PAR Laboratory Assignment Lab 1: Experimental setup and tools

PAR2102

Alejandro Alarcón Torres
Guillem Reig Gaset

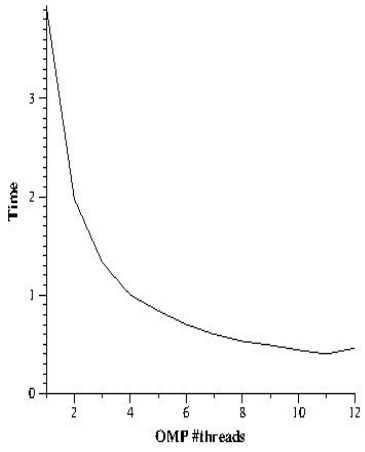
1- Node architecture and memory

	boada-1 to boada-4	boada-5	boada-6 to boada-8
Number of sockets per node	2	2	2
Number of cores per socket	6	6	8
Number of threads per core	2	2	1
Maximum core frequency	2395	2600	1700
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)	12GB	31GB	16GB
Main memory size (per node)	23GB	63GB	31GB

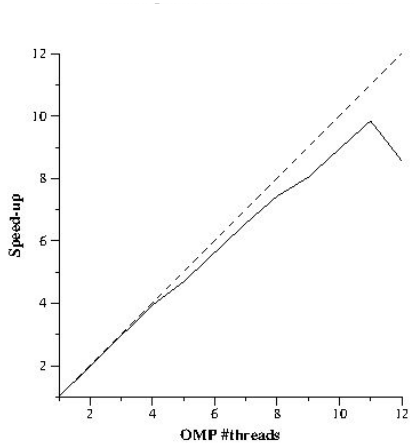
This is the diagram that we obtain when we use the command lstopo with the "--of fig map.fig" flag in the boada-1:

2- Strong vs weak scalability

Strong scalability refers to the increment in the number of threads without changing the size of the problem to solve it faster.



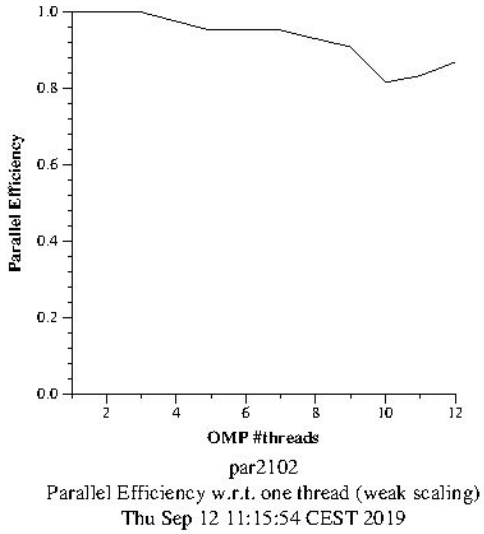
par2102
Min elapsed execution time
Thu Sep 12 11:13:39 CEST 2019



par2102
Speed-up wrt sequential time
Thu Sep 12 11:13:39 CEST 2019

As we can see, as the amount of threads increases, the time decreases, generally. However, given too many threads the time increases right back as there is too much work due to the overheads.

Weak scalability refers to the proportional increment in the number of threads as well as the size of the problem. Therefore, within the same amount of time we can solve larger and more complex problems.



In an ideal both of the techniques can get an infinite speed-up due to the parallelization of any problem using infinite processors, but as we said before, in the real world there are overheads that would drop down this speed-up.

3- Analysis of task decomposition with Tareador

Version	T1	T(infinite)	Parallelism
seq	639780	639760	1.000031
v1	639780	639760	1.000031
v2	639780	361190	1.771
v3	639780	154354	4.145
v4	639780	64018	9.994
v5	639780	-	-

3.1- Original version

```
/* Initialize Tareador analysis */
tareador_ON ();
START_COUNT_TIME;

tareador_start_task("start_plan_forward");
start_plan_forward(in_fftw, &pld);
tareador_end_task("start_plan_forward");

STOP_COUNT_TIME("3D FFT Plan Generation");

START_COUNT_TIME;

tareador_start_task("init_complex_grid");
init_complex_grid(in_fftw);
tareador_end_task("init_complex_grid");

STOP_COUNT_TIME("Init Complex Grid FFT3D");

START_COUNT_TIME;

tareador_start_task("fftsi_and_transpositions");
fftsi_planes(pld, in_fftw);

transpose_xy_planes(tmp_fftw, in_fftw);

fftsi_planes(pld, tmp_fftw);

transpose_zx_planes(in_fftw, tmp_fftw);

fftsi_planes(pld, in_fftw);

transpose_zx_planes(tmp_fftw, in_fftw);

transpose_xy_planes(in_fftw, tmp_fftw);
tareador_end_task("fftsi_and_transpositions");

STOP_COUNT_TIME("Execution FFT3D");

/* Finalize Tareador analysis */
tareador_OFF ();
```

Figure 3.1.1: Original code

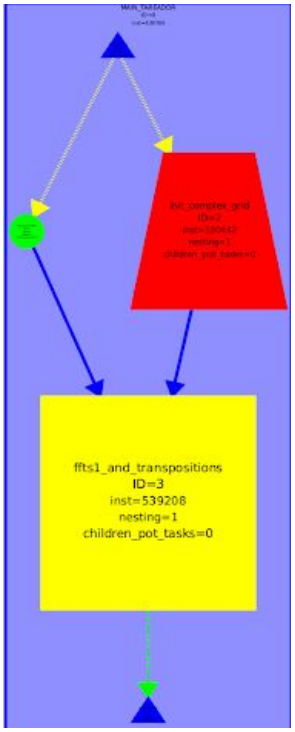


Figure 3.1.2: Diagram for original code

Figure 3.1.2: Simulation for 1 core

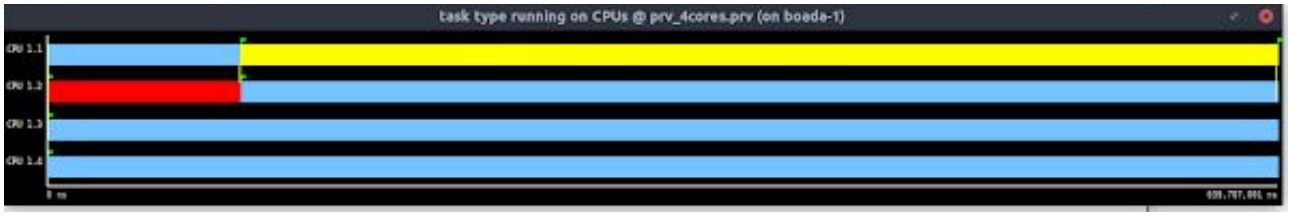


Figure 3.1.3: Simulation for 4 cores

The seq version has problems because a lot of time is used doing the init_complex_grid and the ffts1_and_transpositions tasks, as we can see in the figure 3.1.2. In the figure 3.1.1 we can see that we only use one task for each function, so if we increase the granularity the functions will be executed faster.

3.2- Version 1

```
//tareador_start_task("ffts1_and_transpositions");

    tareador_start_task("ffts1");
    ffts1_planes(p1d, in_fftw);
    tareador_end_task("ffts1");

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

tareador_start_task("ffts1");
ffts1_planes(p1d, tmp_fftw);
    tareador_end_task("ffts1");

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

    tareador_start_task("ffts1");
ffts1_planes(p1d, in_fftw);
    tareador_end_task("ffts1");

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

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

//tareador_end_task("ffts1_and_transpositions");

STOP_COUNT_TIME("Execution FFT3D");

/* Finalize Tareador analysis */
```

Figure 3.2.1: Code for v1

Figure 3.2.2: Diagram for v1l

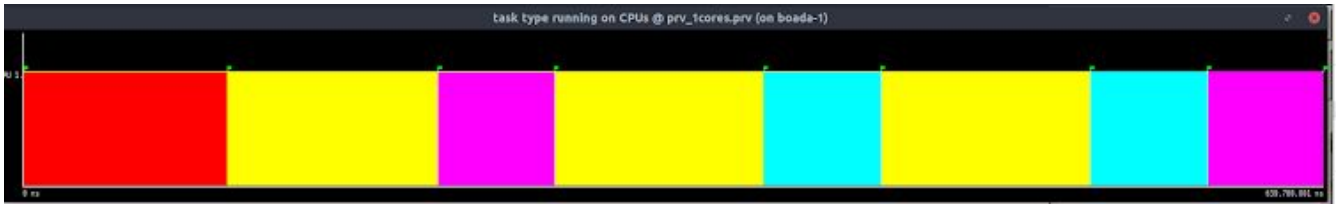


Figure 3.2.3: Simulation for 1 core

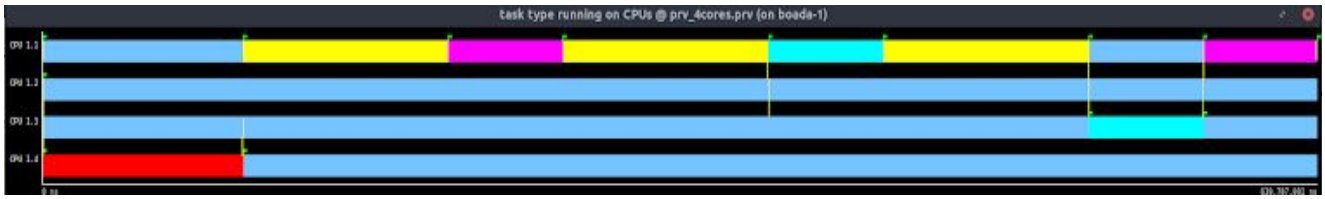


Figure 3.2.4: Simulation for 4 cores

In this version we don't see any difference because the critical path is the same as the seq version. Even though we've increased the granularity, all the tasks maintain their dependencies (as we can see in the figure 3.2.2), so we have the same problem as before.

3.3- Version 2

```
void fftsi_planes(fftwf_plan pid, fftwf_complex in_fftw[][N][N]) {
    int k,j;
    for (k=0; k<N; k++) {
    tareador_start_task("fftsi_loop_k");
     for (j=0; j<N; j++) {
       fftwf_execute_dft( pid, (fftwf_complex *)in_fftw[k][j][0], (fftwf_complex *)in_fftw[k][j][0]);
     }
    tareador_end_task("fftsi_loop_k");
    }
}
```

Figure 3.3.1: Code for v2

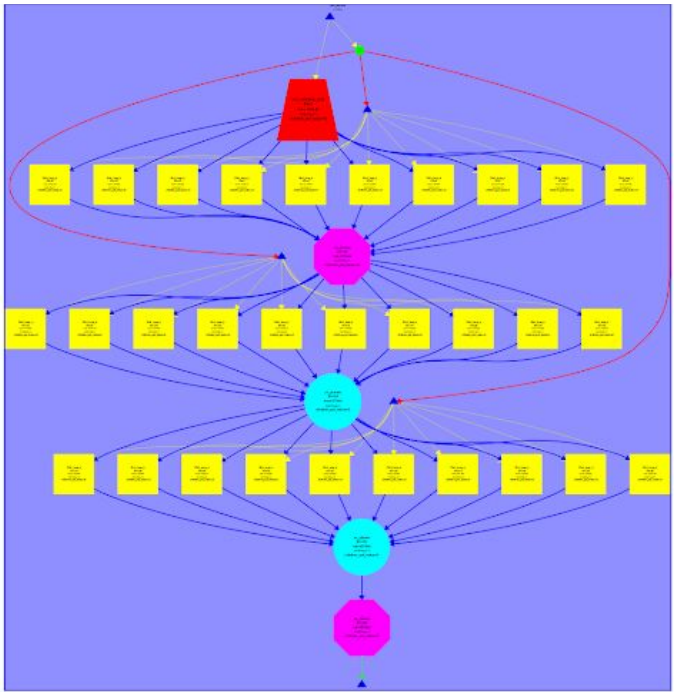


Figure 3.3.2: Diagram for v2

Figure 3.3.3: Simulation for 1 core

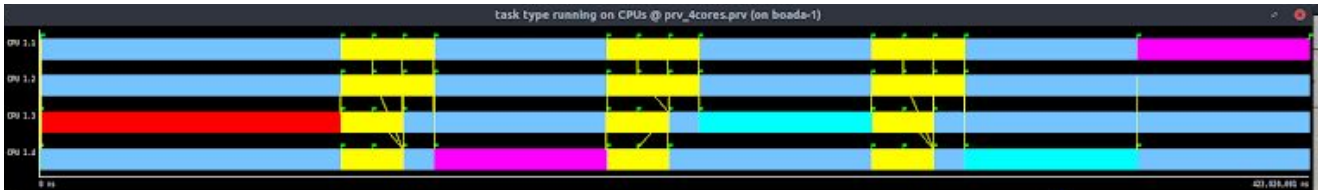


Figure 3.3.4: Simulation for 4 cores

In this version we can see an improvement of the Parallelism, because the loop in the ffts1_planes function can be executed in parallel (figure 3.3.2), solving part of the problem that we had in the previous versions. The transposition tasks have the same dependencies as before, so they still need to wait for the ffts1_planes tasks to be completed.

3.4- Version 3

```
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("xy_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("xy_loop_k");
    }
}

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("zx_loop_k");
     for (j=0; j<N; j++) {
       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("zx_loop_k");
    }
}
```

Figure 3.4.1: Code for v3

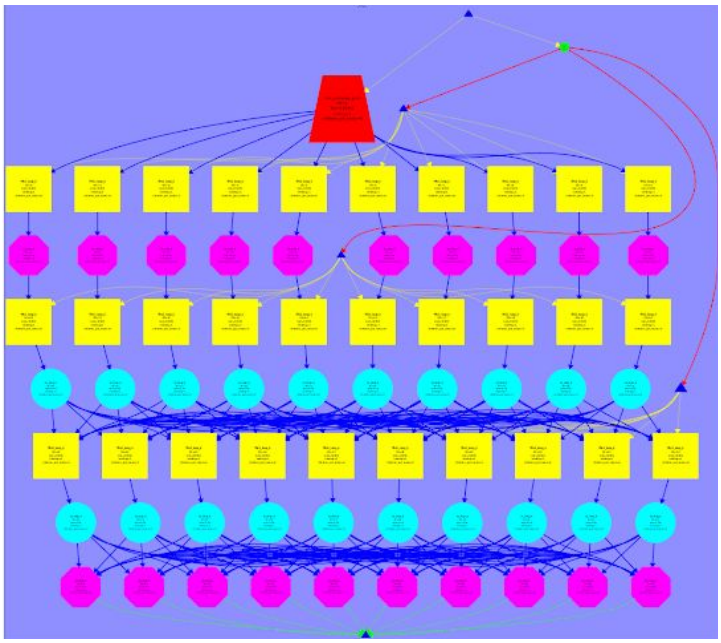


Figure 3.4.2: Diagram for v3

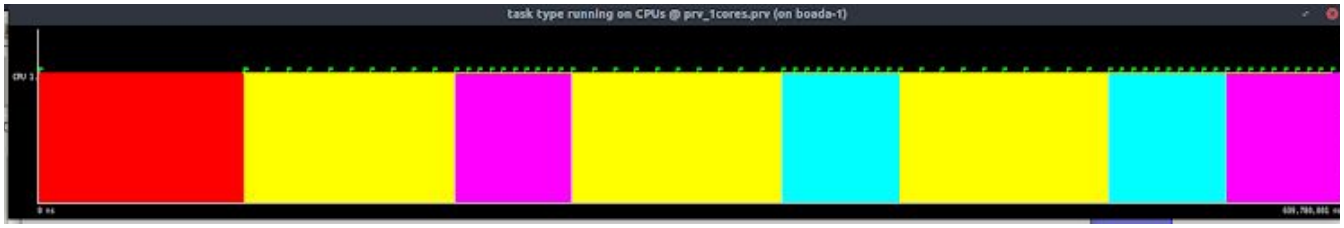


Figure 3.4.3: Simulation for 1 core

Figure 3.4.4: Simulation for 4 cores

In this version we can see that the Parallelism has improved greatly, this is because the loops of the transposition functions can be executed in parallel (figure 3.4.2). The only task that draws back the start of the program is the init_complex_grid as we can see in the figure 3.4.4, because all the other tasks depend on it to start their execution.

3.5- Version 4

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

  for (k = 0; k < N; k++) {
        tareador_start_task("init_complex_grid_loop_k");
    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("init_complex_grid_loop_k");
  }
}
```

Figure 3.5.1: Code for v4

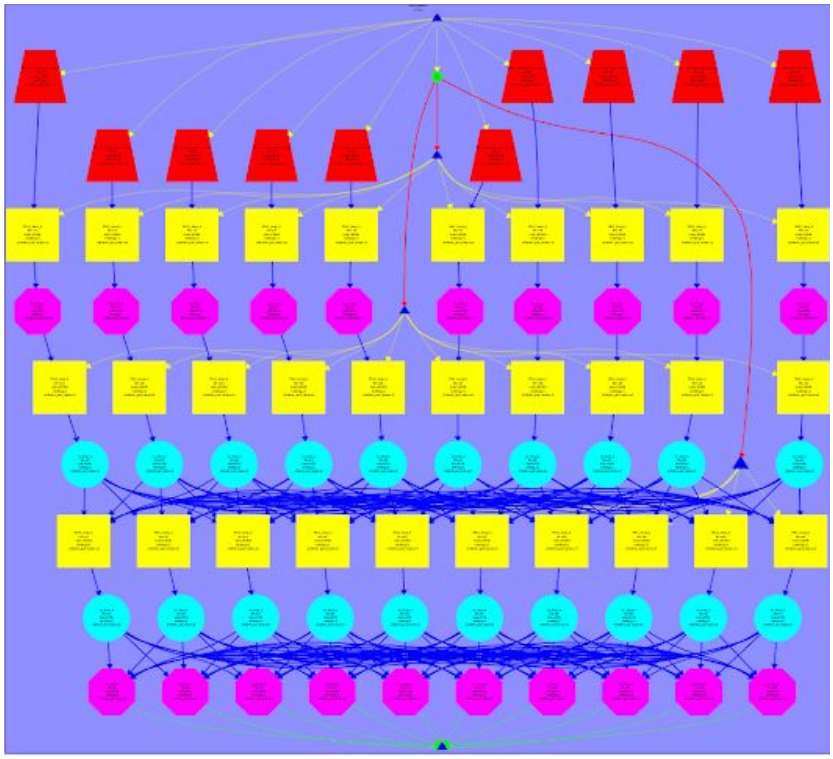


Figure 3.5.2: Diagram for v4

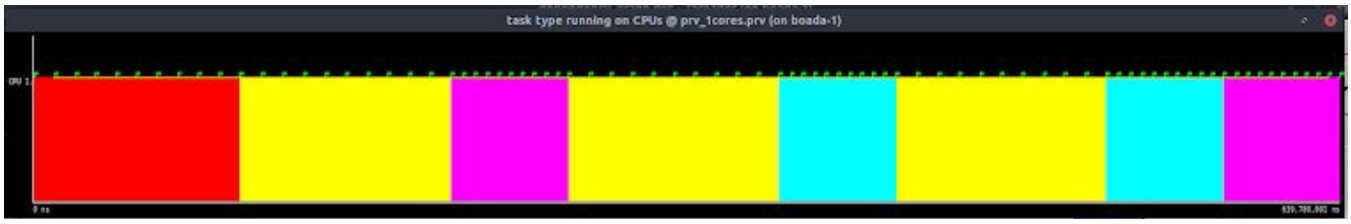


Figure 3.5.3: Simulation for 1 core

Figure 3.5.4:Simulation for 2 cores

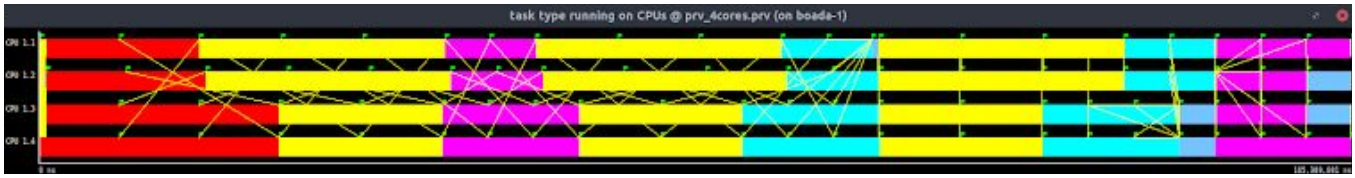


Figure 3.5.5: Simulation for 4 cores

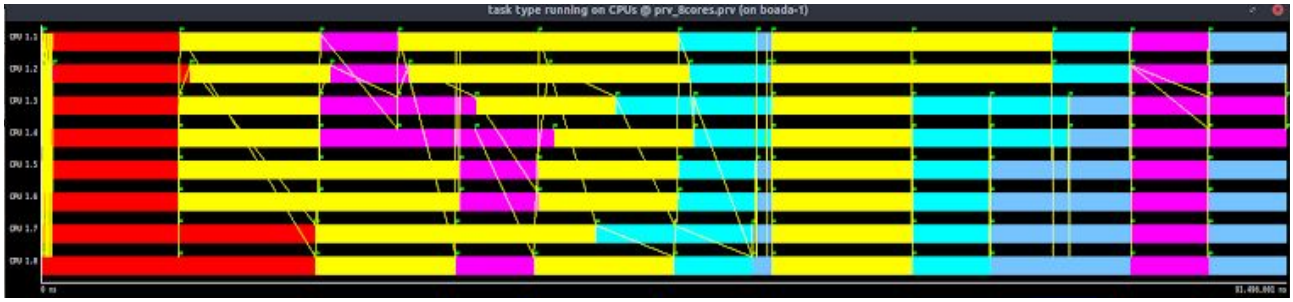


Figure 3.5.6: Simulation for 8 cores

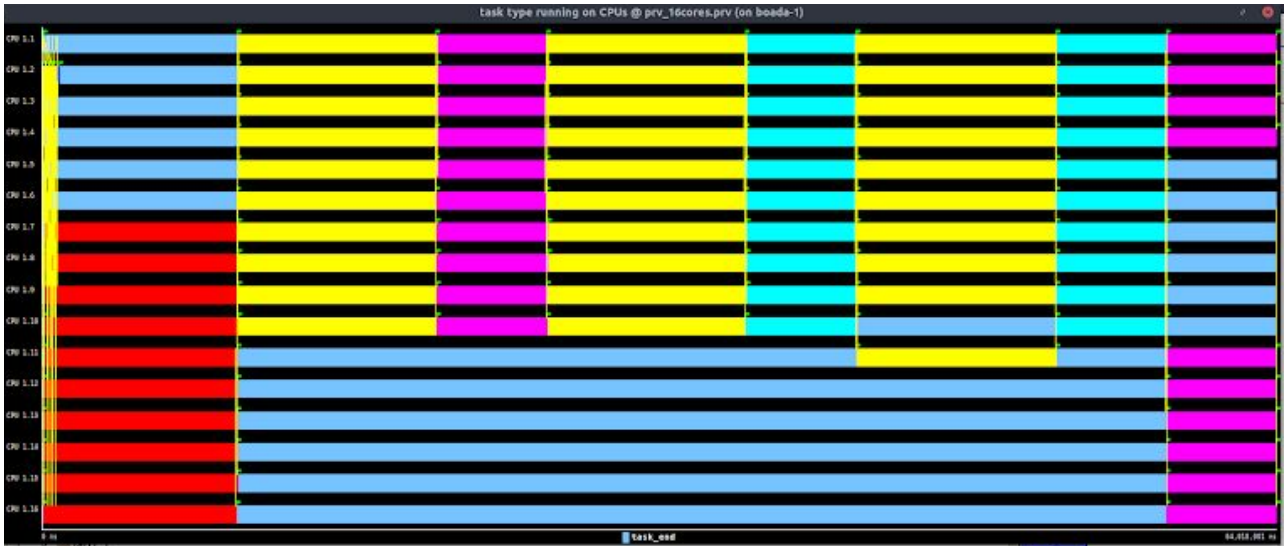


Figure 3.5.7: Simulation for 16 cores

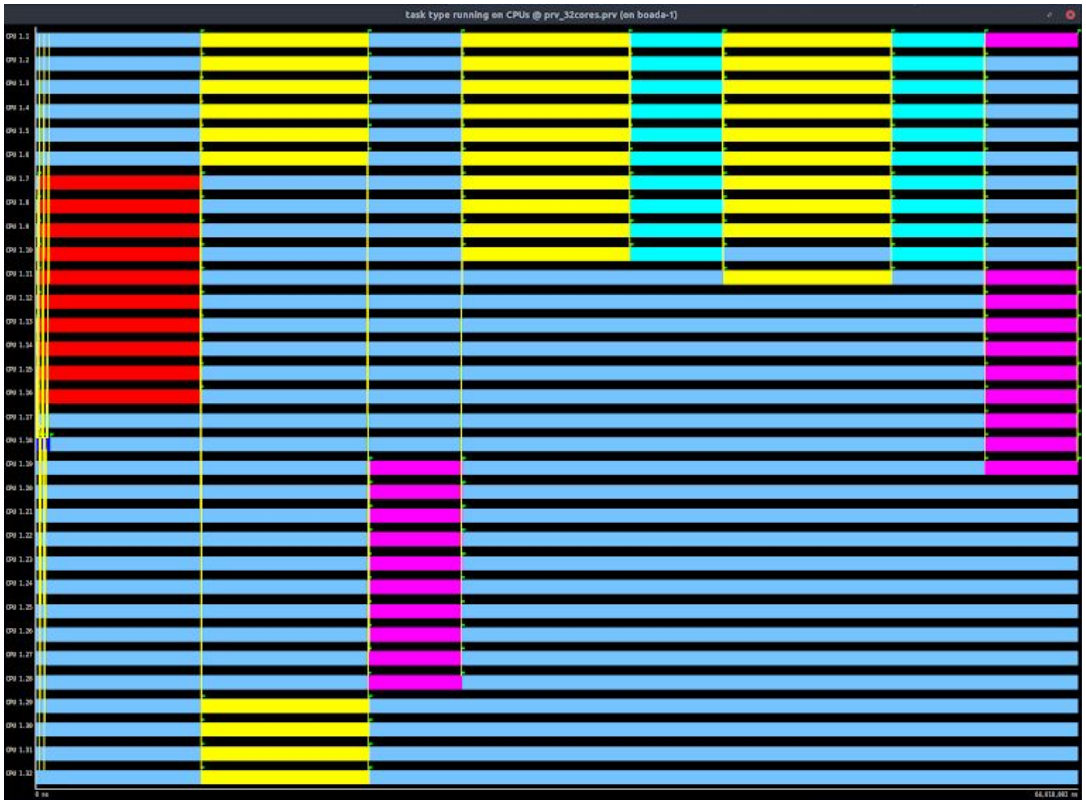
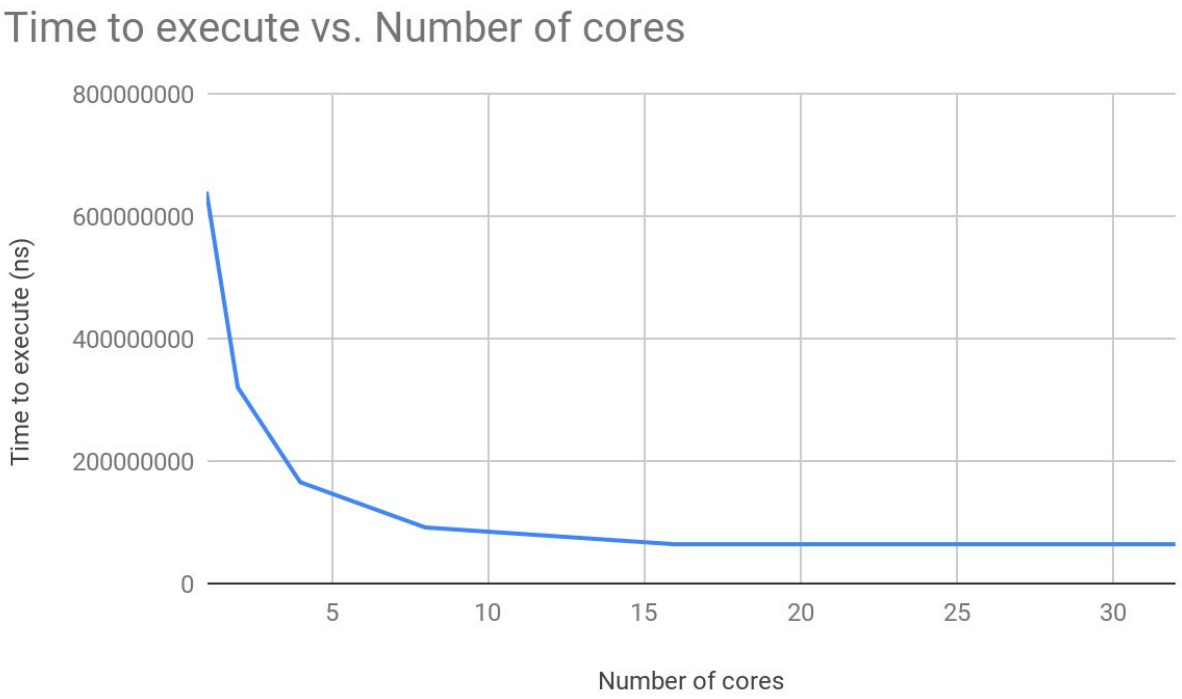


Figure 3.5.8: Simulation for 32 cores

In this version we can see that the Parallelism has improved greatly another time, this is because the init_complex_grid function has been parallelized in the loop k, solving the only problem that we had. As we can see in the table below, the time improves in all cases except when the number of cores goes from 16 to 32. This happens because the minimum number of cores to reach T infinity is 11 (10 for the loop tasks and 1 for the other small task), this can be “solved” increasing the granularity of the tasks.

Number of cores	Time to execute
1	639780001 ns
2	320310001 ns
4	165389001 ns
8	91496001 ns
16	64018001 ns
32	64018001 ns



3.6- Version 5

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

  for (k = 0; k < N; k++) {
    //tareador_start_task("init_complex_grid_loop_k");
    for (j = 0; j < N; j++) {
      tareador_start_task("init_complex_grid_loop_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("init_complex_grid_loop_j");
    }
    //tareador_end_task("init_complex_grid_loop_k");
  }
}
```

Figure 3.6.1:

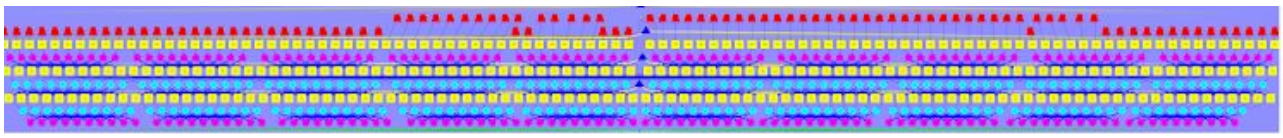
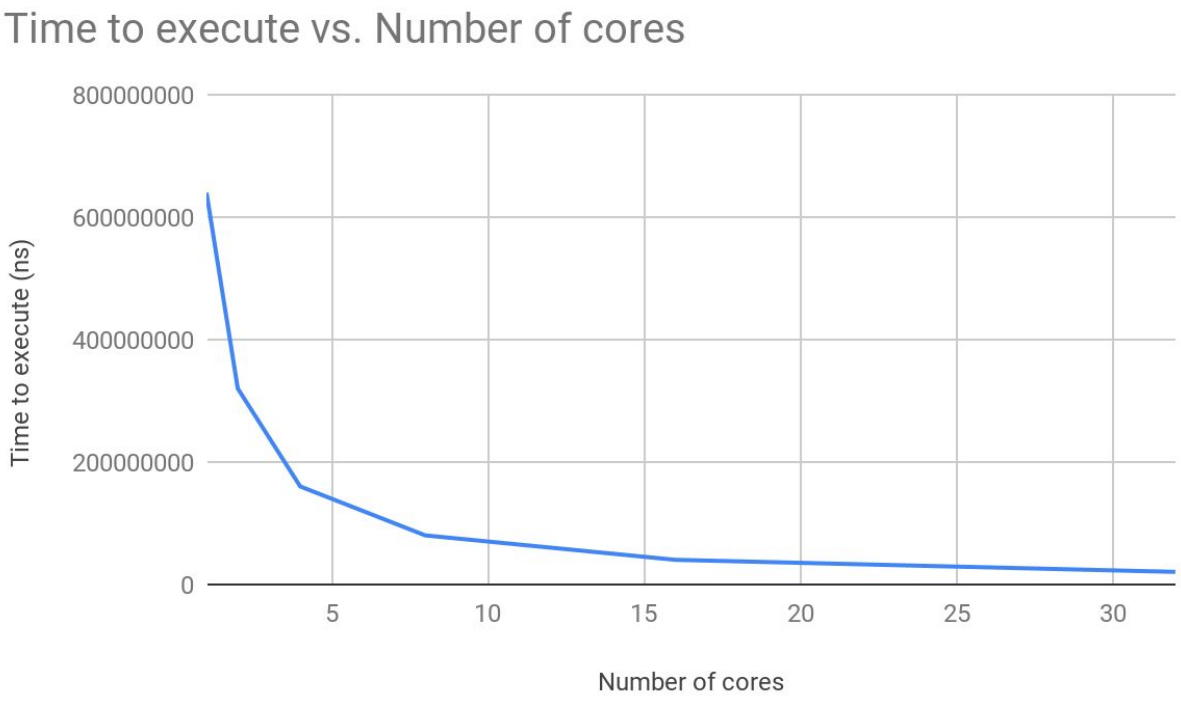


Figure 3.7.2:

In the v5 the critical path is very difficult to calculate due to the amount of tasks that we've created, but it is obvious that the critical path is smaller than the other versions (figure 3.7.2), so the Parallelism increases again. In the table below we can see that the times have improved, but reaching this level of granularity probably isn't that useful, because the overheads generated

by the creation of the tasks may be greater than the time that the tasks consume. In this version we've created tasks on the j loop of all functions (there is an example in the figure 3.6.1), so the number of tasks created increases a lot. We can have more granularity if we create the tasks in the innermost loop of all functions, but as said before, the overheads will probably render this granularity useless.

Number of cores	Time to execute
1	639780001 ns
2	320091001 ns
4	160092001 ns
8	80179001 ns
16	40264001 ns
32	20592001 ns



4- Understanding the parallel execution of 3DFFT

To trace the parallel execution we've used Paravern and the 3dfft_omp.c executing it with 1 and 8 threads. We've also modified the code two times to improve the parallel fraction and to reduce the parallelisation overheads.

Version	φ	S_∞	T1	T8	S8
Initial version in 3dfft_omp.c	0.603	2.5	2,654,773,560 ns	1,371,653,353 ns	193%
New version with improved φ	0.904	10	2,386,928,026 ns	844,526,351 ns	282%
Final version with reduced parallelisation overheads	0.903	10	2,327,011,080 ns	644,168,575 ns	361%

4.1- Original version

In this version we can see that there are defined three different tasks, one in the k loop of the transpose_zx_planes function, the second in the k loop of the transpose_xy_planes function and the third one in the k loop of the ffts1_planes function.

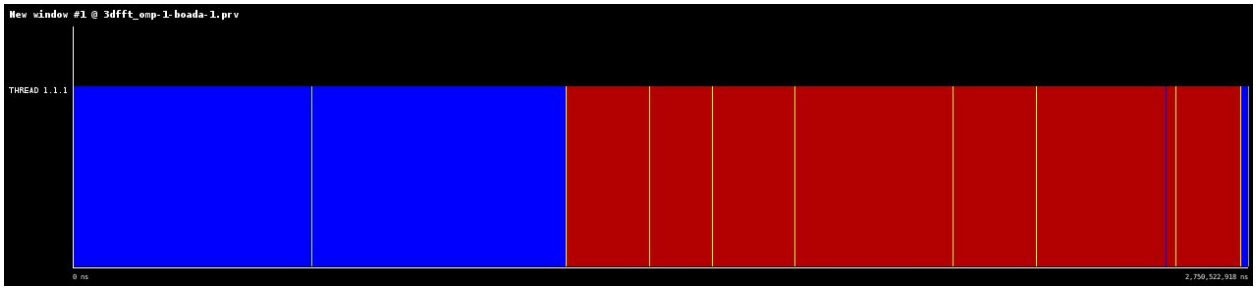


Figure 4.1: Execution for 1 thread

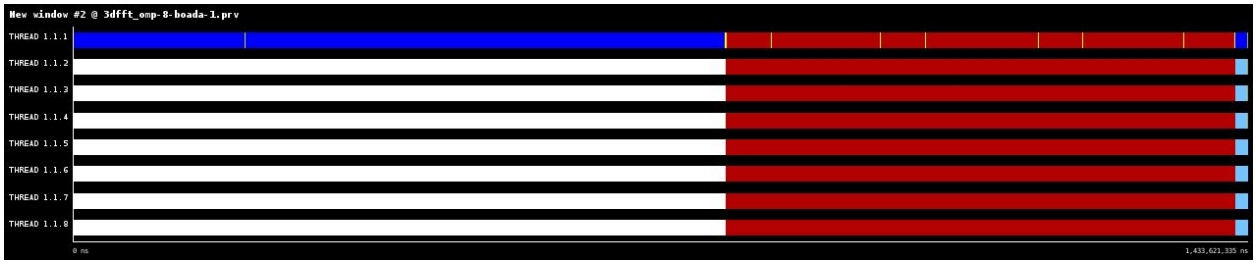


Figure 4.2: Execution for 8 threads

	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	1,250,060,362 ns	-	182,074,216 ns	1,213,865 ns	271,071 ns	1,821 ns
THREAD 1.1.2	508,665,379 ns	795,851,670 ns	111,614,225 ns	-	241,699 ns	-
THREAD 1.1.3	425,655,034 ns	795,805,369 ns	194,671,183 ns	-	218,910 ns	-
THREAD 1.1.4	505,349,829 ns	795,851,752 ns	114,928,745 ns	-	240,110 ns	-
THREAD 1.1.5	608,244,976 ns	795,861,629 ns	12,010,323 ns	-	240,266 ns	-
THREAD 1.1.6	501,886,159 ns	795,894,434 ns	118,338,806 ns	-	275,861 ns	-
THREAD 1.1.7	502,873,211 ns	795,939,382 ns	117,305,595 ns	-	230,285 ns	-
THREAD 1.1.8	607,765,149 ns	795,974,347 ns	12,375,981 ns	-	246,325 ns	-
Total	4,910,500,099 ns	5,571,178,583 ns	863,319,074 ns	1,213,865 ns	1,964,527 ns	1,821 ns
Average	613,812,512.38 ns	795,882,654.71 ns	107,914,884.25 ns	1,213,865 ns	245,565.88 ns	1,821 ns
Maximum	1,250,060,362 ns	795,974,347 ns	194,671,183 ns	1,213,865 ns	275,861 ns	1,821 ns
Minimum	425,655,034 ns	795,805,369 ns	12,010,323 ns	1,213,865 ns	218,910 ns	1,821 ns
StDev	246,965,898.81 ns	53,578.25 ns	62,860,313.11 ns	0 ns	17,978.22 ns	0 ns
Avg/Max	0.49	1.00	0.55	1	0.89	1

Figure 4.3: Total statistics

	[53,687.6..54,128.8)	[54,128.8..54,570)	[55,011.3..55,452.5)	[62,071.1..62,512.4)	[123,845..124,286)	[133,111..133,552)	[137,523..137,964)
THREAD 1.1.1	53,688.20 us	54,211 us	55,232.53 us	62,166.31 us	124,007.66 us	133,299.04 us	137,733.12 us
THREAD 1.1.2	53,687.66 us	54,210.92 us	55,176.06 us	62,165.73 us	124,007.73 us	133,298.67 us	137,732.84 us
THREAD 1.1.3	53,688.17 us	54,210.48 us	55,222.19 us	62,166.09 us	124,007.55 us	133,298.81 us	137,732.92 us
THREAD 1.1.4	53,687.67 us	54,210.79 us	55,175.66 us	62,165.36 us	124,007.31 us	133,298.85 us	137,732.93 us
THREAD 1.1.5	53,687.57 us	54,211.03 us	55,166.03 us	62,165.93 us	123,993.28 us	133,298.74 us	137,732.73 us
THREAD 1.1.6	53,688.80 us	54,211.92 us	55,133.21 us	62,165.54 us	123,993.74 us	133,298.80 us	137,732.96 us
THREAD 1.1.7	53,687.81 us	54,211.30 us	55,088.30 us	62,165.98 us	123,992.64 us	133,300.38 us	137,732.40 us
THREAD 1.1.8	53,687.59 us	54,211.04 us	55,053.29 us	62,165.32 us	123,992.37 us	133,298.79 us	137,732.72 us
Total	429,503.47 us	433,688.48 us	441,247.27 us	497,326.26 us	992,002.29 us	1,066,392.07 us	1,101,862.63 us
Average	53,687.93 us	54,211.06 us	55,155.91 us	62,165.78 us	124,000.29 us	133,299.01 us	137,732.83 us
Maximum	53,688.80 us	54,211.92 us	55,232.53 us	62,166.31 us	124,007.73 us	133,300.38 us	137,733.12 us
Minimum	53,687.57 us	54,210.48 us	55,053.29 us	62,165.32 us	123,992.37 us	133,298.67 us	137,732.40 us
StDev	0.40 us	0.39 us	57.86 us	0.33 us	7.29 us	0.53 us	0.20 us
Avg/Max	1.00	1.00	1.00	1.00	1.00	1.00	1.00

Figure 4.4: Parallel duration statistics

4.2- Version 1 (improved parallel fraction)

In this version we've added another task, this one in the k loop of the init_complex_grid function. Doing this we can improve the parallel fraction, because the Tseq is reduced and the Tpar is increased, as we can see comparing figures 4.5 and 4.6 with 4.1 and 4.2, but the overheads due to the creation of parallel tasks increase, as we can see comparing figure 4.7 with 4.3, by looking at the scheduling and forking column.

Figure 4.5: Execution for 1 thread

Figure 4.6: Execution for 8 threads

	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	730,139,599 ns	-	78,943,681 ns	1,306,491 ns	290,282 ns	1,781 ns
THREAD 1.1.2	588,748,972 ns	187,543,043 ns	14,616,798 ns	-	270,890 ns	-
THREAD 1.1.3	516,758,738 ns	187,469,170 ns	86,679,974 ns	-	243,722 ns	-
THREAD 1.1.4	510,133,362 ns	187,420,186 ns	93,355,025 ns	-	247,126 ns	-
THREAD 1.1.5	584,939,429 ns	188,221,153 ns	17,747,694 ns	-	267,212 ns	-
THREAD 1.1.6	579,931,016 ns	188,312,130 ns	22,664,635 ns	-	292,451 ns	-
THREAD 1.1.7	582,190,334 ns	188,311,659 ns	20,405,606 ns	-	273,305 ns	-
THREAD 1.1.8	578,167,855 ns	188,311,694 ns	24,428,421 ns	-	268,623 ns	-
Total	4,671,009,305 ns	1,315,589,035 ns	358,841,834 ns	1,306,491 ns	2,153,611 ns	1,781 ns
Average	583,876,163.12 ns	187,941,290.71 ns	44,855,229.25 ns	1,306,491 ns	269,201.38 ns	1,781 ns
Maximum	730,139,599 ns	188,312,130 ns	93,355,025 ns	1,306,491 ns	292,451 ns	1,781 ns
Minimum	510,133,362 ns	187,420,186 ns	14,616,798 ns	1,306,491 ns	243,722 ns	1,781 ns
StDev	62,661,399.54 ns	404,133.16 ns	32,442,649.86 ns	0 ns	16,376.21 ns	0 ns
Avg/Max	0.80	1.00	0.48	1	0.92	1

Figure 4.7: Total statistics

	[36,481.1..37,079.6)	[59,824.3..60,422.9)	[92,744.3..93,342.9)	[93,342.9..93,941.4)	[93,941.4..94,540)	[95,138.5..95,737.1)	[150,205..150,803)
THREAD 1.1.1	109,972.44 us	59,887.98 us	-	93,657.11 us	94,069.32 us	95,431.20 us	150,489.71 us
THREAD 1.1.2	109,973.27 us	59,888.32 us	-	93,515.45 us	94,068.69 us	95,430.69 us	150,489.34 us
THREAD 1.1.3	109,971.93 us	59,887.99 us	-	93,589.44 us	94,069.12 us	95,431.04 us	150,489.20 us
THREAD 1.1.4	109,972.34 us	59,887.87 us	-	93,638.54 us	94,069.04 us	95,431.02 us	150,489.59 us
THREAD 1.1.5	109,972.40 us	59,888.32 us	92,837.36 us	-	94,068.77 us	95,430.93 us	150,489.35 us
THREAD 1.1.6	109,972.53 us	59,888.53 us	92,746.49 us	-	94,068.51 us	95,430.57 us	150,489.03 us
THREAD 1.1.7	109,972.24 us	59,887.93 us	92,746.90 us	-	94,068.69 us	95,430.77 us	150,489.42 us
THREAD 1.1.8	109,972.60 us	59,887.79 us	92,747.10 us	-	94,068.76 us	95,430.74 us	150,489.29 us
Total	879,779.74 us	479,104.71 us	371,077.84 us	374,400.54 us	752,550.90 us	763,446.97 us	1,203,914.92 us
Average	109,972.47 us	59,888.09 us	92,769.46 us	93,600.13 us	94,068.86 us	95,430.87 us	150,489.36 us
Maximum	109,973.27 us	59,888.53 us	92,837.36 us	93,657.11 us	94,069.32 us	95,431.20 us	150,489.71 us
Minimum	109,971.93 us	59,887.79 us	92,746.49 us	93,515.45 us	94,068.51 us	95,430.57 us	150,489.03 us
StDev	0.36 us	0.25 us	39.20 us	54.79 us	0.25 us	0.20 us	0.20 us
Avg/Max	1.00	1.00	1.00	1.00	1.00	1.00	1.00

Figure 4.8: Parallel duration statistics

4.3- Version 2 (reduced overheads)

In this version we've moved up all the pragma calls, doing so we create only a task for each function instead of k tasks. Doing so we can reduce the overheads produced by the creation of tasks without changing a lot the parallel fraction (as we can see in the figure 4.10), because even though we don't have as many tasks as before (we have less granularity), the Tpar and the Tseq are the same (or almost the same).

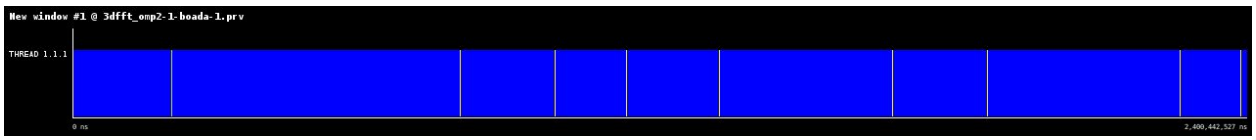


Figure 4.9: Execution for 1 thread

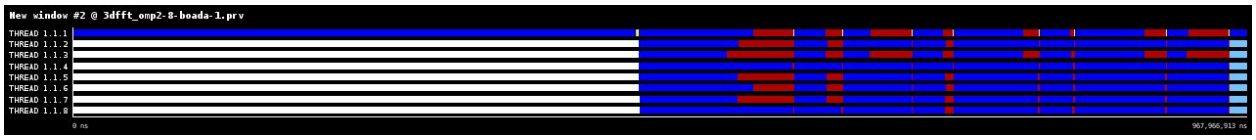


Figure 4.10: Execution for 8 threads

	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	812,127,450 ns	-	154,813,620 ns	988,002 ns	34,953 ns	2,888 ns
THREAD 1.1.2	424,619,529 ns	465,559,358 ns	62,101,882 ns	-	7,904 ns	-
THREAD 1.1.3	308,572,904 ns	465,551,881 ns	178,158,382 ns	-	5,476 ns	-
THREAD 1.1.4	485,225,715 ns	465,560,022 ns	1,494,847 ns	-	5,747 ns	-
THREAD 1.1.5	421,038,861 ns	465,567,237 ns	65,674,545 ns	-	5,332 ns	-
THREAD 1.1.6	434,163,580 ns	465,576,301 ns	52,540,545 ns	-	5,347 ns	-
THREAD 1.1.7	420,083,194 ns	465,919,435 ns	66,277,956 ns	-	5,310 ns	-
THREAD 1.1.8	477,542,577 ns	466,069,811 ns	8,667,473 ns	-	5,572 ns	-
Total	3,783,373,810 ns	3,259,804,045 ns	589,729,250 ns	988,002 ns	75,641 ns	2,888 ns
Average	472,921,726.25 ns	465,686,292.14 ns	73,716,156.25 ns	988,002 ns	9,455.12 ns	2,888 ns
Maximum	812,127,450 ns	466,069,811 ns	178,158,382 ns	988,002 ns	34,953 ns	2,888 ns
Minimum	308,572,904 ns	465,551,881 ns	1,494,847 ns	988,002 ns	5,310 ns	2,888 ns
StDev	137,642,125.25 ns	199,226.10 ns	58,774,482.20 ns	0 ns	9,671.28 ns	0 ns
Avg/Max	0.58	1.00	0.41	1	0.27	1

Figure 4.11: Total statistics

	[28,902.1..29,422)	[33,061.7..33,581.7)	[39,821.2..40,341.1)	[51,260.2..51,780.2)	[57,499.7..58,019.7)	[71,018.6..71,538.5)	[75,698.2..76,218.1)	[127,174..127,694)	[127,694..128,214)
THREAD 1.1.1	28,902.60 us	33,373.91 us	40,154.47 us	51,701.03 us	57,637.63 us	71,044.44 us	76,065.53 us	-	127,941.46 us
THREAD 1.1.2	28,902.07 us	33,373.62 us	40,154.22 us	51,700.33 us	57,636.34 us	71,043.79 us	76,064.94 us	-	127,846.10 us
THREAD 1.1.3	28,902.64 us	33,373.65 us	40,153.59 us	51,700.72 us	57,637.70 us	71,044.29 us	76,065.59 us	-	127,853.12 us
THREAD 1.1.4	28,902.16 us	33,373.55 us	40,152.85 us	51,700.69 us	57,636.92 us	71,043.86 us	76,065.05 us	-	127,845.48 us
THREAD 1.1.5	28,902.31 us	33,373.47 us	40,153.46 us	51,700.29 us	57,636.63 us	71,044.04 us	76,065.05 us	-	127,838.16 us
THREAD 1.1.6	28,902.10 us	33,373.32 us	40,153.56 us	51,700.51 us	57,636.87 us	71,043.72 us	76,064.88 us	-	127,829.17 us
THREAD 1.1.7	28,902.09 us	33,373.45 us	40,153.53 us	51,700.52 us	57,636.74 us	71,043.85 us	76,064.98 us	127,485.00 us	-
THREAD 1.1.8	28,902.08 us	33,373.33 us	40,152.91 us	51,700.62 us	57,636.72 us	71,043.91 us	76,064.80 us	127,335.68 us	-
Total	231,218.04 us	266,988.30 us	321,228.59 us	413,604.71 us	461,095.54 us	568,351.89 us	608,520.82 us	254,821.68 us	767,153.48 us
Average	28,902.26 us	33,373.54 us	40,153.57 us	51,700.59 us	57,636.94 us	71,043.99 us	76,065.10 us	127,410.84 us	127,858.91 us
Maximum	28,902.64 us	33,373.91 us	40,154.47 us	51,701.03 us	57,637.70 us	71,044.44 us	76,065.59 us	127,485.00 us	127,941.46 us
Minimum	28,902.07 us	33,373.32 us	40,152.85 us	51,700.29 us	57,636.34 us	71,043.72 us	76,064.80 us	127,335.68 us	127,829.17 us
StDev	0.22 us	0.18 us	0.52 us	0.22 us	0.45 us	0.24 us	0.27 us	75.16 us	37.66 us
Avg/Max	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

Figure 4.12: Parallel duration statistics

4.4- Strong scalability plots

In general, the tendency for the speed-up is to increase as the number of threads increase, but past some point the overheads are too costly and the speed-up decreases back. As we can see in the plots below, the speed-up for the version 1 is better than the original one, and the one in the version 2 is better than the other two (as we said before, this version has reduced overheads).

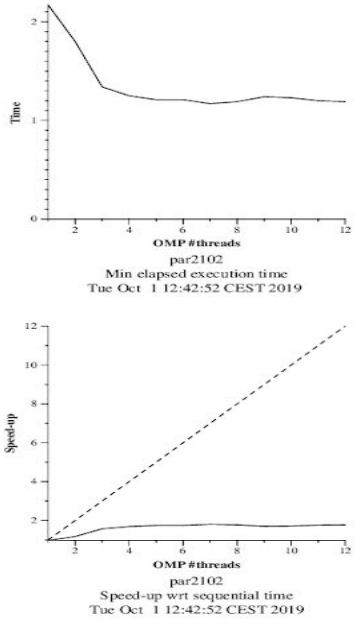


Figure 4.13: Original version plots

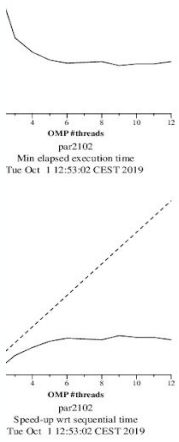


Figure 4.14: Version 1 plots

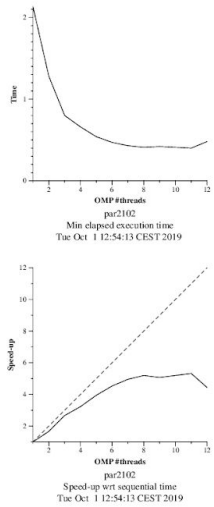


Figure 4.15: Version 2 plots

5- Conclusions

The main conclusion of this assignment is that we can increase the granularity to increase the parallelism, but in the real world there are overheads, so increasing too much the granularity will drop the efficiency of the program, so we need to find the more appropriate granularity for each problem that we want to solve.