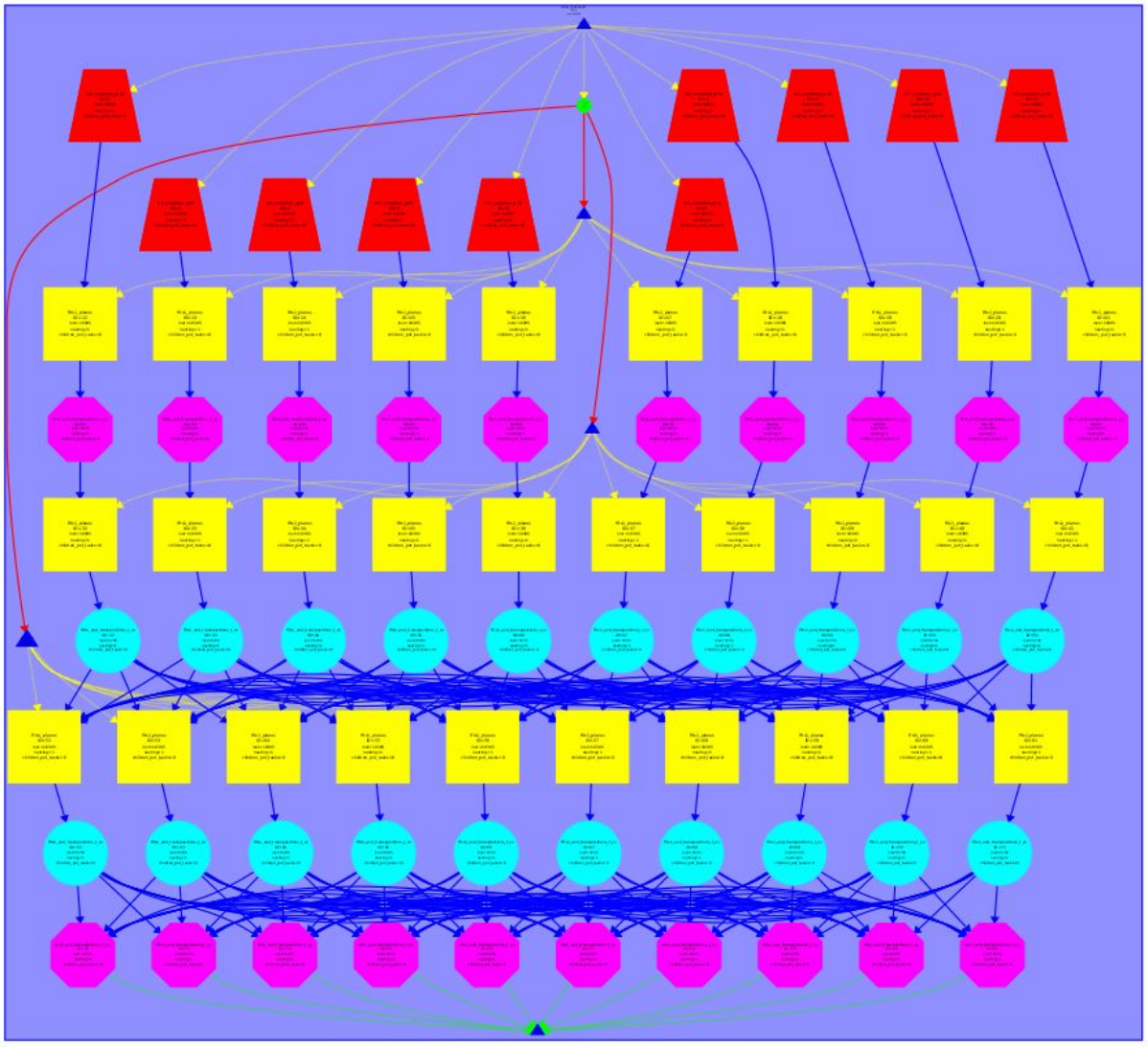
Like in the previous cases, we can now see in the dependency graph our results and the level of granularity we are getting:

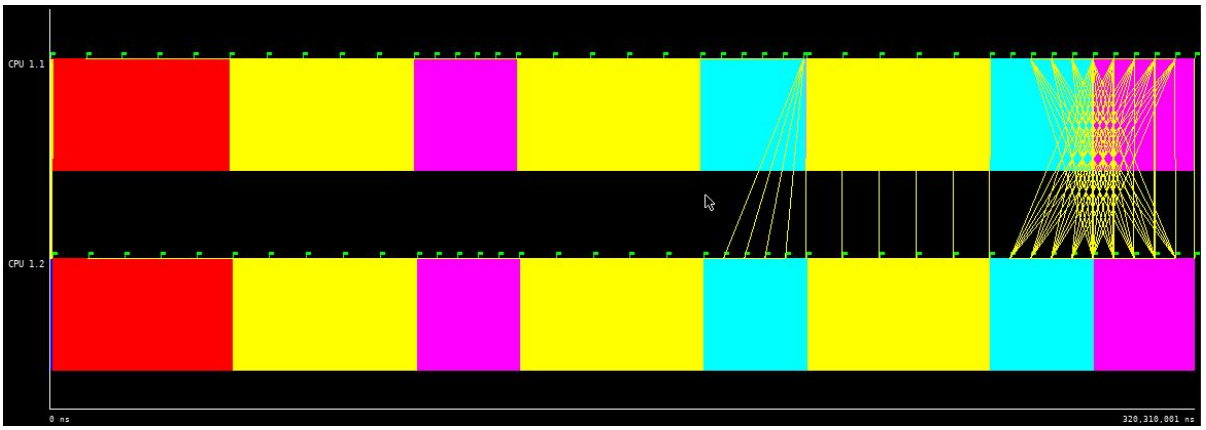


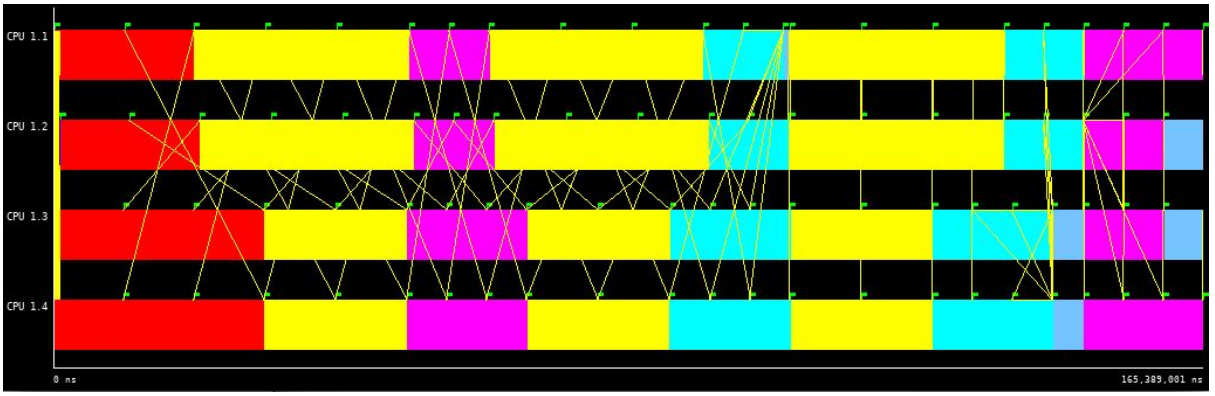
Version 4

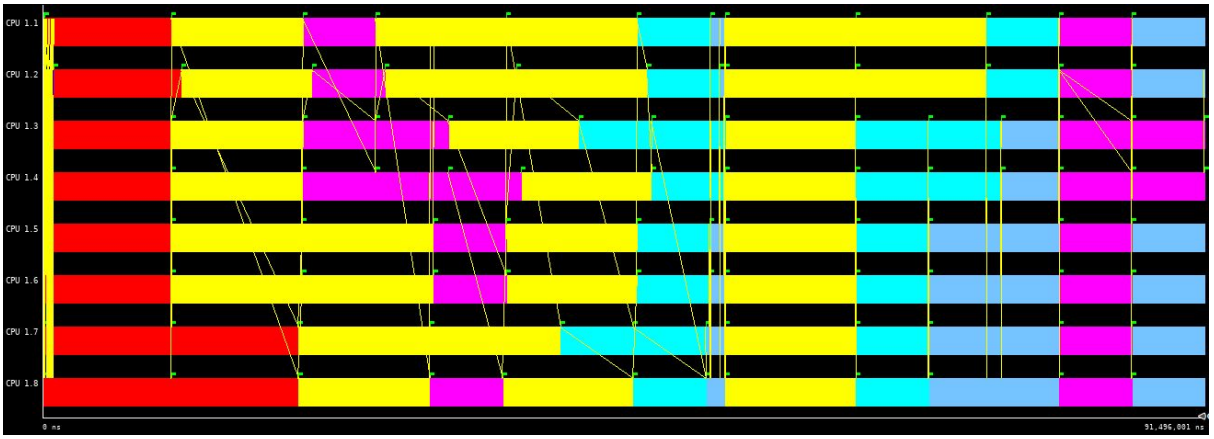
The fourth version consists of replacing the definition of tasks associated with function invocations *init_complex_grid* with finer-grained tasks inside the corresponding body functions associated to individual iterations of the k loop, like it was made in the previous version.

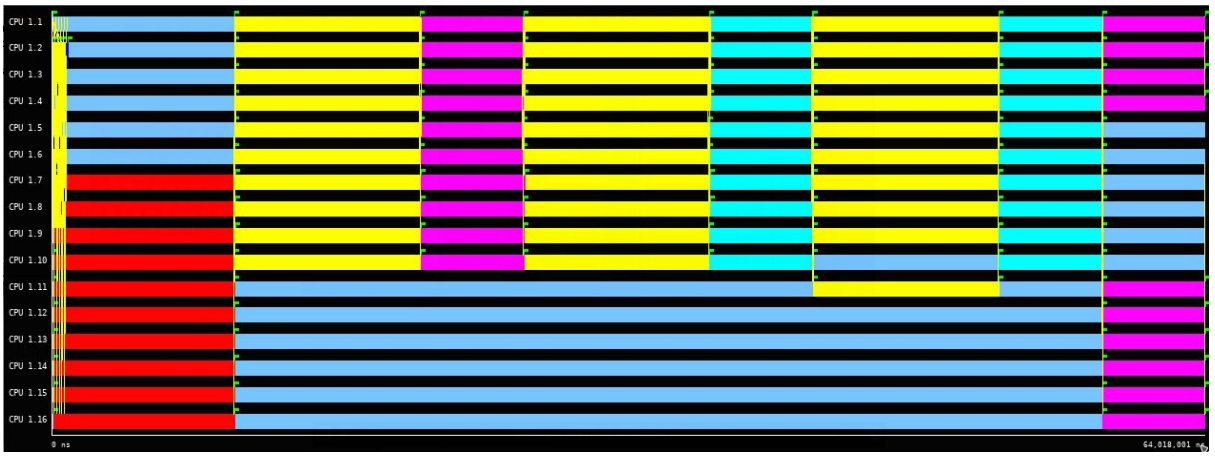
```
1 void init_complex_grid(fftwf_complex in_fftw[][N][N]) {
2   int k,j,i;
3
4   for (k = 0; k < N; k++) {
5     tareador_start_task("init_complex_grid");
6     for (j = 0; j < N; j++) {
7       for (i = 0; i < N; i++)
8       {
9         in_fftw[k][j][i][0] = (float) (sin(M_PI*((float)i)/64.0)+sin(M_PI*((float)i)/32.0)+sin(M_PI*
  ((float)i/16.0)));
10        in_fftw[k][j][i][1] = 0;
11 #if TEST
12        out_fftw[k][j][i][0]= in_fftw[k][j][i][0];
13        out_fftw[k][j][i][1]= in_fftw[k][j][i][1];
14 #endif
15      }
16    }
17  tareador_end_task("init_complex_grid");
18  }
19 }
```

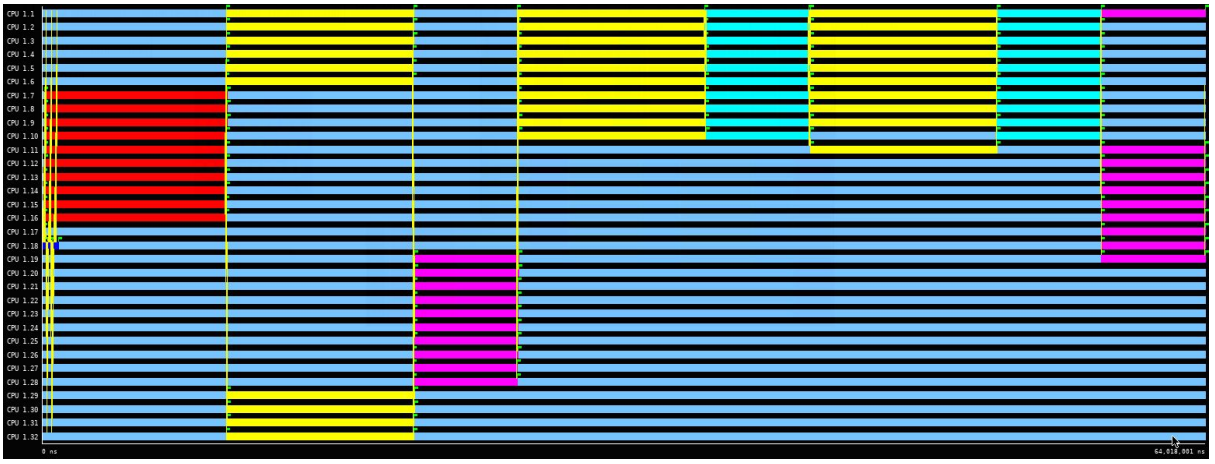



CPU 1.1
CPU 1.2
0 ns
320,310,001 ns


CPU 1.1
CPU 1.2
CPU 1.3
CPU 1.4
0 ns
165,309,001 ns

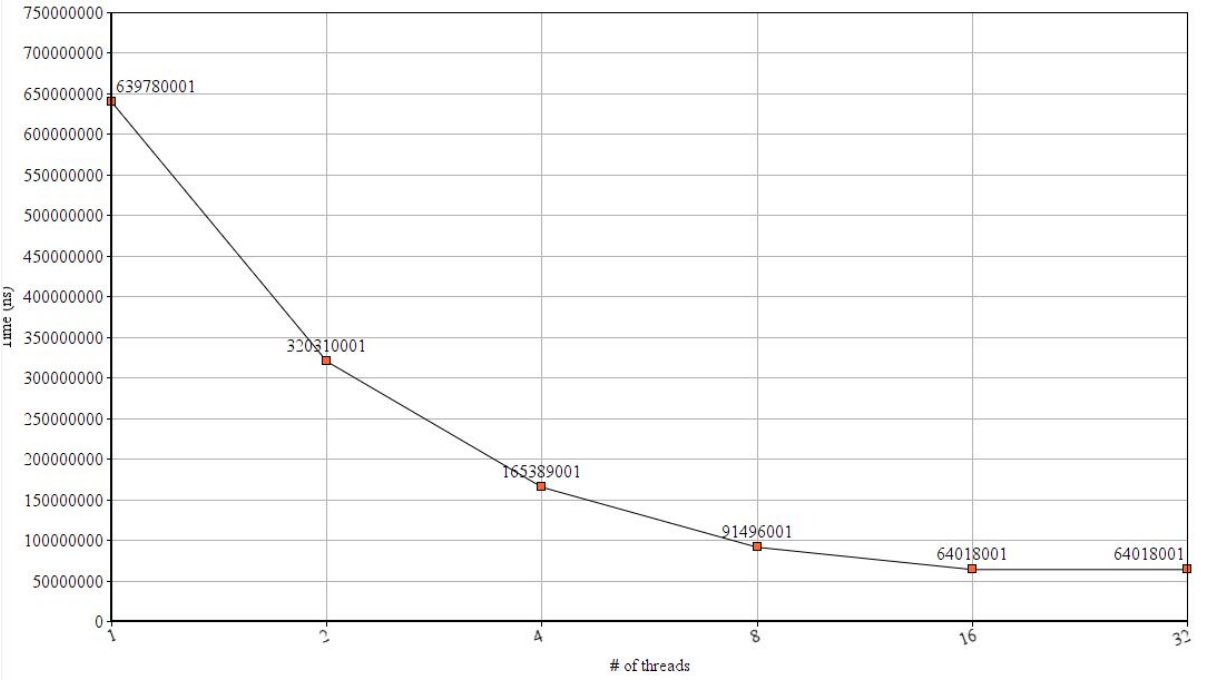

CPU 1.1
CPU 1.2
CPU 1.3
CPU 1.4
CPU 1.5
CPU 1.6
CPU 1.7
CPU 1.8
0 ns
91,496,001 ns


CPU 1.1
CPU 1.2
CPU 1.3
CPU 1.4
CPU 1.5
CPU 1.6
CPU 1.7
CPU 1.8
CPU 1.9
CPU 1.10
CPU 1.11
CPU 1.12
CPU 1.13
CPU 1.14
CPU 1.15
CPU 1.16
0 ns
64,010,001 ns


CPU 1.1
CPU 1.2
CPU 1.3
CPU 1.4
CPU 1.5
CPU 1.6
CPU 1.7
CPU 1.8
CPU 1.9
CPU 1.10
CPU 1.11
CPU 1.12
CPU 1.13
CPU 1.14
CPU 1.15
CPU 1.16
CPU 1.17
CPU 1.18
CPU 1.19
CPU 1.20
CPU 1.21
CPU 1.22
CPU 1.23
CPU 1.24
CPU 1.25
CPU 1.26
CPU 1.27
CPU 1.28
CPU 1.29
CPU 1.30
CPU 1.31
CPU 1.32
0 ns
64,010,001 ns

These images above were the timelines for the execution of this parallel program for 2, 4, 8, 16 and 32 threads respectively. We can see that, the more threads we add the more the execution time is shrunken. The last image is the execution time for 11 threads, which is the number of threads from which it makes no effect if we add more threads (In fact, it would be 10). We can see the table for this metric and plot below. Note that as we keep adding more threads the time reduction is not linear, instead it has a reverse logarithmic growth as we will need more task synchronization, and more task overhead will be produced as we keep adding more threads.

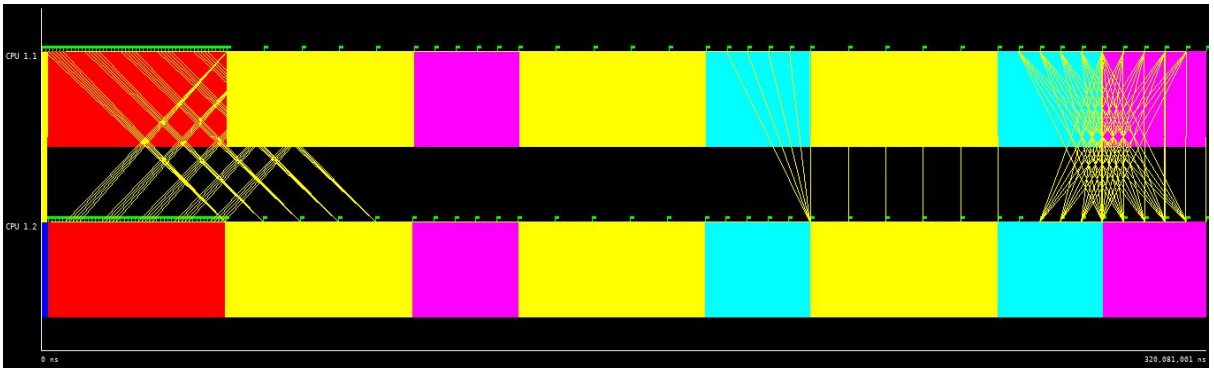
#threads	**time**
1	639 780 001 ns
2	320 310 001 ns
4	165 389 001 ns
8	91 496 001 ns
16	64 018 001 ns
32	64 018 001 ns

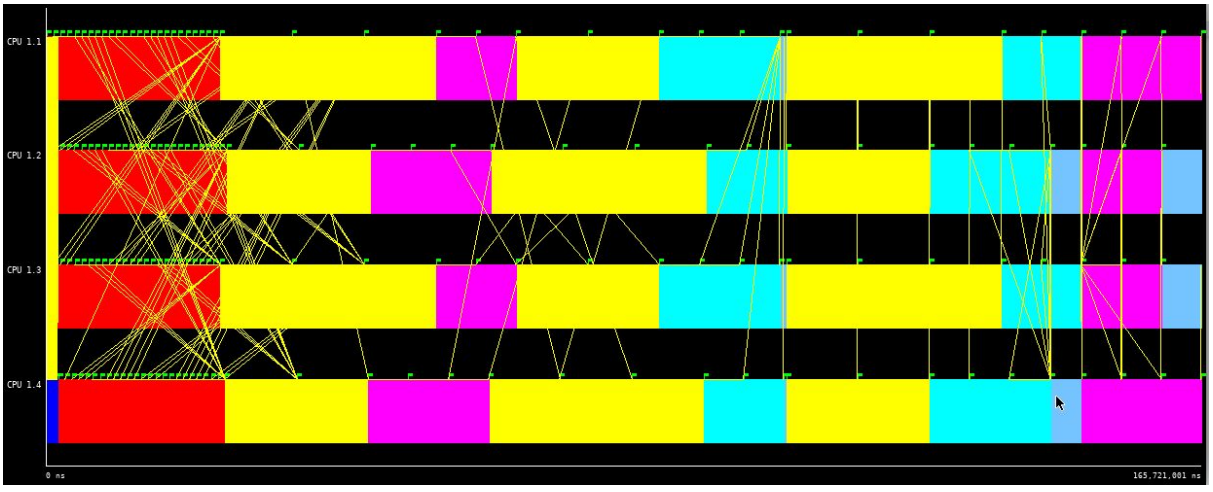


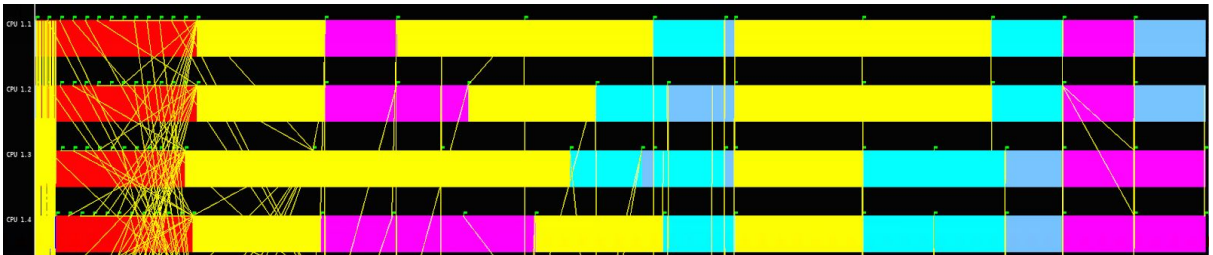
Version 5

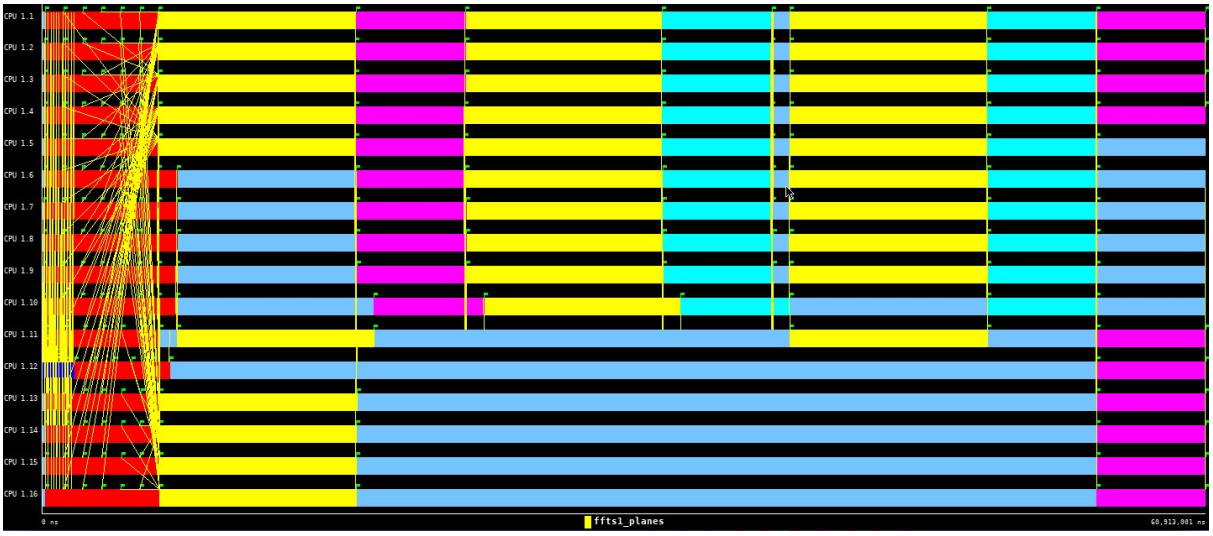
This is the final version where we will get even more finer-grained tasks. In order to continue we observed the figure given by Tareador that corresponds to the fourth version of the code. We deepened the tasks to the inner loop ***j*** (inside the ***k*** loop) of the corresponding function.

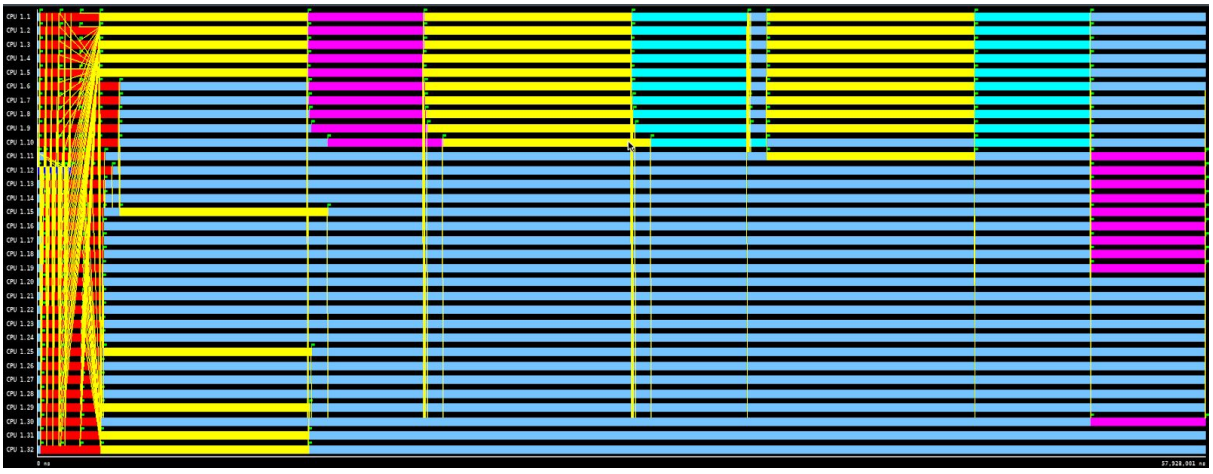
```
1 void init_complex_grid(fftwf_complex in_fftw[][N][N]) {
2   int k,j,i;
3
4   for (k = 0; k < N; k++) {
5     for (j = 0; j < N; j++) {
6       tareador_start_task("init_complex_grid");
7       for (i = 0; i < N; i++)
8       {
9         in_fftw[k][j][i][0] = (float) (sin(M_PI*((float)i)/64.0)+sin(M_PI*((float)i)/32.0)+sin(M_PI*
  ((float)i)/16.0)));
10        in_fftw[k][j][i][1] = 0;
11 #if TEST
12        out_fftw[k][j][i][0]= in_fftw[k][j][i][0];
13        out_fftw[k][j][i][1]= in_fftw[k][j][i][1];
14 #endif
15      }
16    tareador_end_task("init_complex_grid");
17    }
18  }
19 }
```





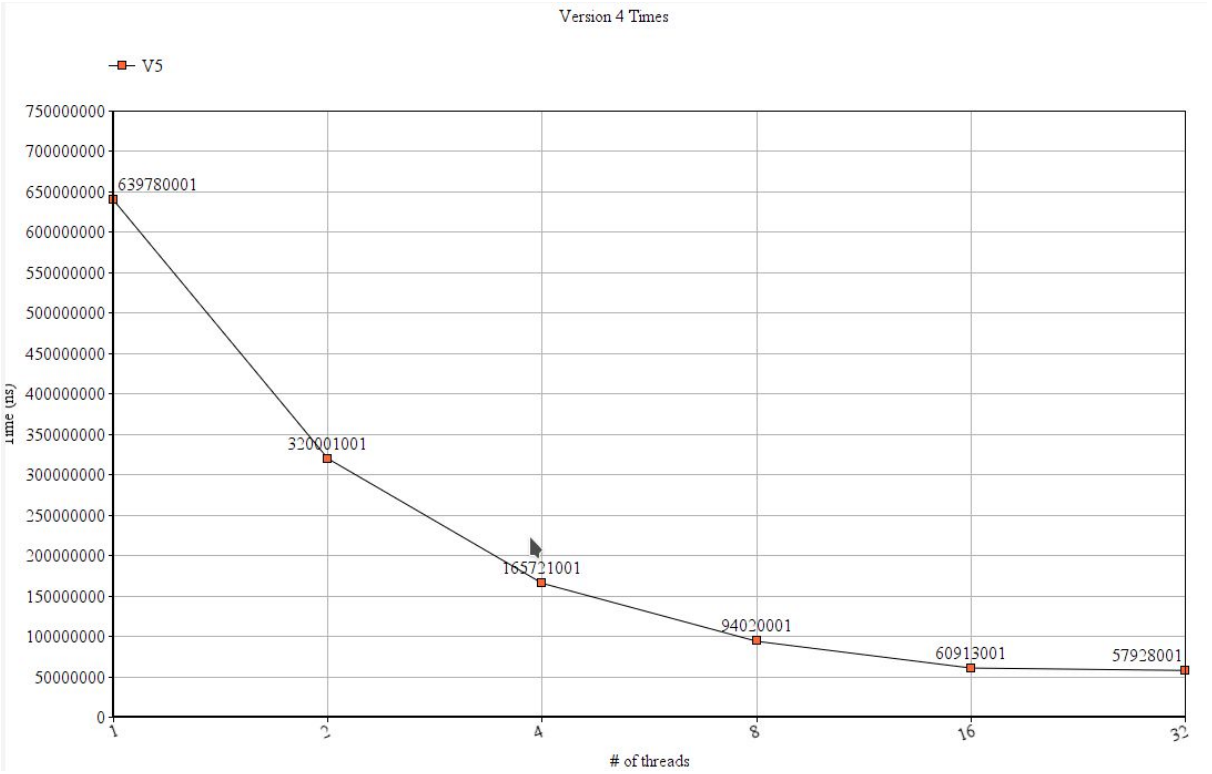






Here we have the same information and plots above and below as the previous version.

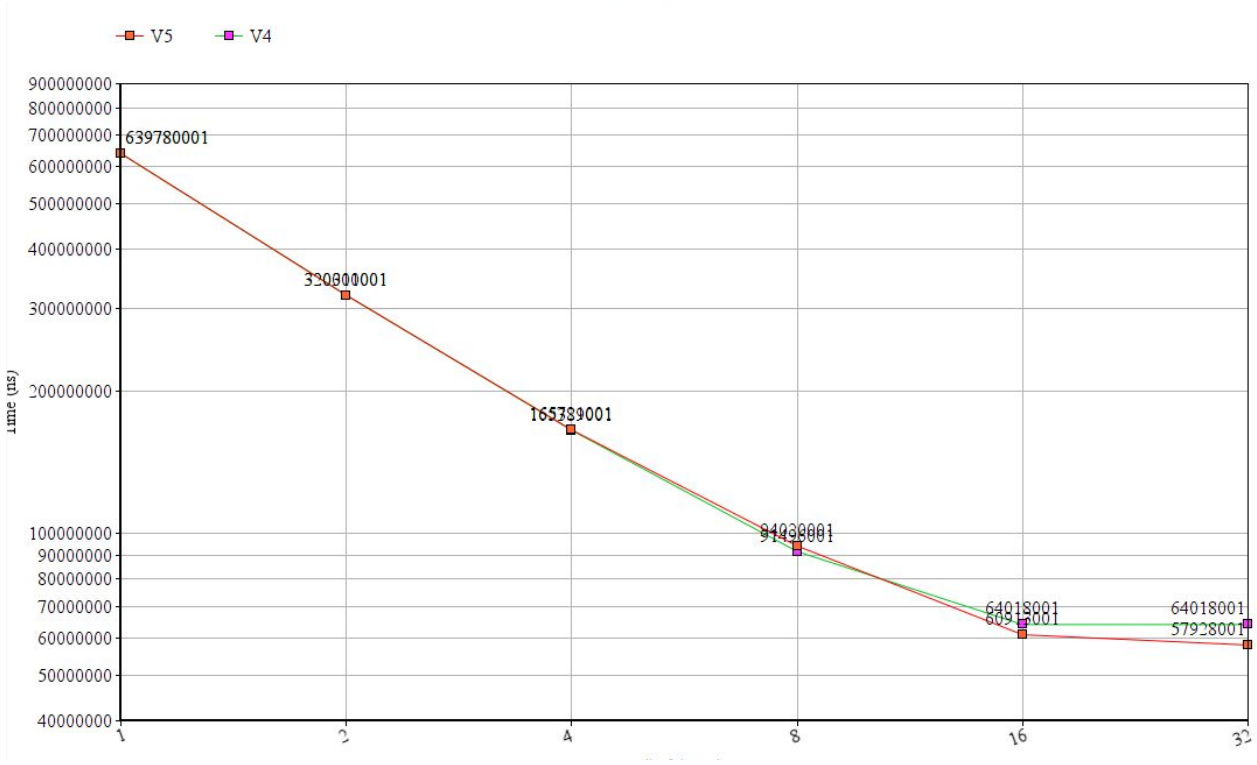
#threads	**time**
1	639 780 001 ns
2	320 001 001 ns
4	165 721 001 ns
8	94 020 001 ns
16	60 913 001 ns
32	57 928 001 ns



Linear scale Version 5

The following plot is comparing the execution time reduction we get from Version 4 in comparison to Version 5. We can see that, for a number of threads up to 16 we can't see any big differences but as we keep adding threads up to 32 (the maximum we are asked to try) we see that the performance starts getting slightly better for Version 5. This is because of the finer-grained tasks we have on version 5, where a bigger number of threads will also allow us to have a better performance.

We can also see that, for 8 processors, Version 4 is getting a better performance than Version 5. The reason for this could be justified with the time being focused for making the finer-grained tasks work together because of the overhead.



Logarithmic scale; V5 vs V4.

Conclusions

	T1	Tinf	Parallelism
seq	639 780 001 ns	639 707 001 ns	1
v1	639 780 001 ns	639 707 001 ns	1
v2	639 780 001 ns	361 190 001 ns	1.7713
v3	639 780 001 ns	154 354 001 ns	4.14488
v4	639 780 001 ns	64 018 001 ns	9.994
v5	639 780 001 ns	57 928 001 ns	11.044

The table above shows the different execution times for 1 to ->infinity processors and the parallelism we get with each version. We can see that in all the versions except for the first ones, we increase the parallelism.

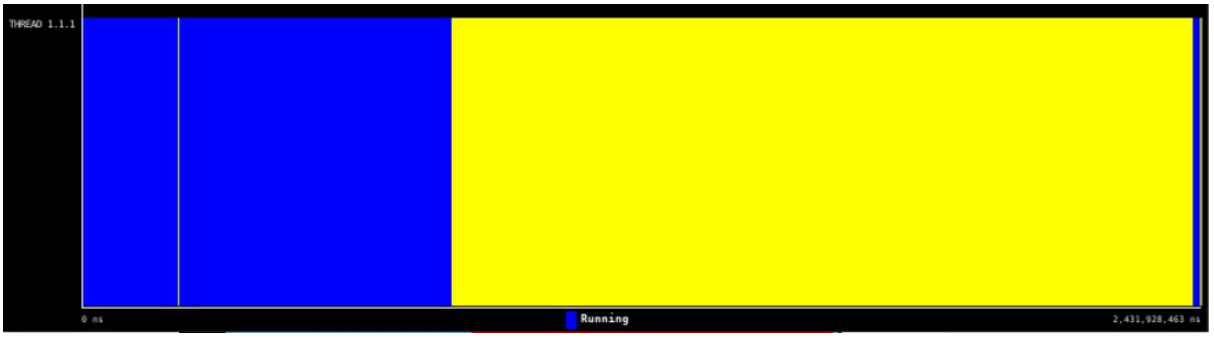
Session 3: Understanding the execution of OpenMP programs

Initial Version

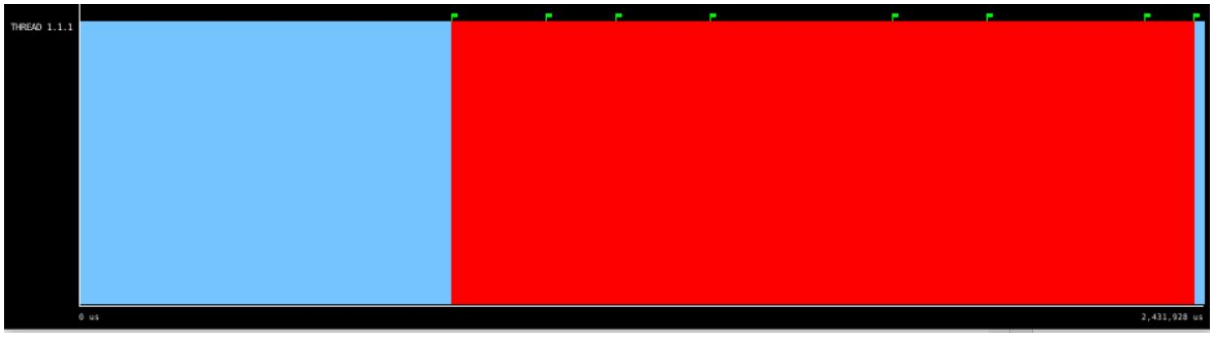
This initial version is the one which has been given to us by the PAR professors and it was partially parallelized. We can see that the for loop inside the *init_complex_grid* function is not parallelized. This will allow us to expand the parallel fraction in the later versions.

However, we still have a good improvement from using 8 threads instead of 1 as the execution time has been almost halved. In fact, due to the low parallel fraction of this version, ideal speed-up with infinite processors is 2.12 while speed-up with 8 is 2.07, which is pretty close to the limit.

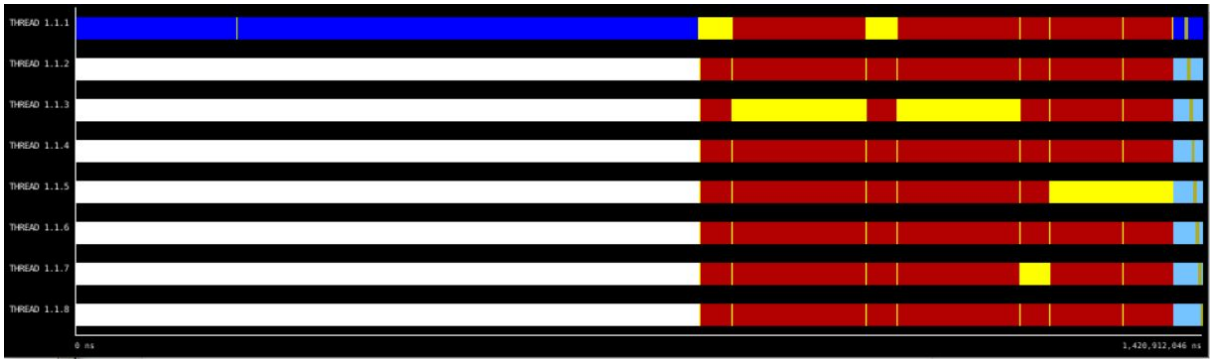
We can appreciate in the plot below how 8 threads have actually the best performance, and if we keep increasing the number of threads then overhead starts to make speed-up slightly decay.



Timeline of 3dfft_omp.c with one thread (similar to Tseq): 2 431 928 463 ns

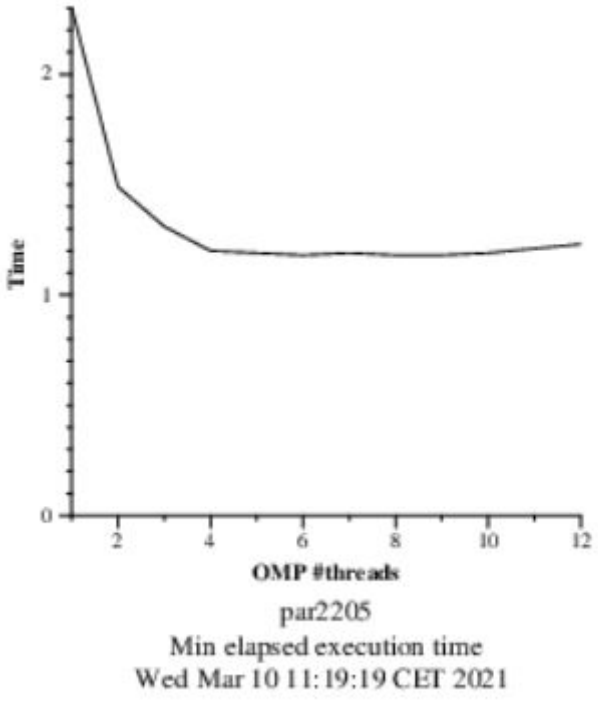


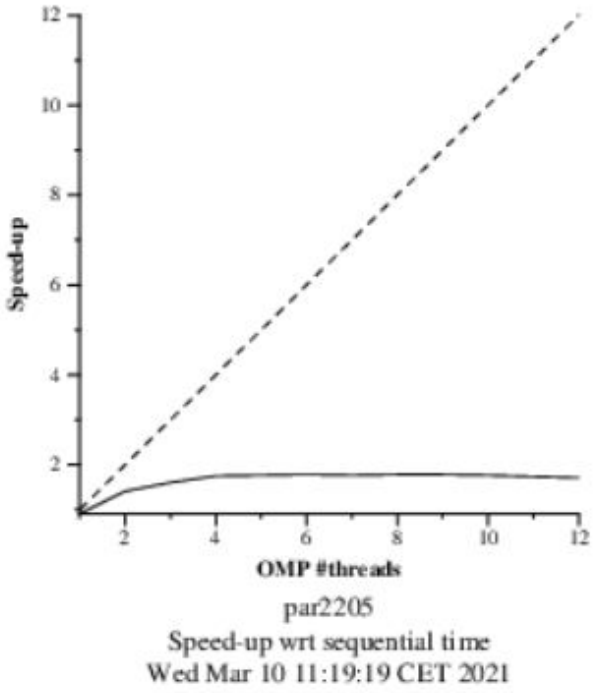
Parallel constructs of the timeline of 3dfft_omp.c with one thread (similar to Tpar): 2 431 928 μs



Timeline of 3dfft_omp.c with eight threads (similar to Tseq): 1 420 912 046 ns

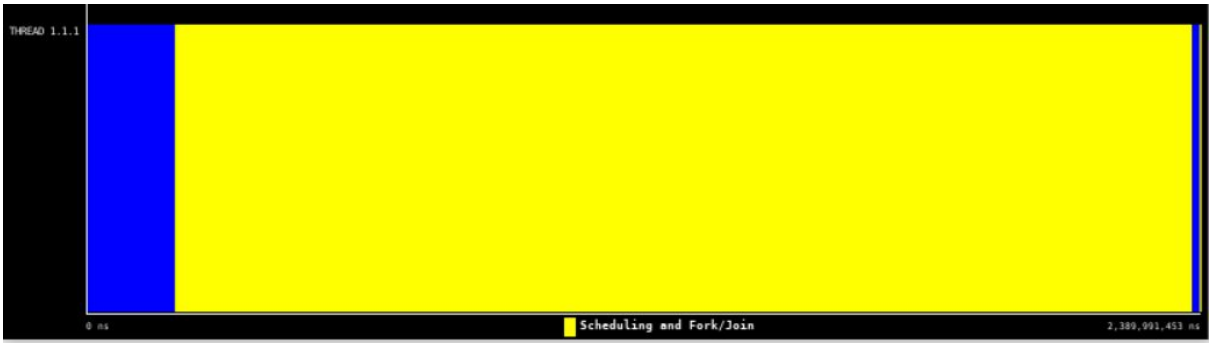
Parallel constructs of the timeline of 3dfft_omp.c with eight threads (similar to Tpar): 1 420 912 μs



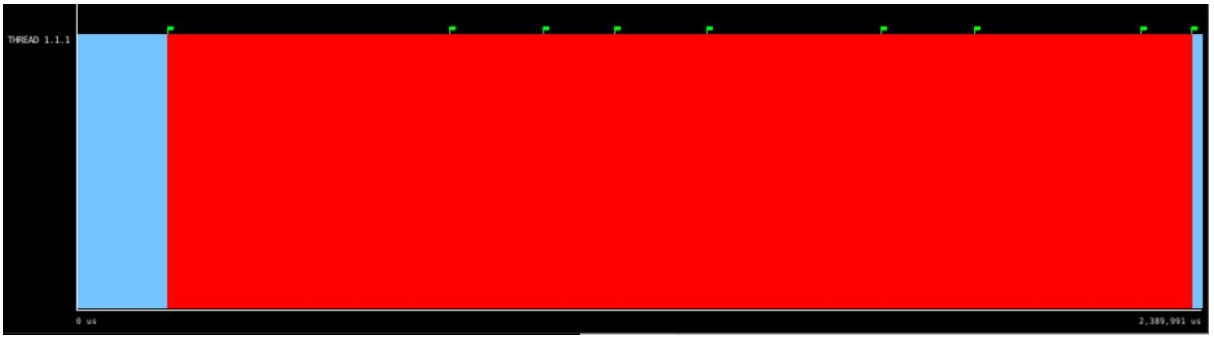


Improved ϕ

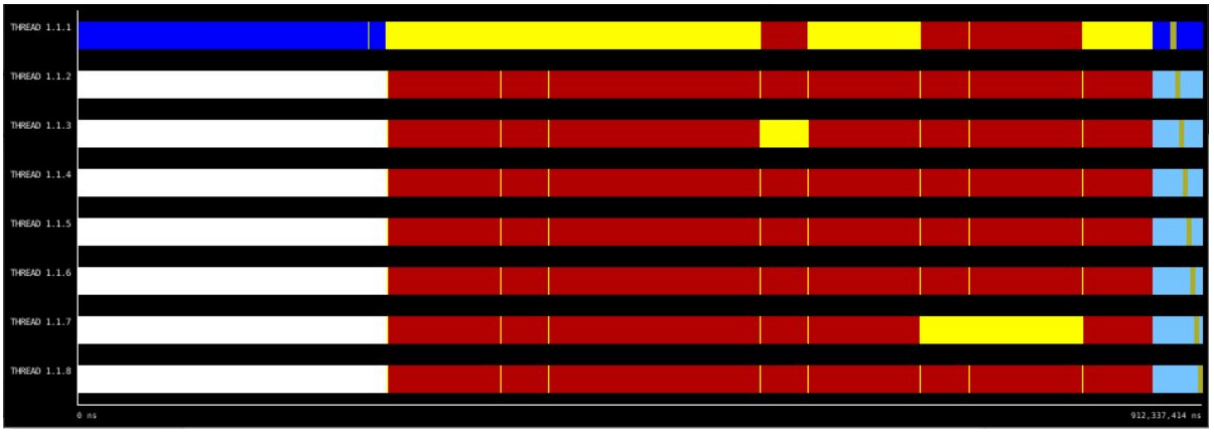
Now we uncommented a line that was inside the *init_complex_grid* function in order to allow parallelization in the *for* loop. This has increased the parallel fraction notably.



Timeline of 3dfft_omp.c with improved ϕ with one thread (similar to Tseq): 2 389 991 453 ns

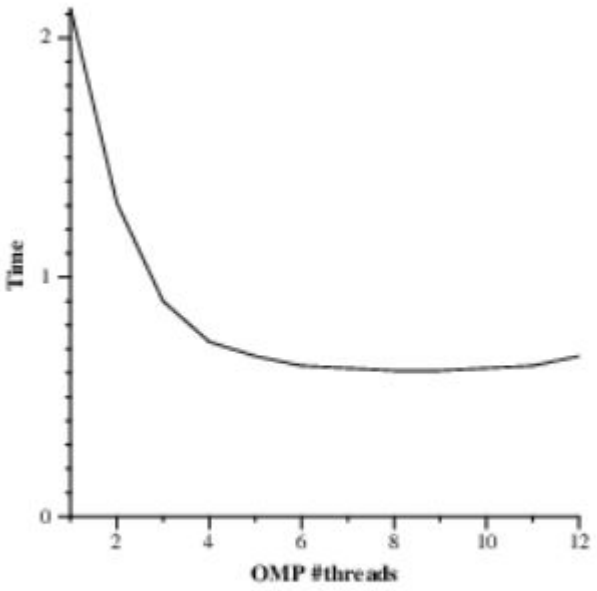


Parallel constructs of the timeline of 3dfft_omp.c with improved ϕ with eight threads (similar to Tpar): 2 380 991 μs

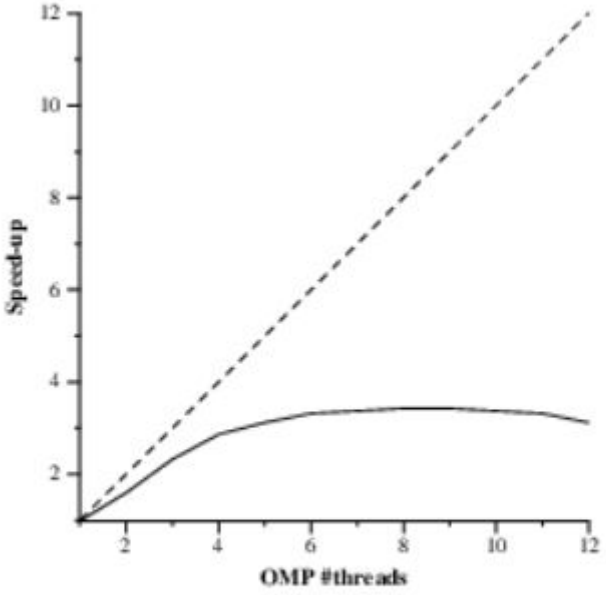


Timeline of 3dfft_omp.c with improved ϕ with eight threads (similar to Tseq): 912 337 414 ns

Parallel constructs of the timeline of 3dfft_omp.c with improved ϕ with eight threads (similar to Tpar): 912 337 μs



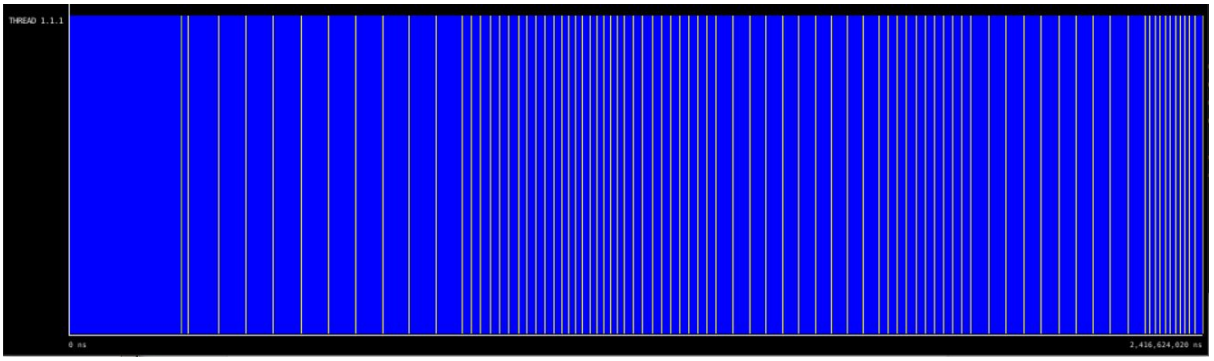
par2205
Min elapsed execution time
Wed Mar 10 17:08:17 CET 2021



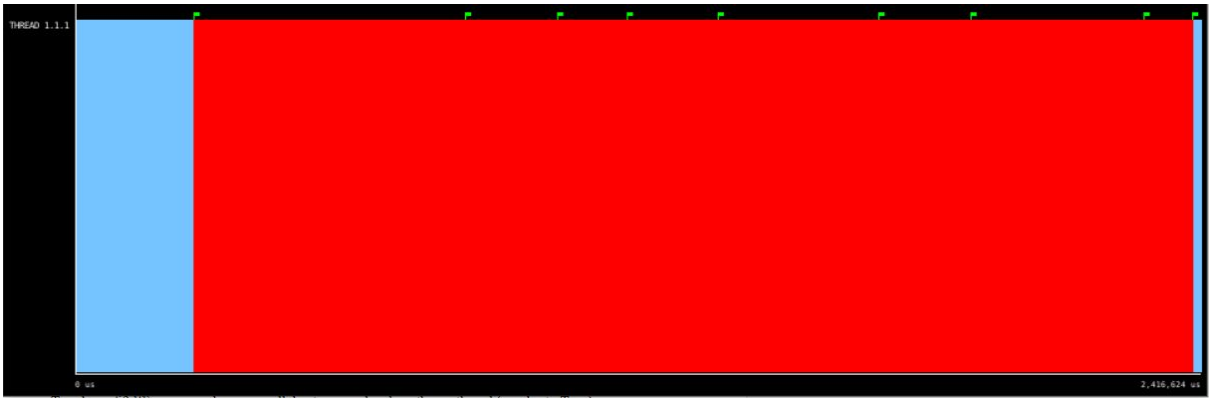
par2205
Speed-up wrt sequential time
Wed Mar 10 17:08:17 CET 2021

Reducing parallelisation overheads

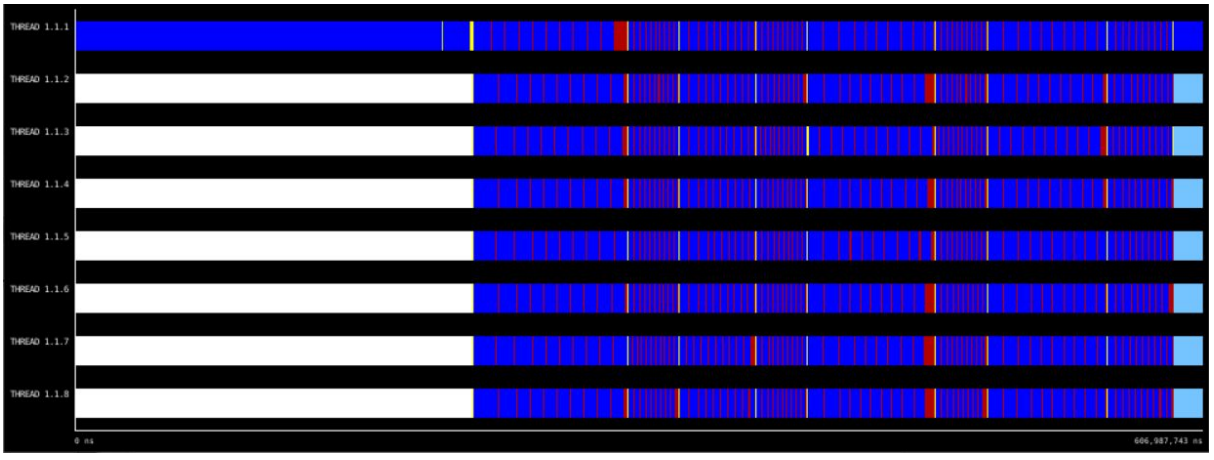
This is the last version where we moved the ***#pragma omp for*** directives one line above in order to also include the upper *for* loop. This will increase the tasks granularity and should reduce the overhead caused by parallelisation.



Timeline of 3dfft_omp.c reducing parallelisation overheads with one thread (similar to Tseq):
2 416 624 020 ns

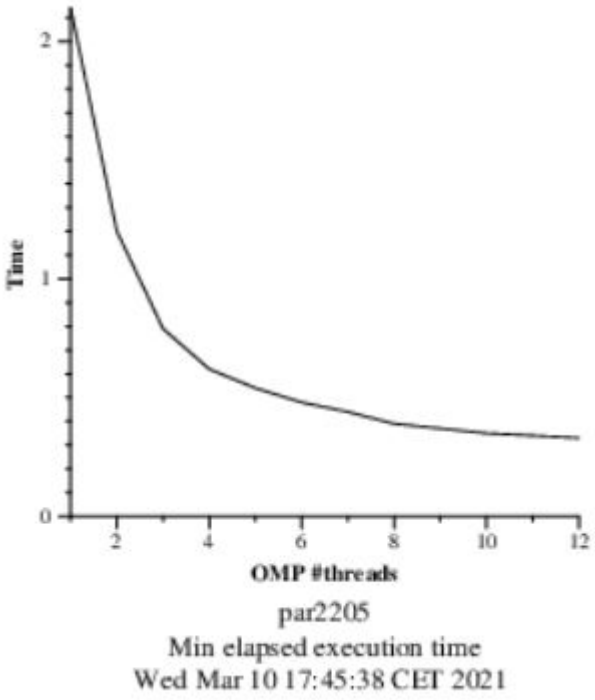


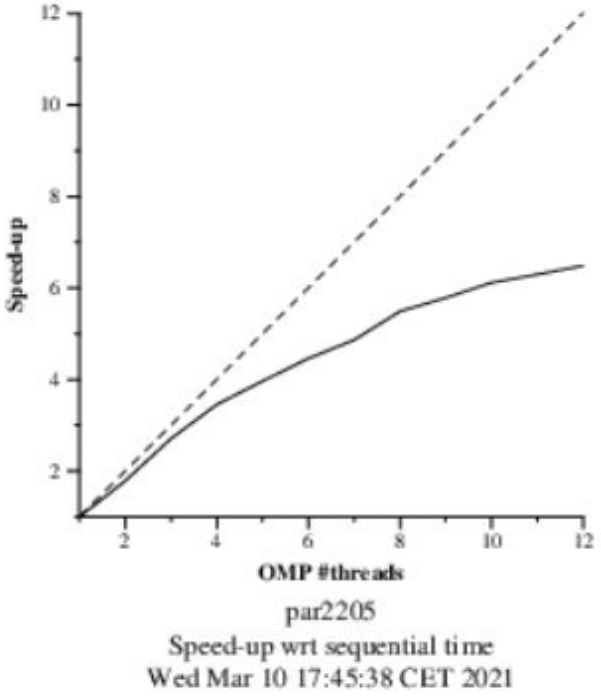
Parallel constructs of the timeline of 3dfft_omp.c reducing parallelisation overheads with one thread (similar to Tpar): 2 416 624 μs



Timeline of 3dfft_omp.c reducing parallelisation overheads with eight threads (similar to Tseq):
606 987 743 ns

Parallel constructs of the timeline of 3dfft_omp.c reducing parallelisation overheads with eight threads (similar to Tpar): 606 987 μs





$$T_x = T_{seq} + T_{par}$$
$$\phi = T_{par} \div (T_{seq} + T_{par})$$
$$S_\infty = 1 \div (1 - \phi)$$

* These results don't have any logic as it should be decreasing instead of increasing...

Version	ϕ	S_∞	T_1	T_8	S_8
initial version in 3dfft_omp.c	0.37	1.59	2 431 928 463 ns	1 420 912 046 ns	1.7
new version with improved ϕ	0.276	1.38	2 389 991 453 ns	912 337 414 ns	2.6
final version with reduced parallelisation overheads	0.2	1.248	2 416 624 020 ns	606 987 743 ns	4.00