PAR Laboratory Assignment
Lab 1: Experimental setup and tools

PAR2303
Joan Manuel Ramos Refusta
Àlex Aguilera Martínez

Node architecture and memory

Boada is a cluster divided in 8 nodes, each of them with different architecture, apart from nodes 1-4, and with different functions. Using "--of fig map.fig" option of lstopo, we could obtain the following information for the nodes Boada 1-4. Due to the students have only access to boada-1, we had to use queues executing the submit-arch.sh shell script in order to obtain the information of the rest nodes.

	Boada 1-4	Boada 5	Boada 6
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	2600MHz	1700MHz
L1-I cache size (per-core)	32k	32k	32k
L1-D cache size (per-core)	32k	32k	32k
L2 cache size (per-core)	256k	256k	256k
Last-level cache size (per-socket)	12288k	15360k	20480k
Main memory size (per socket)	23GB	63GB	31GB
Main memory size (per node)	12GB	31GB	16GB

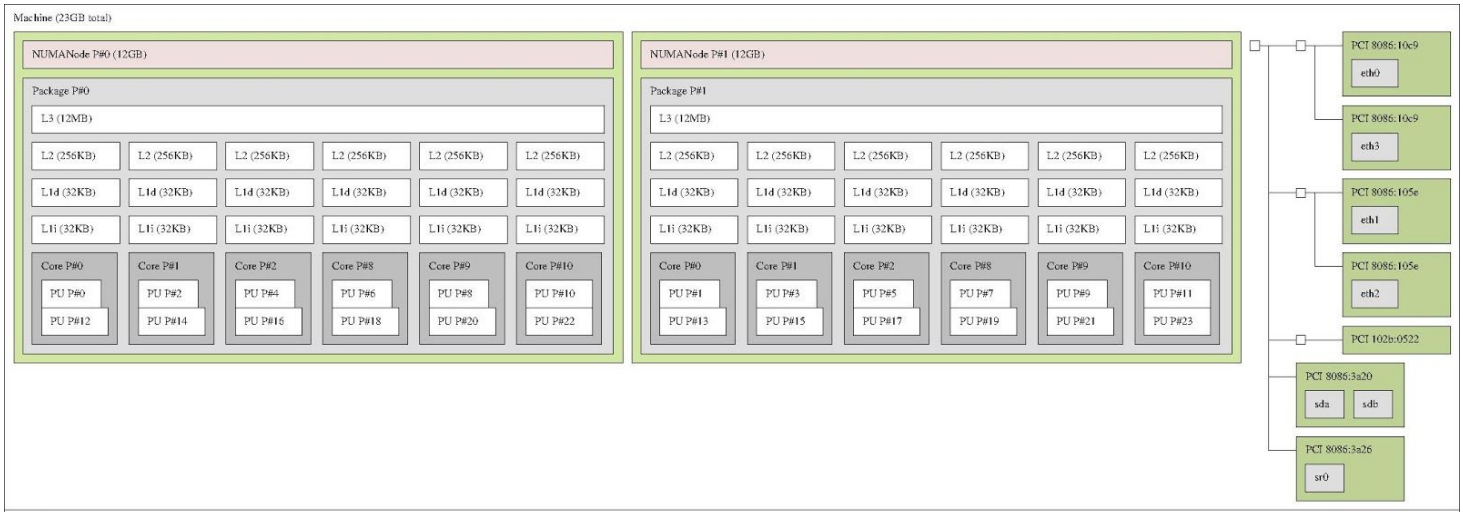
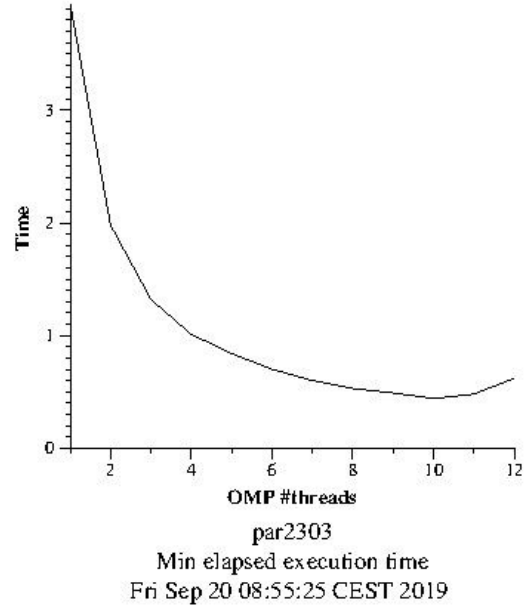


Figure 1. We also include the architectural diagram for the node boada-1, which we obtained when we used the lstopo command (map.fig).

Strong vs. weak scalability



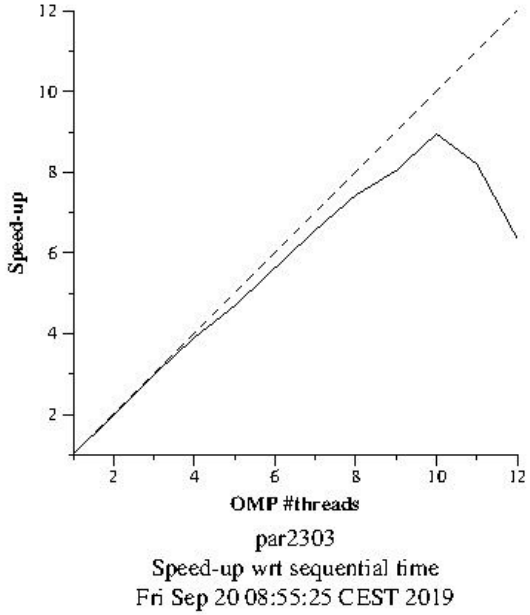


Figure 2
Strong scalability

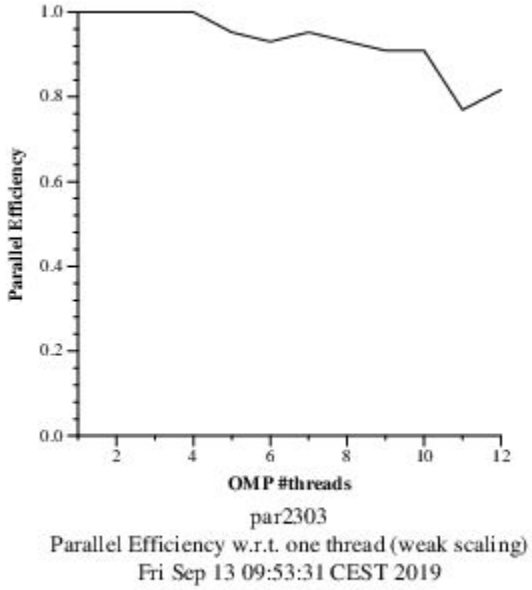


Figure 3
Weak Scalability

Executed with ./submit-strong-omp.sh pi_omp and ./submit-weak-omp.sh pi_omp in boada1-4.

In the plot of time and threads, we can see that the execution time decays logarithmically as we increase the number of threads.
Meanwhile, in the speed-up plot, we can see that the speed-up increases as we increase the number of threads, and reaches its limit at 10~11 threads, since then, the speed-up starts to decrease.

For the plot of weak scalability (figure 3), we can see that the Parallel Efficiency is 1 from 1 to 4 threads, then starts decreasing and his minimum parallel efficiency its at 10~11, near to 0.7, where we can hit the maximum speed-up.

Analysis of task decompositions for 3DFFT

We used the “Tareador” and “Paraver” tools for the analysis of task decompositions using different versions of the program 3DFFT. The results are the following ones:

Version	T_1	T_∞	Parallelism
Seq	639780000 us	639760000 us	1,000031262
v1	639780000 us	639760000 us	1,000031262
v2	639780000 us	361525000 us	1,769670147
v3	639780000 us	155065000 us	4,125882694
v4	639780000 us	64750000 us	9,880772201
v5	639780000 us	35361000 us	18.09281411

These results have been calculated using the total number of instructions (we can find it at the top of every “Tareador” execution) and the total time for all the processors (in the view simulation). For instance, in the original version, we can calculate with these values the time to execute an instruction 639780000/639780 = 1000.

Therefore, T_1 = inst * 1000 = 639780000 us (the same in all versions) and T_∞ = 639760000 us (suming the instructions of the critical path). Finally, the parallelism is T_1 / T_∞ = 1,000031262.

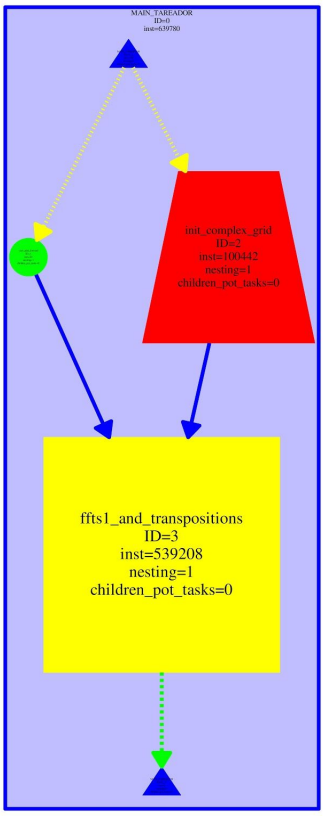


Figure 4, original file.

Version 1

We had to change the main of the 3dfft program in order to get a sequence of finer grained tasks, as you can see in figure 5. In this version, the dependency graph increase the number of tasks and their dependencies. (Figure 6)

```
tareador_start_task("v1_1");
ffts1_planes(p1d, in_fftw);
tareador_end_task("v1_1");
    tareador_start_task("v1_2");
transpose_xy_planes(tmp_fftw, in_fftw);
tareador_end_task("v1_2");
tareador_start_task("v1_3");
ffts1_planes(p1d, tmp_fftw);
    tareador_end_task("v1_3");
    tareador_start_task("v1_4");
transpose_zx_planes(in_fftw, tmp_fftw);
    tareador_end_task("v1_4");
    tareador_start_task("v1_5");
ffts1_planes(p1d, in_fftw);
tareador_end_task("v1_5");
    tareador_start_task("v1_6");
transpose_zx_planes(tmp_fftw, in_fftw);
    tareador_end_task("v1_6");
    tareador_start_task("v1_7");
transpose_xy_planes(in_fftw, tmp_fftw);
tareador_end_task("v1_7");
```

Figure 5, version 1 code.

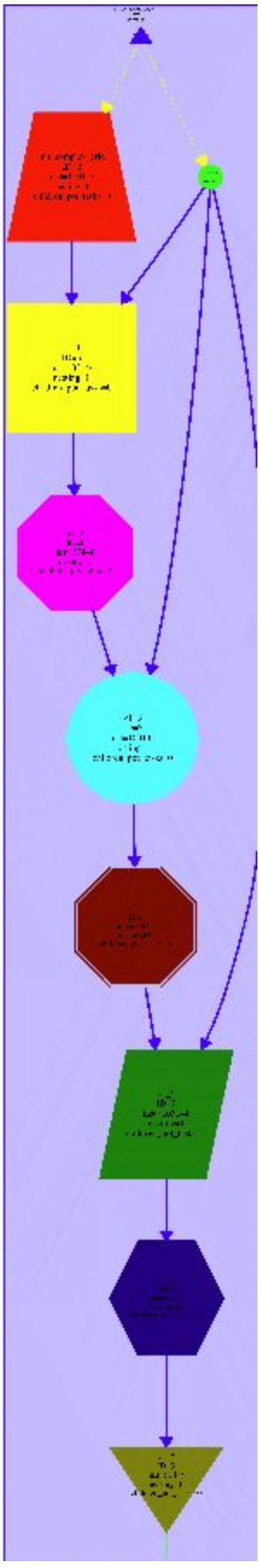


Figure 6, graph dependency version 1.

Version 2

Following the instructions of the document and starting from version 1, we added tareador_start_task() and tareador_end_task in function ffts1_planes. Each iteration of the loop creates a new task and hence the dependency graph increases his size a little more in this version.

```
void ffts1_planes(fftwf_plan p1d, fftwf_complex in_fftw[][N][N]) {
    int k,j;
        for (k=0; k<N; k++)  {
                tareador_start_task("v2_1");
                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("v2_1");
        }
}
```

Figure 7, version 2 code.

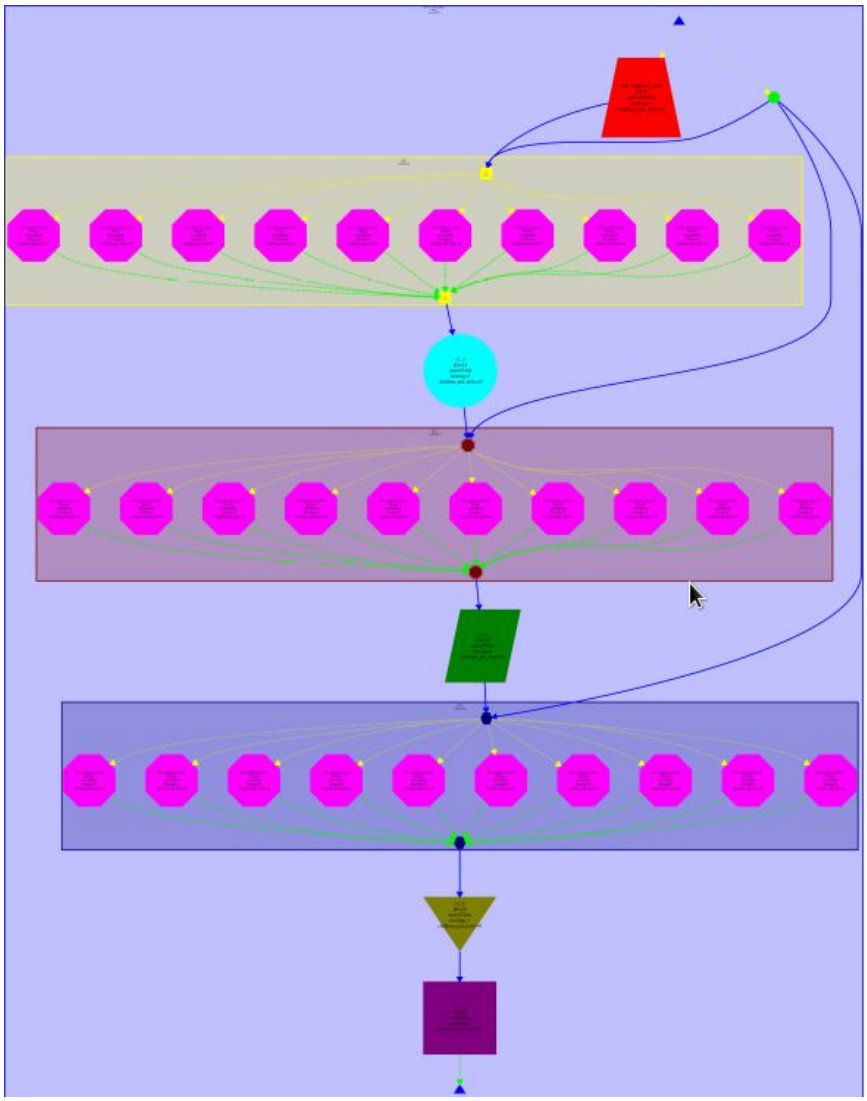


Figure 8, graph dependency of version 2

Version 3

In this version, the same happens as the version 2. We include more tareador_start_task() and tareador_end_task and that's why the dependency graph continues increasing his size.

```
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++) {
                tareador_start_task("v3_1");
     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("v3_1");
    }
}
```

Figure 9, code of version 3

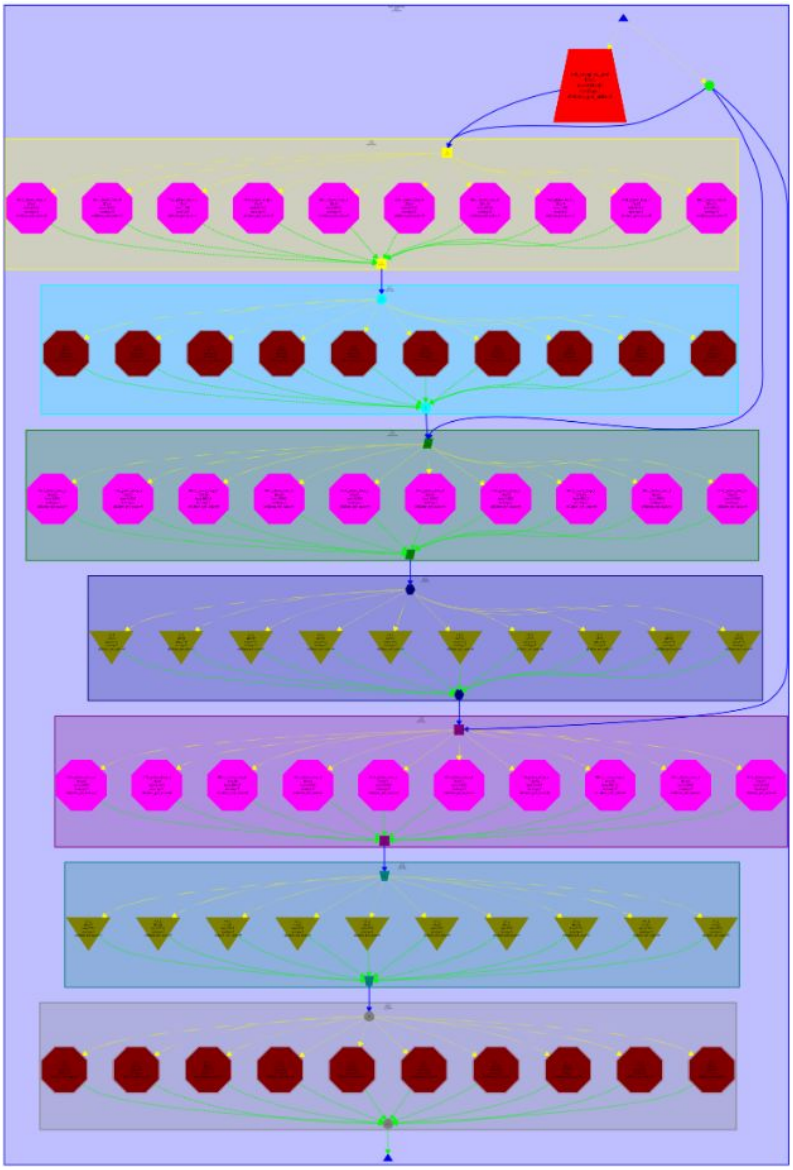


Figure 10, graph dependency of version 3

Version 4

More of the same. More tasks, more dependencies and more size of the graph.

```
void init_complex_grid(fftwf_complex in_fftw[][N][N]) {
  int k,j,i;

  for (k = 0; k < N; k++) {
        tareador_start_task("v4_1");
    for (j = 0; j < N; j++) {
      for (i = 0; i < N; i++)
      {
        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)));
        in_fftw[k][j][i][1] = 0;
#if TEST
        out_fftw[k][j][i][0]= in_fftw[k][j][i][0];
        out_fftw[k][j][i][1]= in_fftw[k][j][i][1];
#endif
      }
    }
    tareador_end_task("v4_1");
  }
}
```

Figure 11, code of version 4

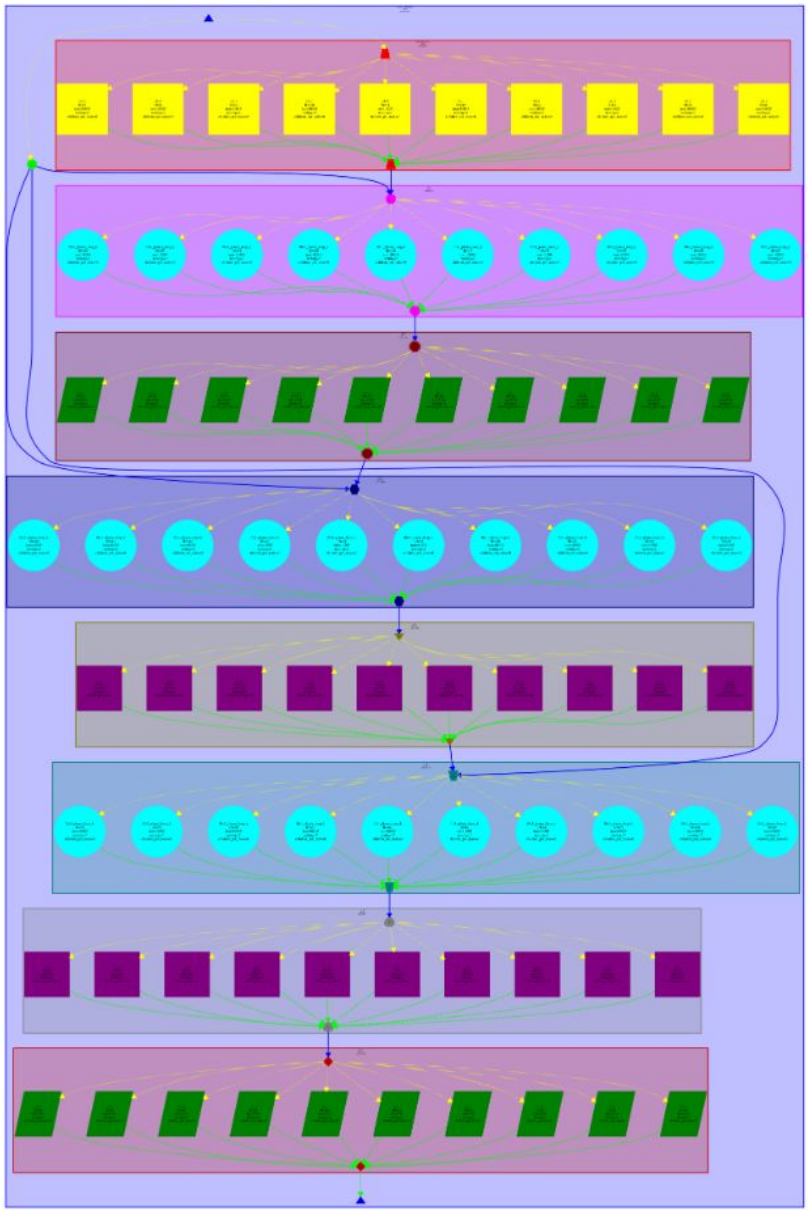


Figure 12, graph dependency of version 4

Version 5

In this last version, we need to include more “tareador_start()/end_task()” between loops in order to get much finer-grained tasks in the different functions of the code. For example:

```
void transpose_zx_planes(fftwf_complex in_fftw[][N][N], fftwf_complex tmp_fftw[][N][N]) {
    int k, j, i;

    for (k=0; k<N; k++) {
     tareador_start_task("v3_2");
     for (j=0; j<N; j++) {
        tareador_start_task("v5_3");
       for (i=0; i<N; i++)
       {
         in_fftw[i][j][k][0] = tmp_fftw[k][j][i][0];
         in_fftw[i][j][k][1] = tmp_fftw[k][j][i][1];
       }
       tareador_end_task("v5_3");
     }
     tareador_end_task("v3_2");
    }
}
```

Figure 13, code of version 5

In this case, the dependency graph is enormous, with a lot of tasks and dependencies between them. (We could not do the image bigger)

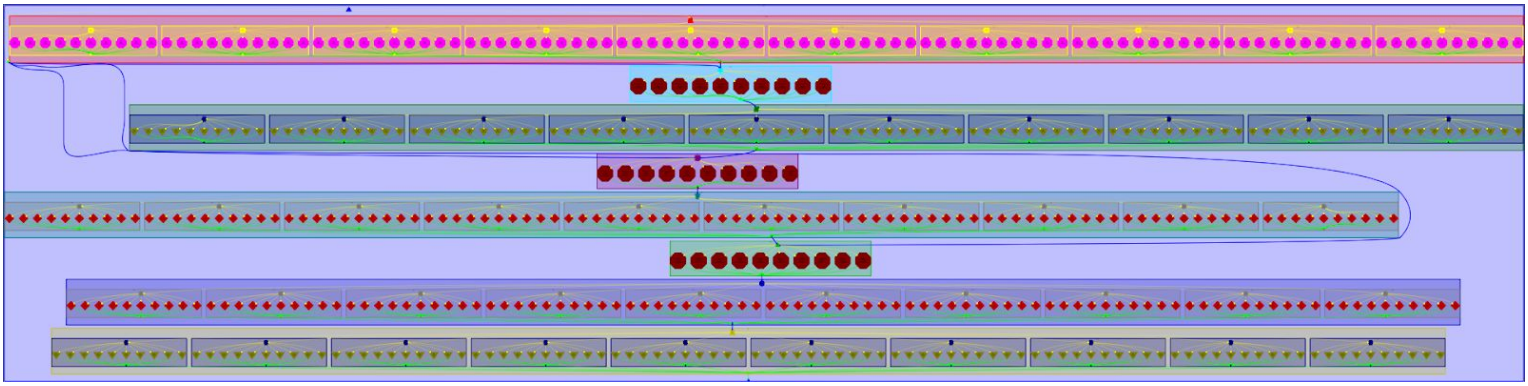


Figure 14, graph dependency of version 5.

These are the timelines of version 4 with 1, 2, 4, 8, 16 and 32 processors:

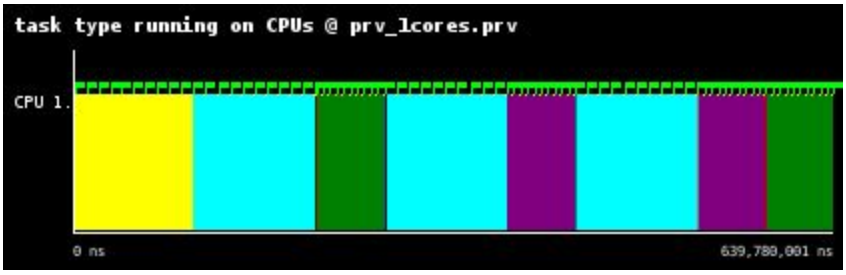


Figure 15, v4 with 1 processor.

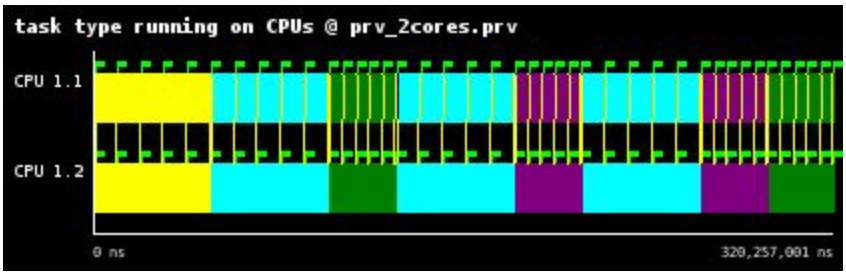


Figure 16, v4 with 2 processors.

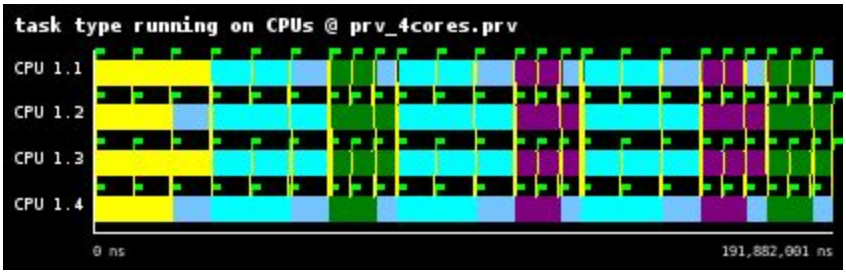


Figure 17, v4 with 4 processors.

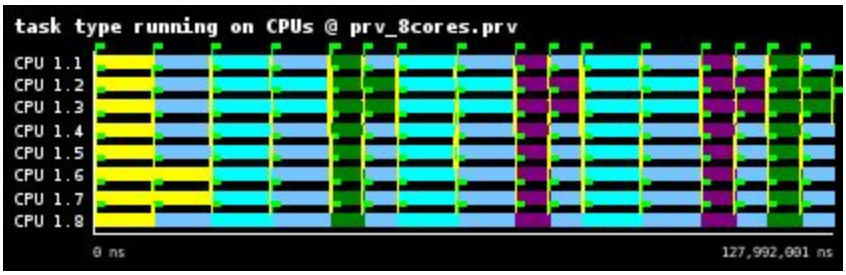


Figure 17, v4 with 8 processors.

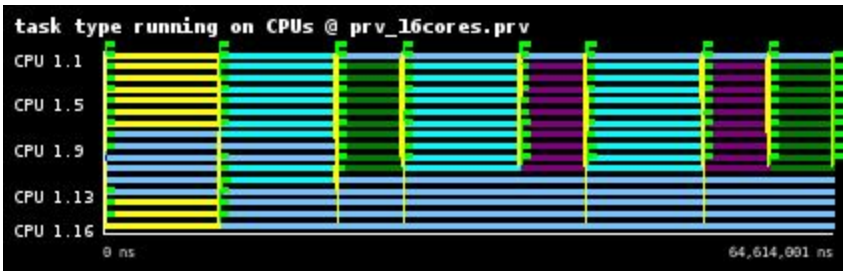


Figure 18, v4 with 16 processors.

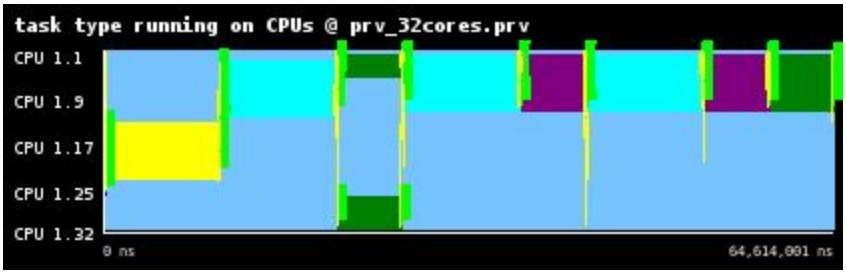


Figure 19, v4 with 32 processors.

And these others are the timelines of version 5 with 1, 2, 4, 8, 16 and 32 processors:

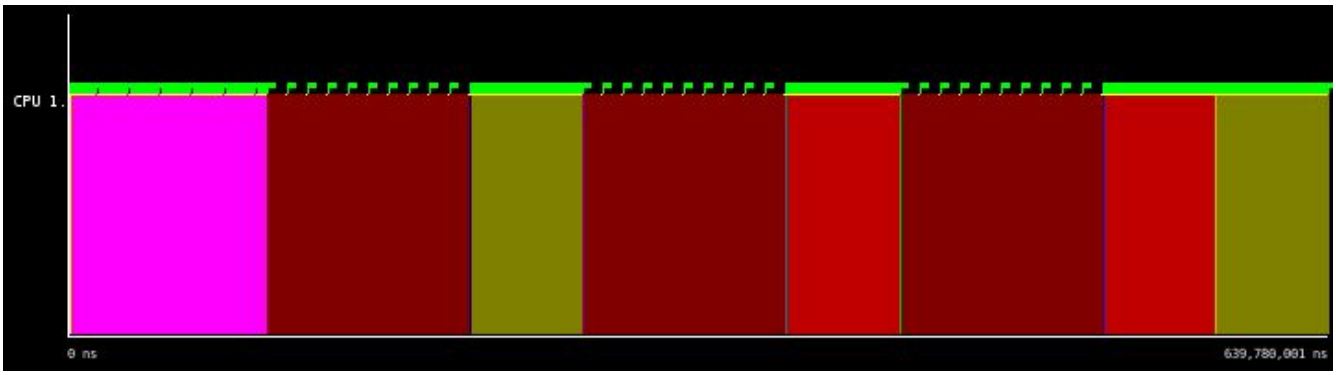


Figure 20, v5 with 1 processor.

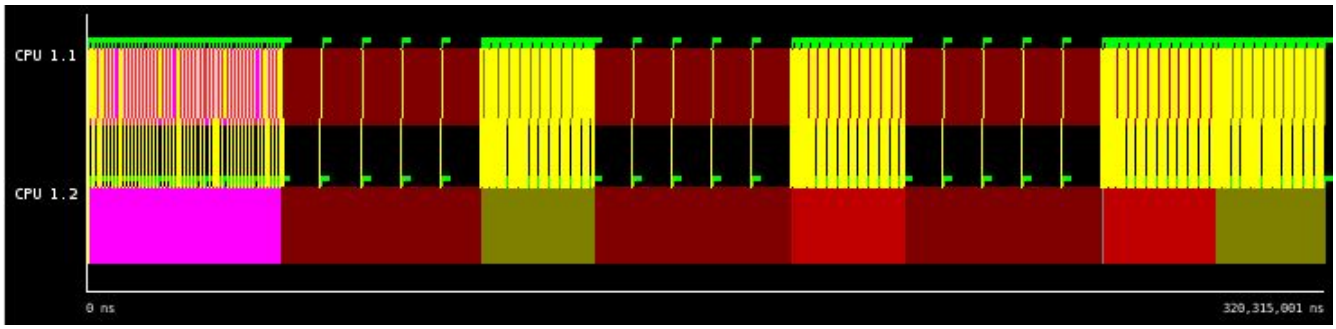


Figure 21, v5 with 2 processors.

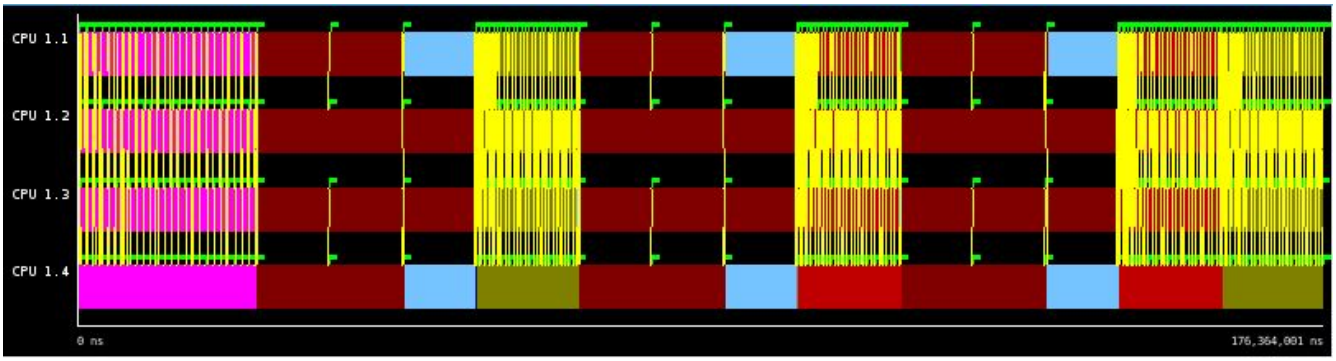


Figure 22, v5 with 4 processors.

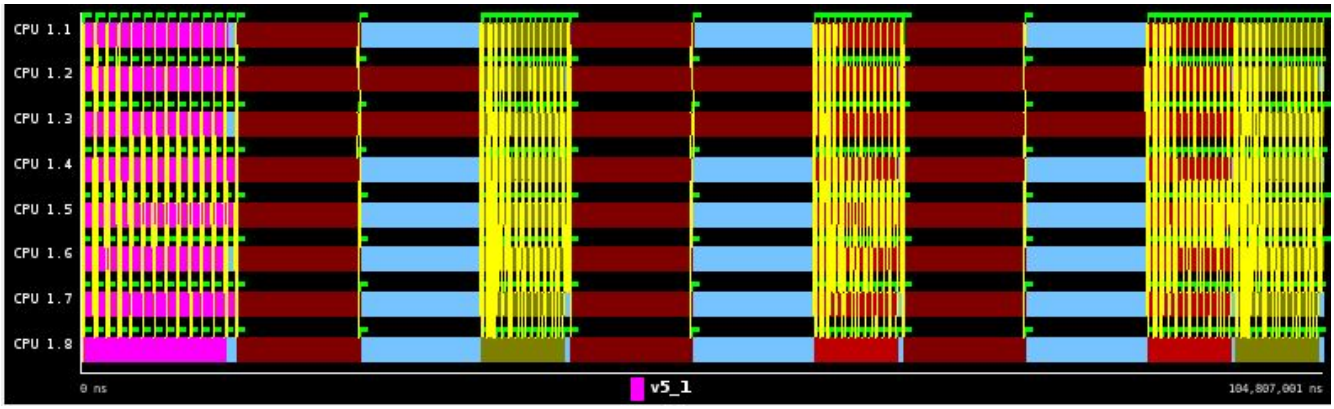


Figure 23, v5 with 8 processors.

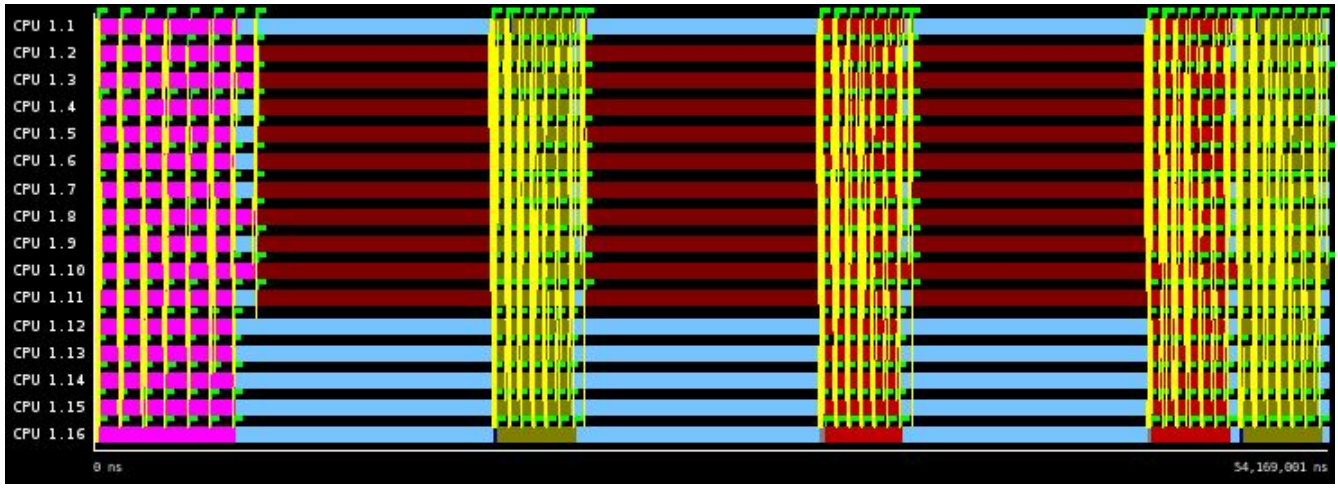


Figure 24, v5 with 16 processors.

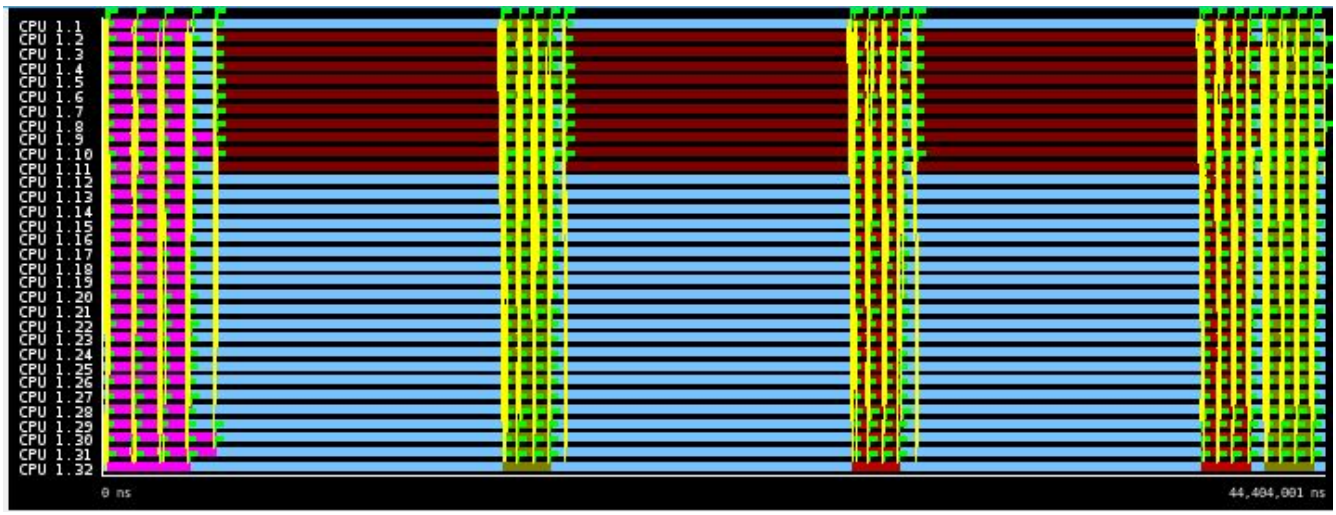


Figure 25, v5 with 32 processors.

And this is the plot comparing the two versions and their timelines:

Threads	Time v4(ns)	Time v5(ns)
1	639780001	639780001
2	320257001	320315001
4	191882001	176364001
8	127992001	104807001
16	64614001	54169001
32	64614001	44404001

Figure 26, table with all the data

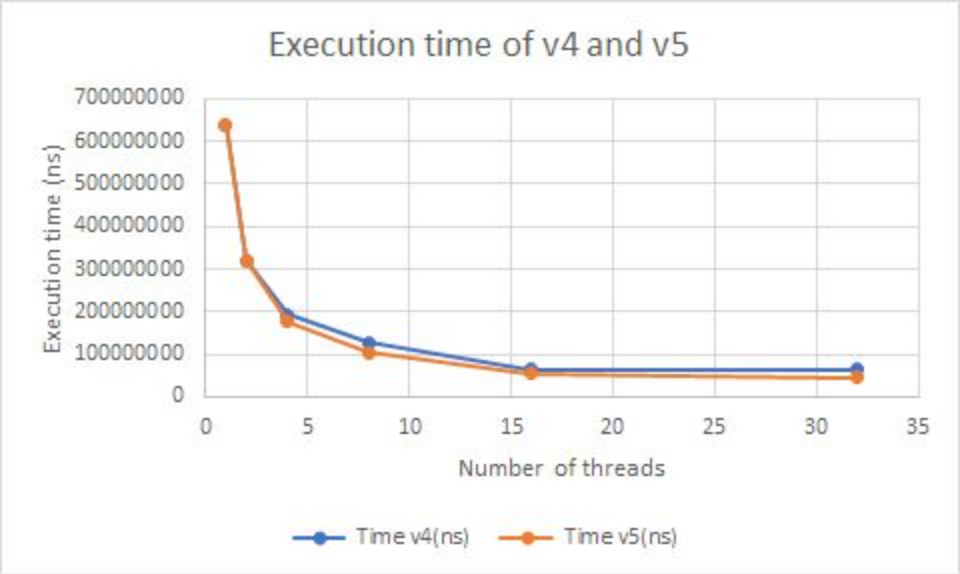


Figure 27, plot comparing v4 and v5

The version 5 can reach less execution time than the version 4 due to the parallelism that we have implemented in code (see figure 13). We can observe that in version 4, the execution time with 16 and 32 processors is exactly the same, this doesn't happens for the version 5, which is an improvement of version 4.

Understanding the parallel execution of 3DFFT

We have calculated the different values, which you asked, in the 3 versions. Here, it is the result:

Version	φ	S∞	T_1	T_8	S_8
initial version in `3dfft_omp.c`	0.657	2.91	2321036862 ns	1841773627 ns	1.26
new version with improved φ	0.82	5.55	2470972960 ns	837878969 ns	2.94
final version with reduced parallelisation overheads	-	-	2414215037 ns	603623966 ns	3.99

Firstly, we can get T_1 and T_8 of the timelines in bottom right corner. Secondly, we can calculate S_8 dividing T_1 and T_8. Finally, φ is calculated dividing Tpar (time which can be parallelized) and T_1. Besides, S∞ is calculated using the formula, which we saw in class, $S_\infty = 1/(1-\varphi)$.

Executing “Paraver” and the script “submit-strong-omp.sh”, we obtained the following timelines, plots and OpenMP state profiles with the different versions:

Initial version:

This version is useful to see the differences between the other versions and to see if there is some improvement or not. We obtained the following screenshots:

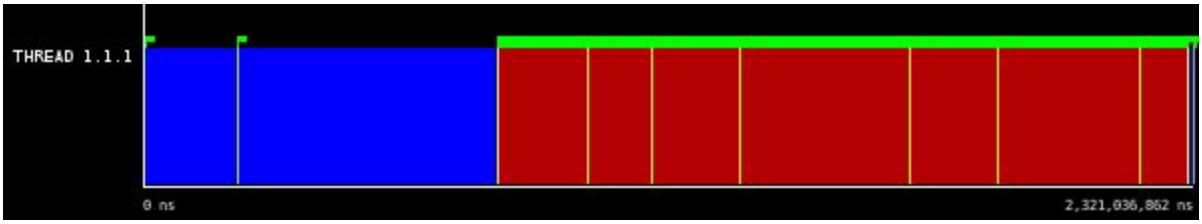


Figure 28, timeline of initial version with 1 thread

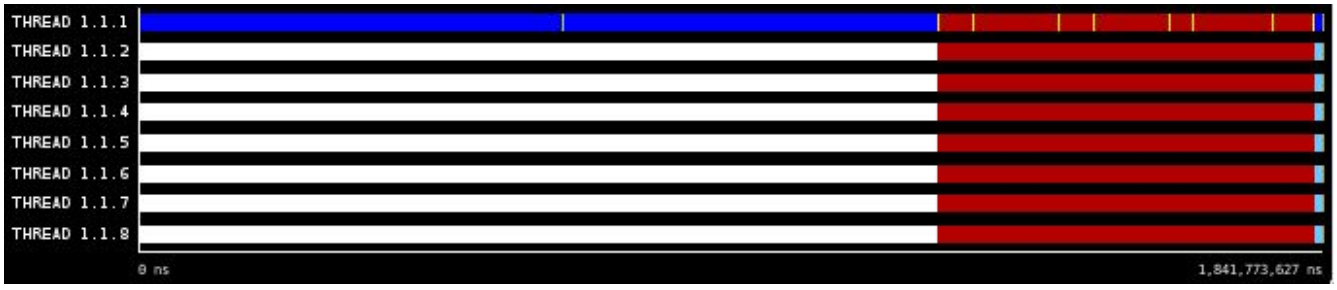


Figure 29, timeline of initial version with 8 threads

2D profile @ 3dfft_omp-8-boada-1.prv (on boada-1)

	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	95.03 %	-	4.90 %	0.05 %	0.02 %	0.00 %
THREAD 1.1.2	28.22 %	67.99 %	3.77 %	-	0.01 %	-
THREAD 1.1.3	25.18 %	67.99 %	6.82 %	-	0.01 %	-
THREAD 1.1.4	27.75 %	67.99 %	4.24 %	-	0.01 %	-
THREAD 1.1.5	28.03 %	67.99 %	3.96 %	-	0.01 %	-
THREAD 1.1.6	30.62 %	67.99 %	1.37 %	-	0.01 %	-
THREAD 1.1.7	29.34 %	67.99 %	2.66 %	-	0.01 %	-
THREAD 1.1.8	29.12 %	67.00 %	2.87 %	-	0.01 %	-
Total	293.30 %	475.95 %	30.59 %	0.05 %	0.11 %	0.00 %
Average	36.66 %	67.99 %	3.82 %	0.05 %	0.01 %	0.00 %
Maximum	95.03 %	67.00 %	6.82 %	0.05 %	0.02 %	0.00 %
Minimum	25.18 %	67.99 %	1.37 %	0.05 %	0.01 %	0.00 %
StDev	22.11 %	0.00 %	1.52 %	0 %	0.00 %	0 %
Avg/Max	0.39	1.00	0.56	1	0.88	1

Figure 30, OpenMP state profile of initial version

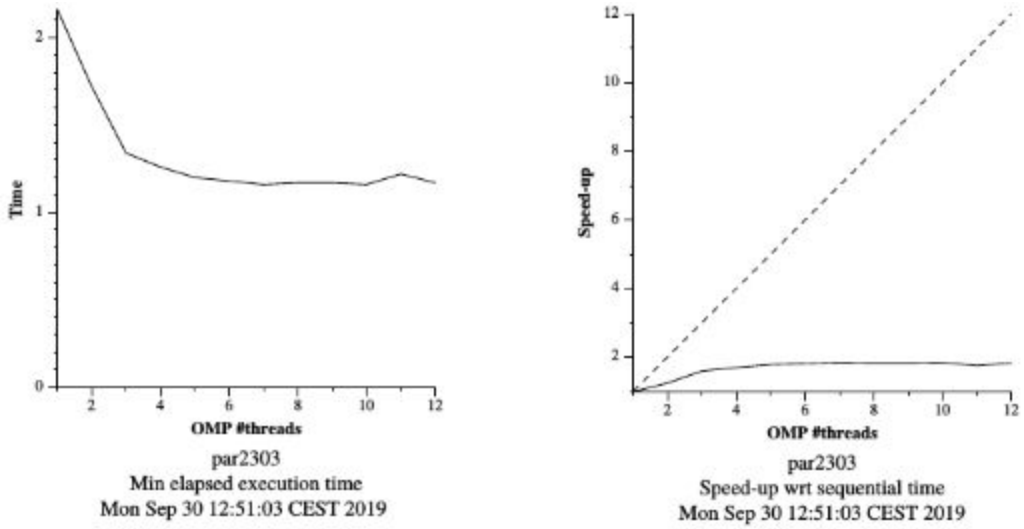


Figure 31, plots of initial version generated with “submit-strong-omp.sh”

Improved φ version:

Uncommenting the pragmas in the code, we get a bigger Tpar (more red space in figure 32 and 33) and hence a bigger φ. We can see that in the following screenshots:

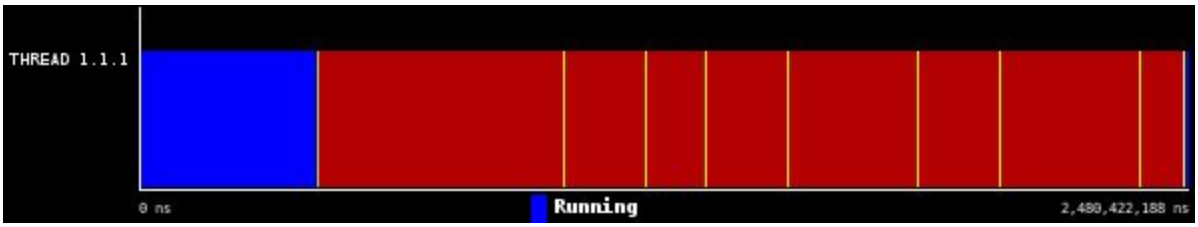


Figure 32, timeline of improved version with 1 thread

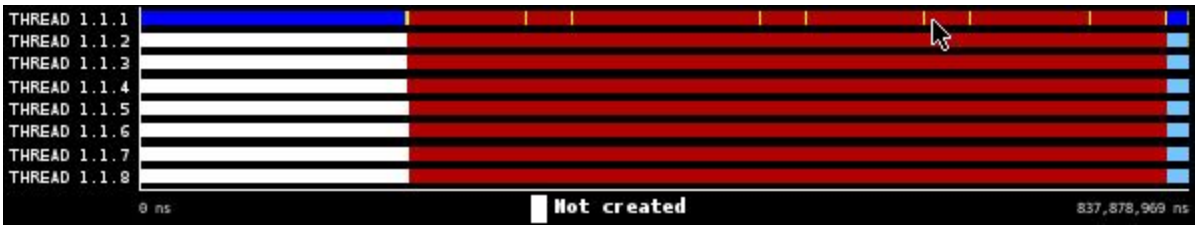


Figure 33, timeline of improved version with 8 threads

2D profile @ 3dfft_omp_improving-8-boada-1.prv (on boada-1)

	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	90.52 %	-	9.29 %	0.15 %	0.03 %	0.00 %
THREAD 1.1.2	72.32 %	25.00 %	1.65 %	-	0.03 %	-
THREAD 1.1.3	63.26 %	26.00 %	10.70 %	-	0.03 %	-
THREAD 1.1.4	62.84 %	26.02 %	11.11 %	-	0.03 %	-
THREAD 1.1.5	71.72 %	26.04 %	2.21 %	-	0.03 %	-
THREAD 1.1.6	71.11 %	26.09 %	2.77 %	-	0.03 %	-
THREAD 1.1.7	71.20 %	26.10 %	2.66 %	-	0.03 %	-
THREAD 1.1.8	70.92 %	26.14 %	2.90 %	-	0.03 %	-
Total	573.90 %	182.39 %	43.30 %	0.15 %	0.26 %	0.00 %
Average	71.74 %	26.06 %	5.41 %	0.15 %	0.03 %	0.00 %
Maximum	90.52 %	26.14 %	11.11 %	0.15 %	0.03 %	0.00 %
Minimum	62.84 %	25.00 %	1.65 %	0.15 %	0.03 %	0.00 %
StDev	7.95 %	0.05 %	3.88 %	0 %	0.00 %	0 %
Avg/Max	0.79	1.00	0.49	1	0.96	1

Figure 34, OpenMP state profile of improved version

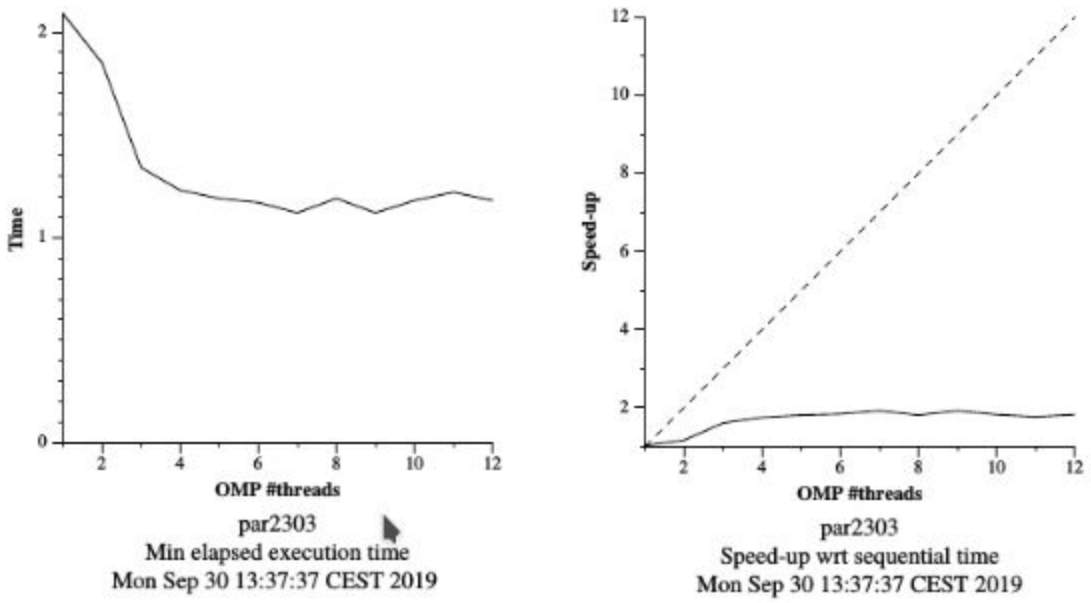


Figure 35, plots of improved version generated with “submit-strong-omp.sh”

Final version:

Simply moving the "#pragma omp for"s one line up in the four functions, we reduce parallelisation overheads, synchronization (red color almost disappear) and time execution. We can see that in the following screenshots:

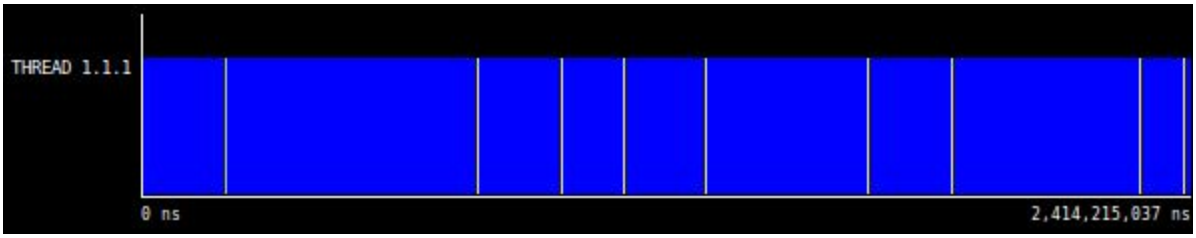


Figure 36, timeline of final version with 1 thread

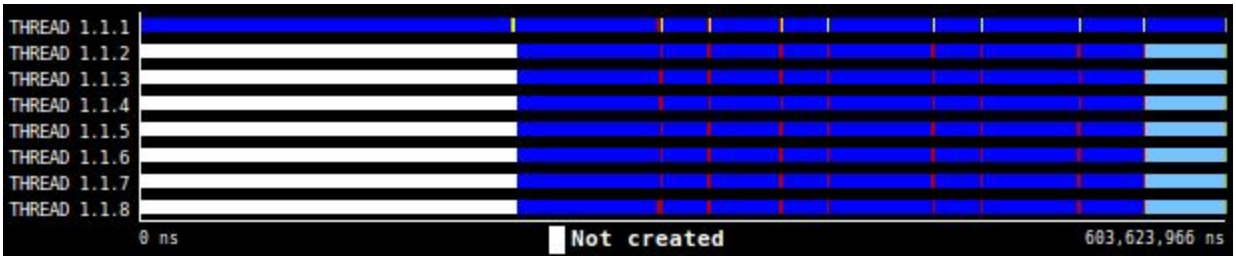


Figure 37, timeline of final version with 8 threads

2D profile @ 3dfft_omp_overheads-8-boada-1.prv (on boada-1)

	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	99.26 %	-	0.55 %	0.19 %	0.00 %	0.00 %
THREAD 1.1.2	61.90 %	37.32 %	0.77 %	-	0.00 %	-
THREAD 1.1.3	62.05 %	37.43 %	0.51 %	-	0.00 %	-
THREAD 1.1.4	62.43 %	37.35 %	0.22 %	-	0.00 %	-
THREAD 1.1.5	62.01 %	37.32 %	0.67 %	-	0.00 %	-
THREAD 1.1.6	61.96 %	37.34 %	0.70 %	-	0.00 %	-
THREAD 1.1.7	61.92 %	37.34 %	0.74 %	-	0.00 %	-
THREAD 1.1.8	61.81 %	37.38 %	0.82 %	-	0.00 %	-
Total	533.34 %	261.48 %	4.98 %	0.19 %	0.01 %	0.00 %
Average	66.67 %	37.35 %	0.62 %	0.19 %	0.00 %	0.00 %
Maximum	99.26 %	37.43 %	0.82 %	0.19 %	0.00 %	0.00 %
Minimum	61.81 %	37.32 %	0.22 %	0.19 %	0.00 %	0.00 %
StDev	12.32 %	0.04 %	0.18 %	0 %	0.00 %	0 %
Avg/Max	0.67	1.00	0.76	1	0.44	1

Figure 38, OpenMP state profile of final version

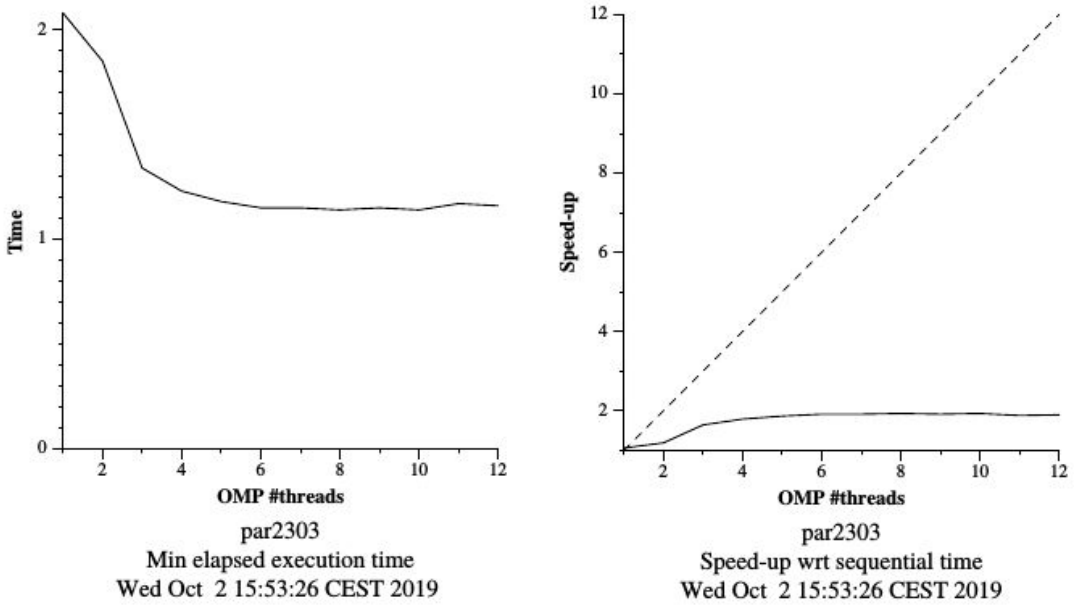


Figure 39, plots of final version generated with “submit-strong-omp.sh”

And about the plots, we can observe that in the three versions (initial version (figure 31), new version with improved φ (figure 35), and final version with reduced parallelisation overheads (figure 39)), the execution time decreases until the 2~4 threads. Then, the execution time it is practically the same for the rest of number of threads. In the Speed-up plot, we obtain something similar: the speed-up increases until the 2~4 threads and then, the speed-up its practically the same for the rest of number of threads.