Lab 1: Experimental setup and tools

par4111
Adrià Cabeza, Xavier Lacasa
Departament d' Arquitectura de Computadors

March 6, 2019
2018 - 19 PRIMAVERA

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Contents

1 Introduction

In order to do properly this subject, first, we have to introduce some new concepts and hardware and software environment that we will use during this semester to do all laboratory assignments. The following document contains an introductory approach, step by step introducing those concepts. We will introduce the *Boada* architecture, some of the most important parallelism concepts and several tests to see its effects.

2 Experimental setup

2.1 Node architecture and memory

Boada is a multiprocessor server located at the Computer Architecture Department divided in different nodes, each of them with different architecture and diffferent uses. *Boada* is composed of 8 nodes (from boada-1 to boada-8) and they can be grouped as the following table:

Node name	Processor generation	Interactive	Queue name
boada-1	Intel Xeon E5645	Yes	batch
boada-2 to 4	Intel Xeon E5645	No	execution
boada-5	Intel Xeon E5-2620 v2 + Nvidia K40c	No	cuida
boada-6 to 8	Intel Xeon E5-2609 v4	No	execution2

However in this course we are going to use mainly from boada-1 to boada-4. The easiest way to obtain the information of the hardware used in each node is using the linux commands lscpu and lstopo(see Figure 1 and 2). This commands can be easily executed in the boada-1 node (because it is interactive), but if we want to use the other nodes we can use the submit-*.sh script provided by the PAR professors and use the queue system.

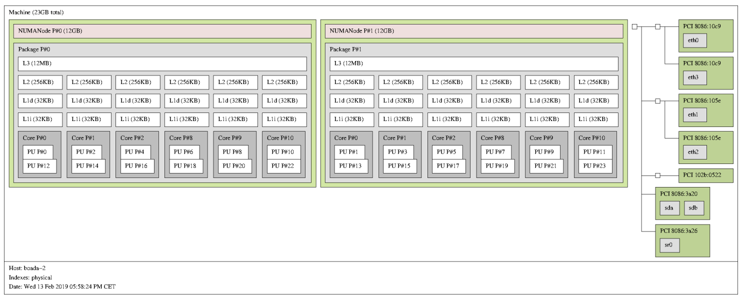


Figure 1: Boada-2 architecture outputed by lstopo.

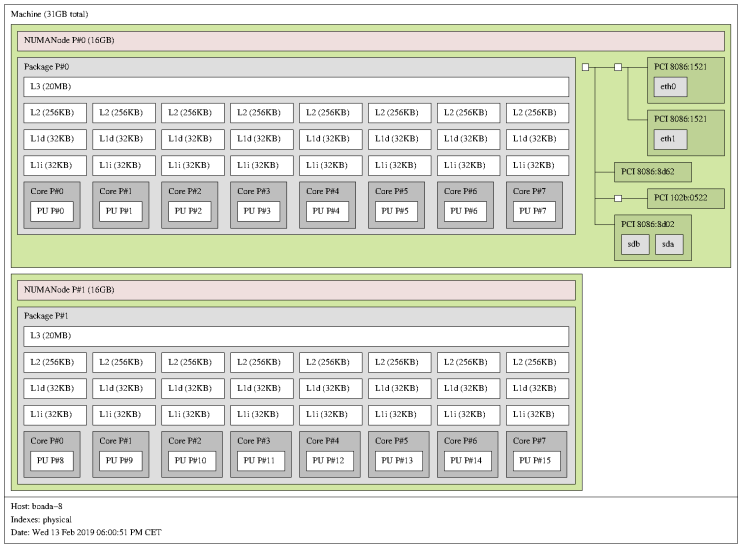


Figure 2: Boada-8 architecture outputed by lstopo.

After creating the scripts and applying them to each of the nodes, we obtained the following hardware information:

	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
L1-I cache size (per-core)	32 KB	32 KB	32 KB
L1-D cache size (per core)	32 KB	32 KB	32 KB
L2 cache size (per-core)	256 KB	256 KB	256 KB
Last-level cache size (per-socket)	12 MB	15 MB	20 MB
Main memory size (per socket)	12 GB	31 GB	16 GB
Main memory size (per node)	23 GB	63 GB	31 GB

The previous table gives us useful information that will be necessary in the future to properly use the *boada* system and understand the parallelism decomposition and time we will get.

2.2 Sequential and parallel executions

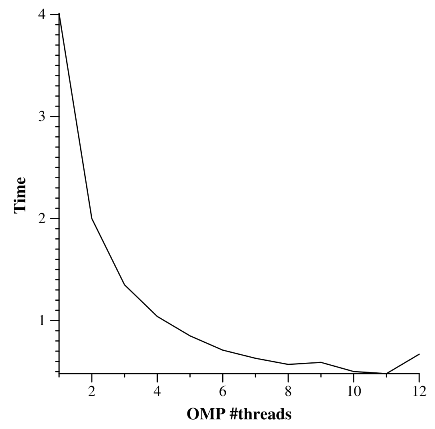
More often than not parallelism offers speed-ups in the execution time of applications. Sometimes, however, that extra speed is used to augment the problem size, which would not be possible otherwise.

In the two following sections we are going to see the differences of two different approaches to parallelism, **strong** and **weak**, applied to the *pi_omp.c* program.

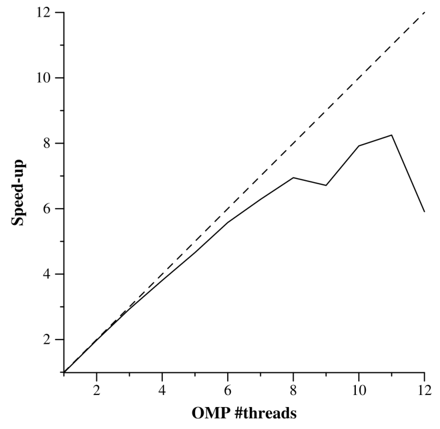
2.2.1 Strong scalability

Strong scalabilty consists in increasing the numberer of processors while keeping the problem size the same. This reduces the amount of work each processor has to do, which speeds-up the execution. Nonetheless, the speed-up is bounded by the parallelization of the program and the overhead generated when doing so. Usually a point is reached where adding processors has no further effect on the program or the overhead generated by further parallelizing the program is greater than the added speed-up.

Boada 1: For Boada 1 we can see how execution time is logarithmically reduced in Figure 3. At 11 threads the time seems to slightly increase. To be sure we can look at the speed-up plot, where we can clearly appreciate how speed-up slowly stops increasing and starts to fall down at around 11 threads. This evidence supports the previous statement about time. This is most likely caused by the overhead that parallelizing a program creates: creating new threads, synchronizing the results, etc. The graphic shows how running the program on 12 threads is actually slower than doing so on 11, and even though we cannot see it in the plots, performance would probably keep going down due to the added overhead with each new thread used.



par4111
Min elapsed execution time
Fri Mar 1 16:54:24 CET 2019

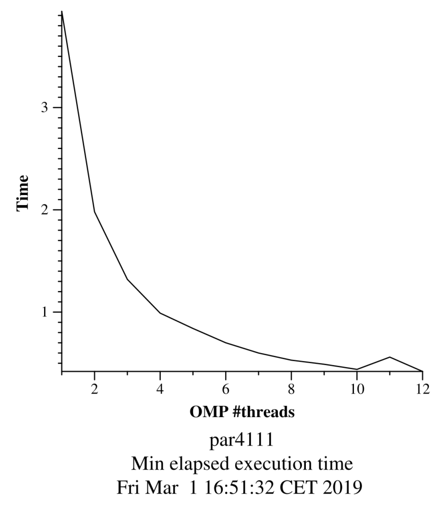


par4111
Speed-up wrt sequential time
Fri Mar 1 16:54:24 CET 2019

Figure 3: *pi_omp* with strong scalability by Boada-1

Boada 2 to 4: Since Boada 1 has the same architecture as 2, 3 and 4, it is fair to expect a very similar performance, and in fact that is what Figure 4 shows. However, Boada 4 speed-up does not quite decrease from using 11 threads to 12, although it certainly does from 10 to 11. Again, it would probably keep decreasing from that point on due to the overhead, but there is an important improvement up until around 10 threads.

It would be fair to say that for the *pi_omp.c* program running Boada 1 to 4, up to 10 threads are beneficial to reduce execution time, but at around 5 threads speed-up starts to increase more slowly at what seems like a logarithmic rate.



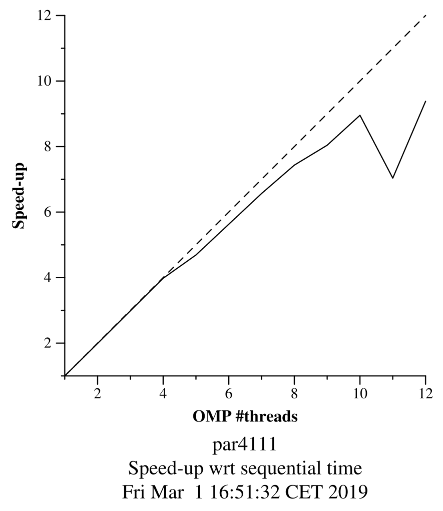
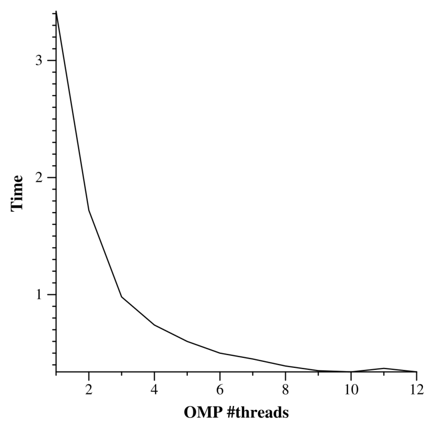
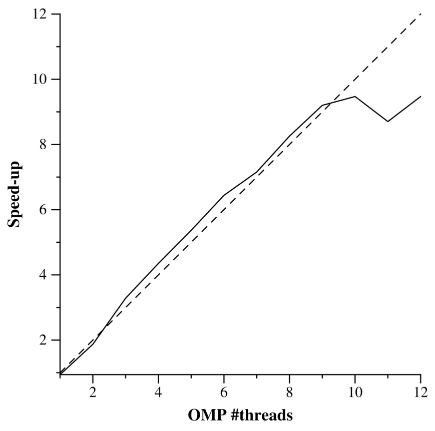


Figure 4: *pi_omp* with strong scalability by Boada-4

Boada 5: Boada 5 uses a different processor, but the overhead principle should still be present because the generated overhead will start to overcome the parallelization speed-up at some point. In fact, looking at the generated graphics, the loss begins at approximately the same number of threads than the previous Boadas (1-4).



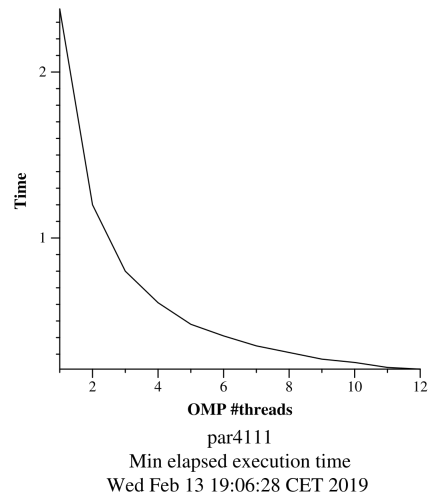
par4111
Min elapsed execution time
Fri Mar 1 16:54:17 CET 2019



par4111
Speed-up wrt sequential time
Fri Mar 1 16:54:17 CET 2019

Figure 5: *pi_omp* with strong scalability by Boada-5

Boada 6 to 8: Boada 6 to 8 use again a different processor. This time, however, the speed-up and time plots look quite different from the previous ones. There is no appreciable decrease in speed-up even up to 12 threads. We speculate that this could be of their processors being newer (Boada 6-8 have Intel Xeon E5-2609 v4), which according to Intel's website have a newer instruction set (Intel® AVX2) and Intel® Transactional Synchronization Extensions New Instructions, which make parallel operations more efficient. Again, we are not sure, it is just a possibility we came up with after doing some research about each Boada's processor architecture.
Nevertheless, speed-up should still decay later on if we kept increasing the number of threads, because a point will be reached where the added overhead will surpass the time-reduction gained by parallelism.



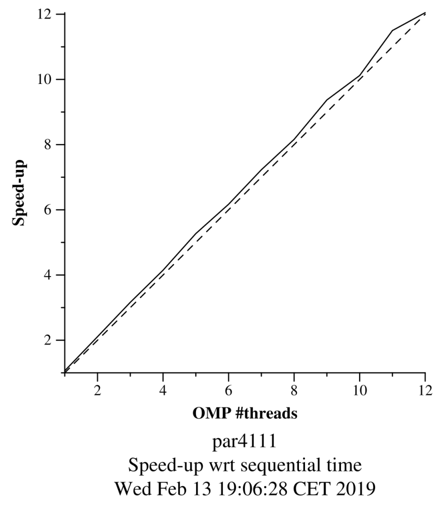


Figure 6: *pi_omp* with strong scalability by Boada-8

2.2.2 Weak scalability

Weak scalability takes a different approach. It takes advantage of the additional power gained by parallelizing the program to increase the problem size proportionally to the number of threads, so that while the speed-up stays more or less the same, the work done increases.

Boada 1: For Boada 1 with weak scalability the graphic clearly shows how speed-up stays mostly the same at first. This is due to increasing the problem size proportionally to the number of threads. At around 9 threads, however, the speed-ups starts to decrease similarly to strong scalability. The reason is the same: excess overhead created by parallelization. To keep speed-up the same from this point on we should start to increase the problem size more slowly while keeping the thread number increment the same.

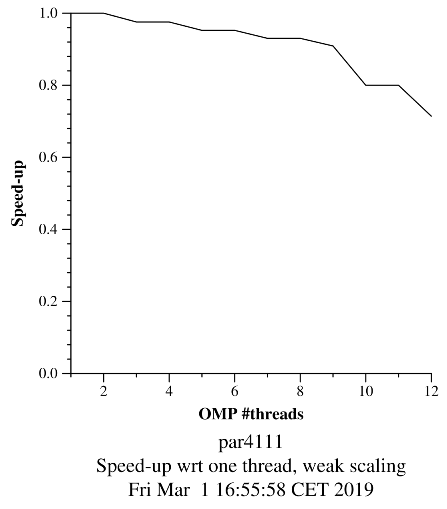


Figure 7: *pi_omp* with weak scalability by Boada-1

Boada 2 to 4: As with strong scalability, Boada 2 to 4 produce a very similar result to Boada 1 due to them having the same processor. There is not much to add with respect to Boada 1.

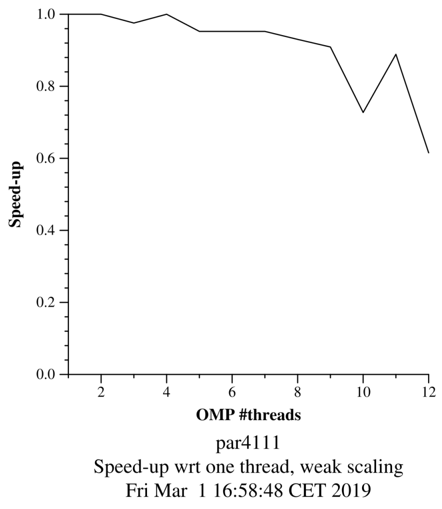


Figure 8: *pi_omp* with weak scalability by Boada-3

Boada 5: For Boada 5 we observe a similar behaviour, but this time speed-up goes over 1, meaning the problem size could be increased faster if we wanted to keep speed-up steady. In the end, however, overhead still counters the extra processors and speed-up goes down at around 10-12 threads.

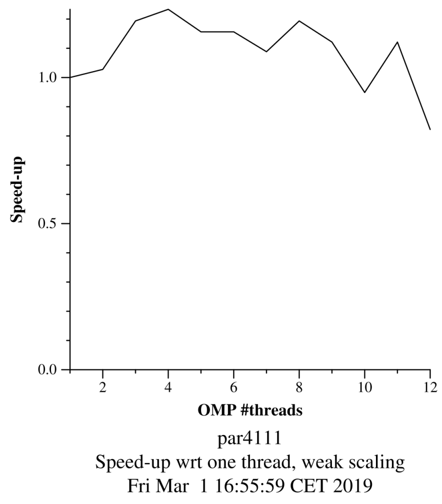


Figure 9: *pi_omp* with weak scalability by Boada-5

Boada 6 to 8: With Boada 6 to 8 we encounter the same situation as in strong scalability. There is no clear sign of overhead occuring in Figure 10. Again, it would happen if we kept increasing the number of threads assuming the program could use them all, but up to 12 threads there is not a real downside of parallelizing the program too much.

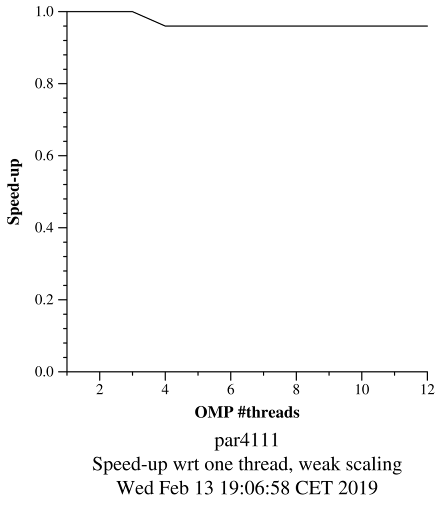


Figure 10: *pi_omp* with weak scalability by Boada-6

3 Systematically analysing task decompositions with Tareador

3.1 Introduction

The objective of this laboratory is learn how to use *Tareador*, an environment to analyse the potential parallelilsm that can be obtained when a certain task decomposition is applied to a code. We will introduce how it works and we will experiment and analyse decomposition with a sequential code called 3DFFT.

3.2 Analysis of task decompositions for 3DFFT

Once we have seen the basic features in *Tareador* we can now proceed to explore new tasks decompositions for a piece of code. Down below we will incrementally generate five new task decompositions and the potential parallelism (T_1/T_∞) from the task dependence graph generated by *Tareador*.
To obtain T_∞ we will assume that each instruction takes one time unit to execute and simulate the execution of the graph with a large number of processors.

Once we have created those tasks, we can visualize the dependency graph using *Tareador*. Each node of the graph represents a task: diferents shapes and colours are used to identify task instances and each one is labeled with a task instance number and some important information like the number of instructions. Also the size of the shape reflects in some way its granularity.

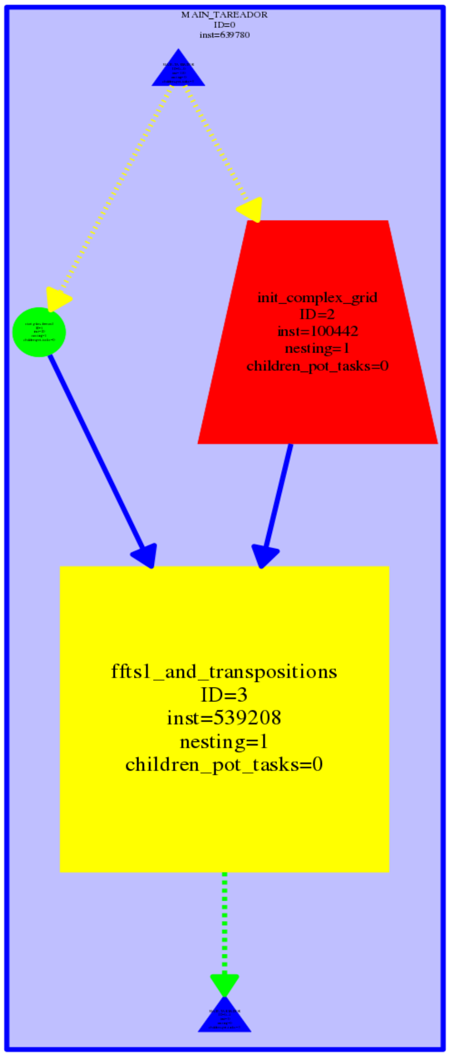


Figure 11: Dependency graph for the original version.

3.2.1 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("0");
    ffts1_planes(p1d, in_fftw);
```

```
tareador_end_task("0");

 tareador_start_task ("1");
transpose_xy_planes(tmp_fftw,  in_fftw );
tareador_end_task("1");

 tareador_start_task ("2");
 ffts1_planes (p1d,  tmp_fftw);
tareador_end_task("2");

 tareador_start_task ("3");
transpose_zx_planes( in_fftw ,  tmp_fftw);
tareador_end_task("3");

 tareador_start_task ("4");
 ffts1_planes (p1d,  in_fftw );
tareador_end_task("4");

 tareador_start_task ("5");
transpose_zx_planes(tmp_fftw,  in_fftw );
tareador_end_task("5");

 tareador_start_task ("6");
transpose_xy_planes( in_fftw ,  tmp_fftw);
tareador_end_task("6");
```

Once we have created all these tasks, we have to execute the script *./run-tareador.sh VERSION1* and visualize the task dependence graph, see Figure 12. As we can see, comparing the original graph, see Figure **??**, now the shape that was associated to `ffts1_and_transpositions` has now been divided into several other shapes which represents more granularity.

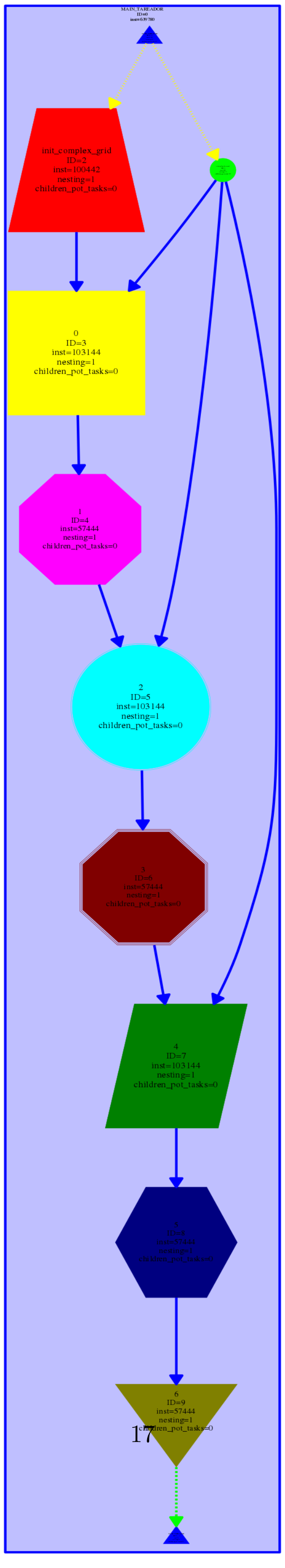


Figure 12: Dependency graph for the first version.

3.2.2 Version 2

The second version, starting from the first one, consists in replacing the definition of tasks associated to function invocations `ffts1_planes` with fine-grained tasks defined inside the function body and associated to individual iterations of the k loop. 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_loop_k");
     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_loop_k");
    }
}


int main(){
...
    tareador_start_task("1");
    transpose_xy_planes(tmp_fftw, in_fftw);
    tareador_end_task("1");

    ffts1_planes(p1d, tmp_fftw);

    tareador_start_task("3");
    transpose_zx_planes(in_fftw, tmp_fftw);
    tareador_end_task("3");

    ffts1_planes(p1d, in_fftw);

    tareador_start_task("5");
    transpose_zx_planes(tmp_fftw, in_fftw);
    tareador_end_task("5");

    tareador_start_task("6");
    transpose_xy_planes(in_fftw, tmp_fftw);
    tareador_end_task("6");
...
}
```

As we can see in the outputed dependency graph given by the *Tareador* our dependency graph has now changed and there are several more shapes. The task that previously was `ffts1_planes` has been divided into several more.

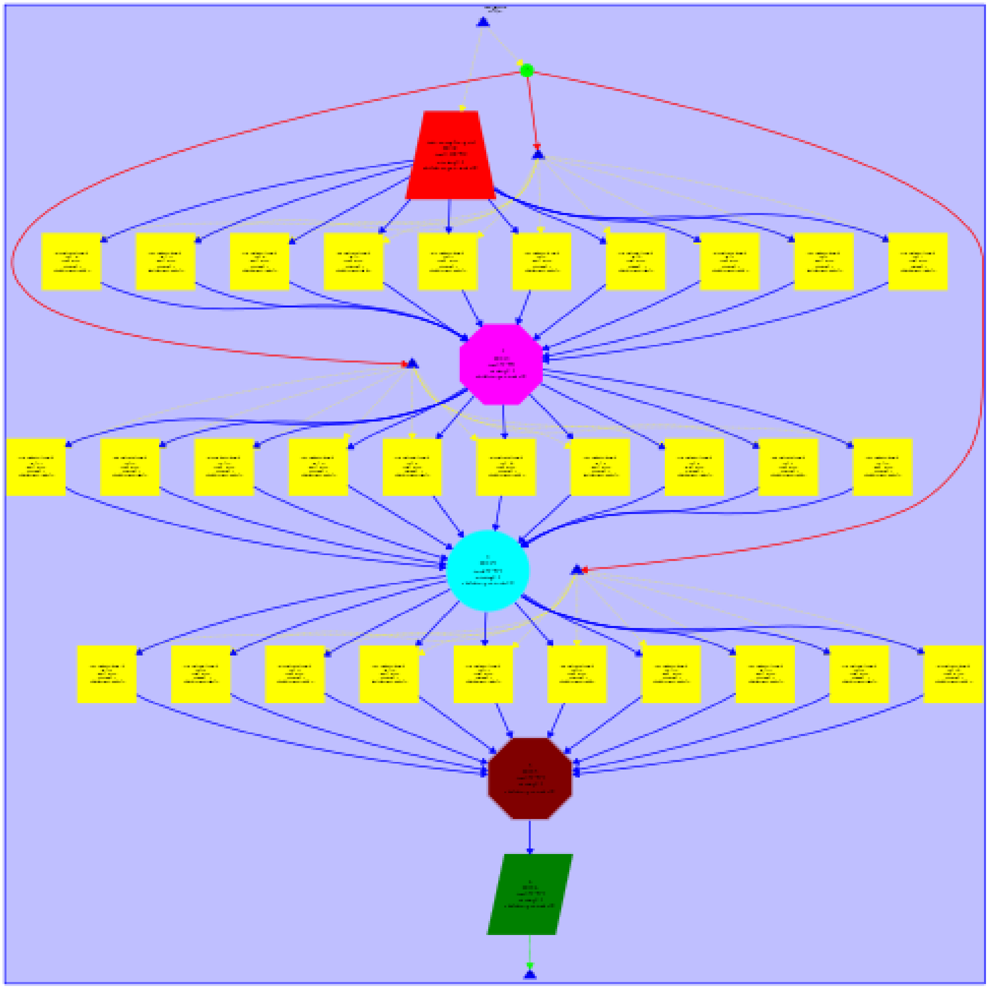


Figure 13: Dependency graph for the second version.

3.2.3 Version 3

The third version, starting from the second one, consists in replacing the definition of tasks associated to function invocations `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 it was made in the second version.

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

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("transpose_zx_planes_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("transpose_zx_planes_loop_k");

    }
}

int main(){
 ...
    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;
    ffts1_planes (p1d,  in_fftw );
    transpose_xy_planes(tmp_fftw,  in_fftw );
    ffts1_planes (p1d, tmp_fftw);
    transpose_zx_planes( in_fftw ,  tmp_fftw);
    ffts1_planes (p1d,  in_fftw );
    transpose_zx_planes(tmp_fftw,  in_fftw );
    transpose_xy_planes( in_fftw ,  tmp_fftw);
 ...
}
```

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

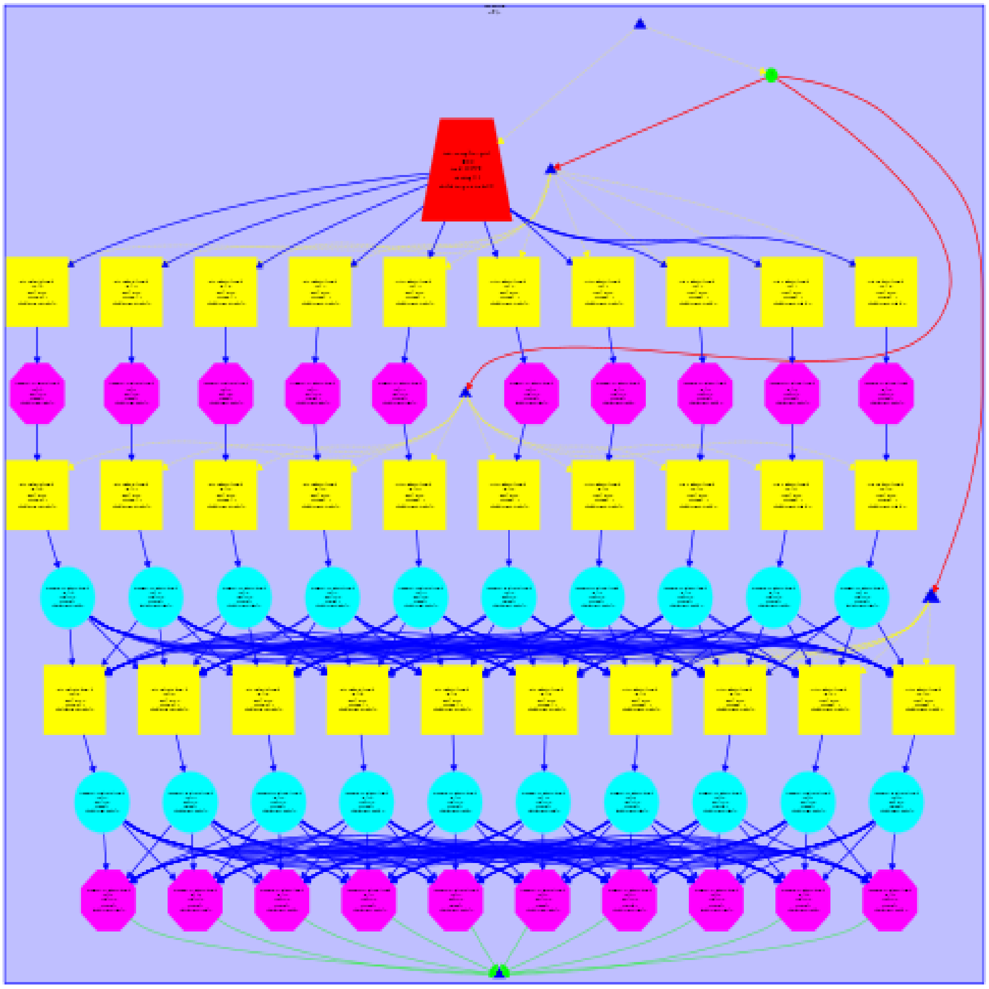


Figure 14: Dependency graph for the third version.

3.2.4 Version 4

The forth version, starting from the third one, consists in replacing the definition of tasks associated to function invocations `init_complex_grid` with fine-grained tasks inside the corresponding body functions and associated to individual iterations of the k loop, similarly it was made in the previous version.

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

  for (k = 0; k < N; k++) {
    tareador_start_task ("transpose_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("transpose_init_complex_grid_loop_k");
 }
}
int main(){
 ...

    init_complex_grid(in_fftw);
    STOP_COUNT_TIME("Init Complex Grid FFT3D");

    START_COUNT_TIME;

     ffts1_planes(p1d, in_fftw);
    transpose_xy_planes(tmp_fftw, in_fftw);
     ffts1_planes(p1d, tmp_fftw);
    transpose_zx_planes(in_fftw, tmp_fftw);
     ffts1_planes(p1d, in_fftw);
    transpose_zx_planes(tmp_fftw, in_fftw);
    transpose_xy_planes(in_fftw, tmp_fftw);
 ...
}
```

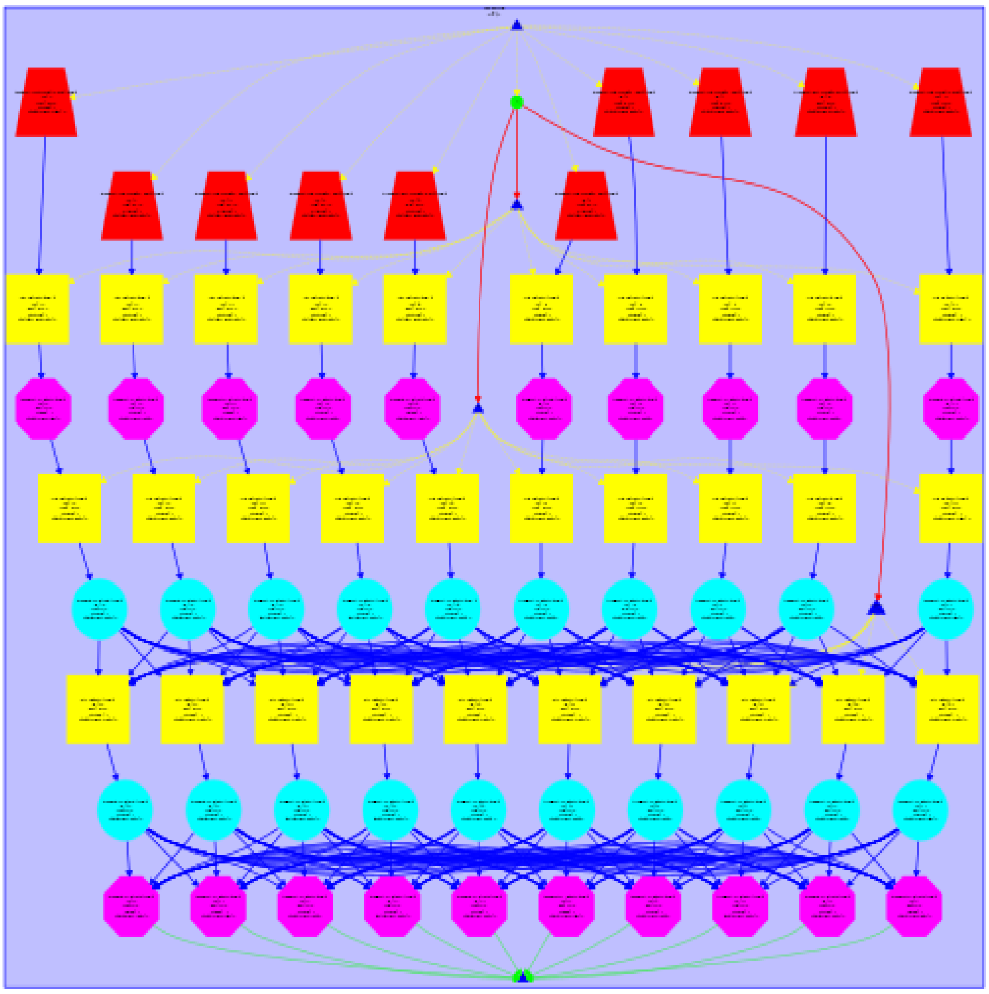


Figure 15: Dependency graph for the forth version.

3.2.5 Version 5

This is the final version; in this version we will explore even more finer-grained tasks. In order to continue this task we observed the forth figure given by *Tareador* which corresponds to the forth version of the code.

As we can see in the Figure **??** the task granularity that has less granularity is the *ffts1_planes_loop_k* with 10305 instructions. So we deepen in the code of the corresponding function and we created tasks in a loop deeper than our first approach.

```
void ffts1_planes(fftwf_plan p1d, fftwf_complex in_fftw[][N][N]) {
    int k,j;
    for (k=0; k<N; k++) {
     for (j=0; j<N; j++) {
        tareador_start_task (" ffts1_planes_loop_j ");

          fftwf_execute_dft ( p1d, (fftwf_complex *) in_fftw [k][j ][0],  (
               fftwf_complex *) in_fftw [k][j ][0]) ;
    tareador_end_task(" ffts1_planes_loop_j ");
          }

    }
}
```

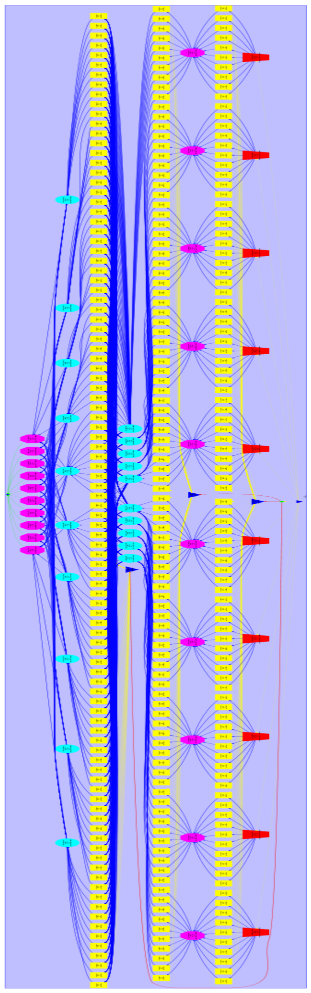


Figure 16: Dependency graph for the final version.

3.2.6 Comparison between version 4 and version 5

Time comparison (in ns) between version 4 and version 5

Number of processors

	1	2	4	8	16	32
v4	639.780.001	320.310.001	165.389.001	91.496.001	64.018.001	64.018.001
v5	639.780.001	321.493.001	172.584.001	99.126.001	53.554.001	44.356.001

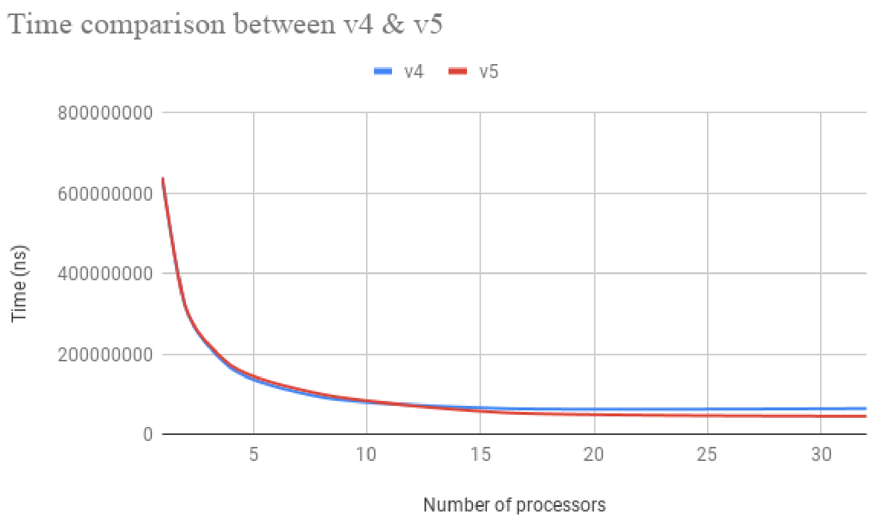


Figure 17: Time comparison between v4 and v5.

Speedup between version 4 and version 5

Number of processors

	1	2	4	8	16	32
Speedup	1	0.99632029	0.95831015	0.92302725	1.1953915	1.4432771

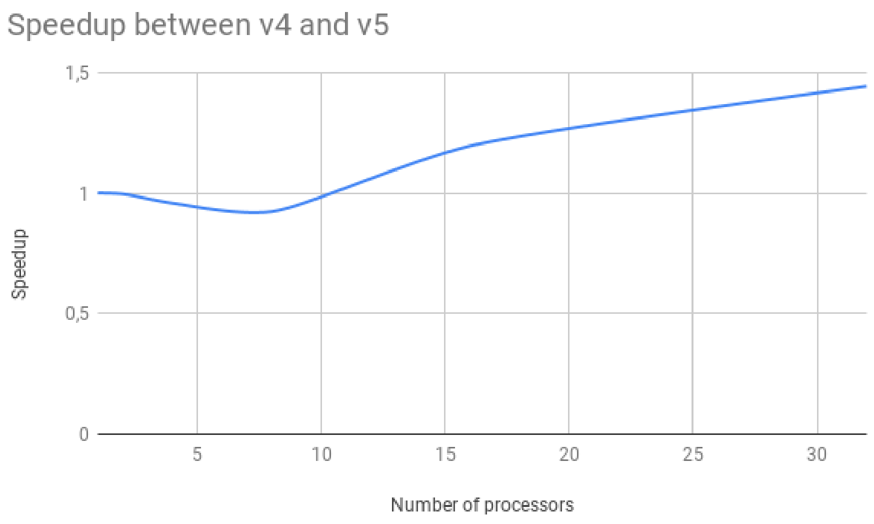


Figure 18: Speedup between v4 and v5.

From this two plots we can observe that there's a moment when it does not matter how many processors you use, you cannot improve the execution time we can get. In the case of the fourth version we can see it clearly comparing the time we get with 16 processors and the time we get with 32.

Moreover, there is still another interesting point we can still analyse. Even though the fifth version had more finer-grained tasks (check Figure 16 when we use less than 16 processors v4 gets a better time execution. That could be justified with the time is focused for making the finer-grained tasks work together because of the overhead.

3.2.7 Summary

To sum up, we show a table with all the data obtained with our experiments. We can see that in all the versions excepting for the first ones, we increase the parallelism.

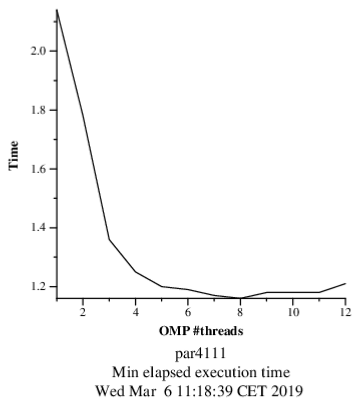
Version	T_1	T_∞	Parallelism
seq	639,780,001 ns	639,707,001 ns	1.00011411474
v1	639,780,001 ns	639,707,001 ns	1.00011411474
v2	639,780,001 ns	361,190,001 ns	1.77131149597
v3	639,780,001 ns	154,354,001 ns	4.14488770524
v4	639,780,001 ns	64,018,001 ns	9.99375161683
v5	639,780,001 ns	38,224,001 ns	16.7376513254

4 Understanding the parallel execution

In this final section we have used *Paraver* and several of its configurations to be able to understand better differently parallelized versions of the *3dfft_omp* program

4.1 Initial version

The first version we have tried is the one given to us, which is already parallelized partially. The for loop inside the init_complex_grid function is not parallelized. As seen in the table below, this will cause the parallel fraction to not be as high as the following versions. Nonetheless, there is still a good improvement from using 8 threads instead of 1, execution time has been 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. In Figure 19 we can appreciate how 8 threads have actually the best performance, and if we keep increasing the number of threads then overhead starts to make speed-up decay.



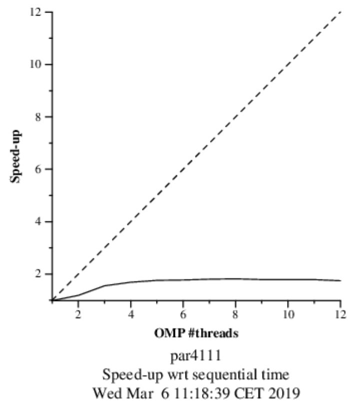


Figure 19: Strong scalability plot of the first version of 3dfft_omp.c

Following are screen captures of *Paraver* state config, timeline and parallel functions durations config (the latter to get $Tpar$):

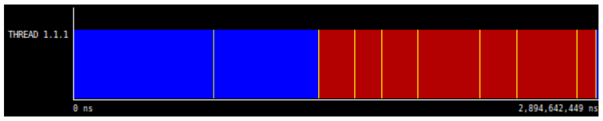
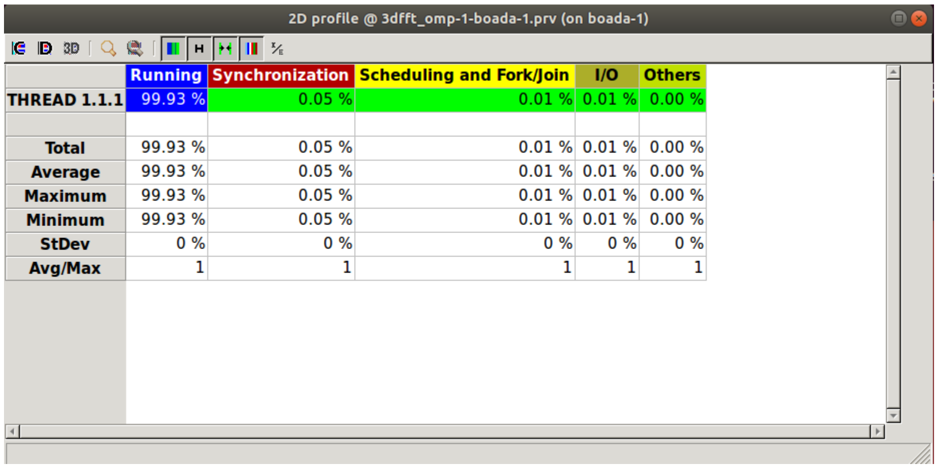


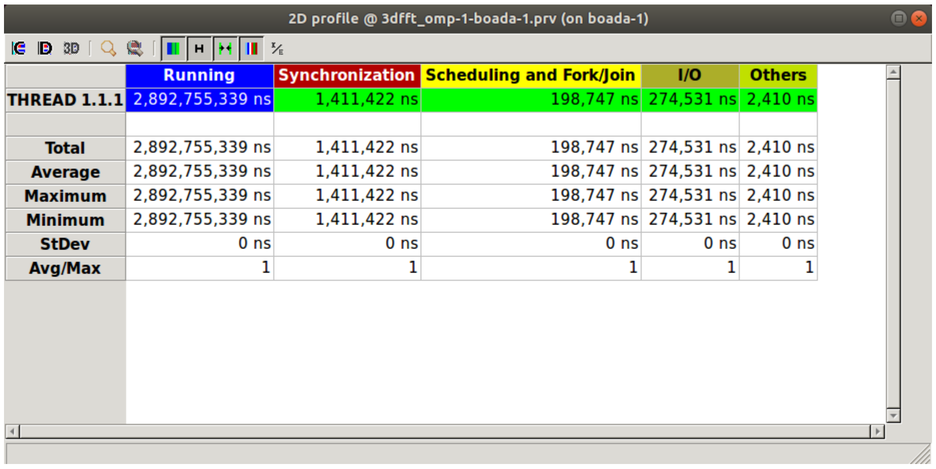
Figure 20: Timeline of the first version of 3dfft_omp.c with 1 thread



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

	Running	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	99.93 %	0.05 %	0.01 %	0.01 %	0.00 %
Total	99.93 %	0.05 %	0.01 %	0.01 %	0.00 %
Average	99.93 %	0.05 %	0.01 %	0.01 %	0.00 %
Maximum	99.93 %	0.05 %	0.01 %	0.01 %	0.00 %
Minimum	99.93 %	0.05 %	0.01 %	0.01 %	0.00 %
StDev	0 %	0 %	0 %	0 %	0 %
Avg/Max	1	1	1	1	1

Figure 21: State config with (% time) of the first version of 3dfft_omp.c with 1 thread



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

	Running	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	2,892,755,339 ns	1,411,422 ns	198,747 ns	274,531 ns	2,410 ns
Total	2,892,755,339 ns	1,411,422 ns	198,747 ns	274,531 ns	2,410 ns
Average	2,892,755,339 ns	1,411,422 ns	198,747 ns	274,531 ns	2,410 ns
Maximum	2,892,755,339 ns	1,411,422 ns	198,747 ns	274,531 ns	2,410 ns
Minimum	2,892,755,339 ns	1,411,422 ns	198,747 ns	274,531 ns	2,410 ns
StDev	0 ns	0 ns	0 ns	0 ns	0 ns
Avg/Max	1	1	1	1	1

Figure 22: State config with (time) of the first version of 3dfft_omp.c with 1 thread

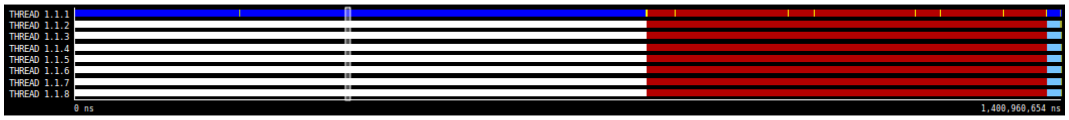
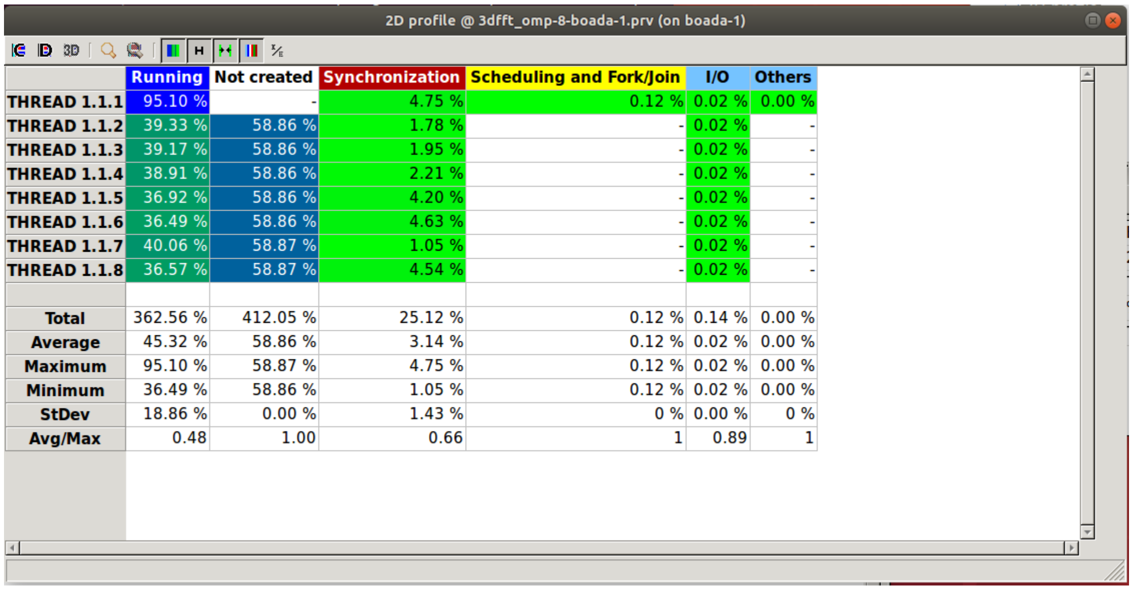


Figure 23: Timeline of the first version of 3dfft_omp.c with 8 threads


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	95.10 %	-	4.75 %	0.12 %	0.02 %	0.00 %
THREAD 1.1.2	39.33 %	58.86 %	1.78 %	-	0.02 %	-
THREAD 1.1.3	39.17 %	58.86 %	1.95 %	-	0.02 %	-
THREAD 1.1.4	38.91 %	58.86 %	2.21 %	-	0.02 %	-
THREAD 1.1.5	36.92 %	58.86 %	4.20 %	-	0.02 %	-
THREAD 1.1.6	36.49 %	58.86 %	4.63 %	-	0.02 %	-
THREAD 1.1.7	40.06 %	58.87 %	1.05 %	-	0.02 %	-
THREAD 1.1.8	36.57 %	58.87 %	4.54 %	-	0.02 %	-
Total	362.56 %	412.05 %	25.12 %	0.12 %	0.14 %	0.00 %
Average	45.32 %	58.86 %	3.14 %	0.12 %	0.02 %	0.00 %
Maximum	95.10 %	58.87 %	4.75 %	0.12 %	0.02 %	0.00 %
Minimum	36.49 %	58.86 %	1.05 %	0.12 %	0.02 %	0.00 %
StDev	18.86 %	0.00 %	1.43 %	0 %	0.00 %	0 %
Avg/Max	0.48	1.00	0.66	1	0.89	1

Figure 24: State config with (% time) of the first version of 3dfft_omp.c with 8 threads

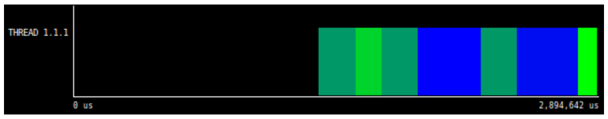
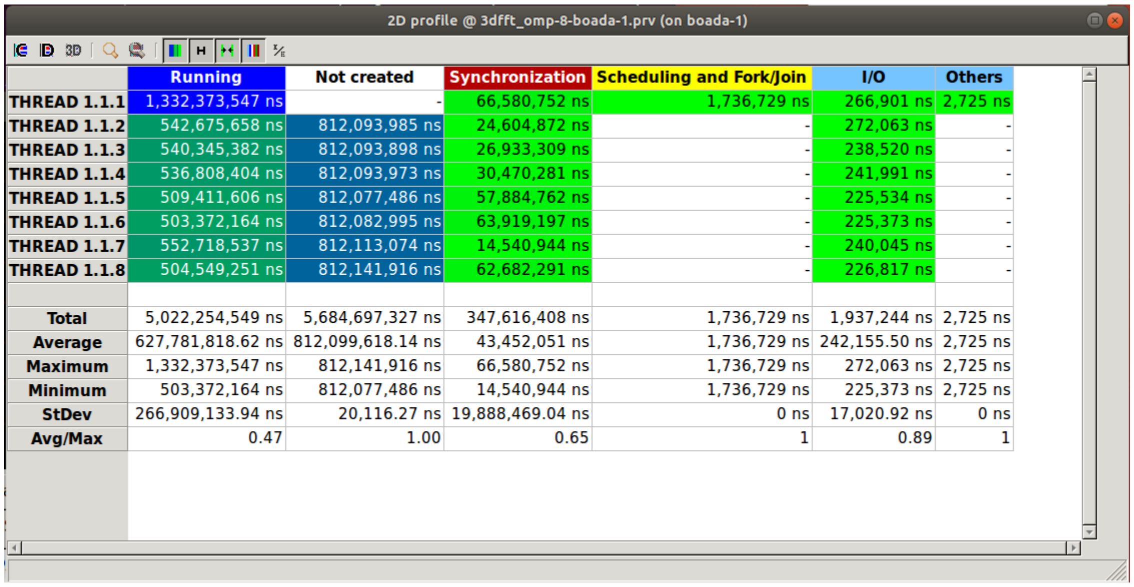


Figure 25: Parallel function duration config with of the first version of 3dfft_omp.c with 1 thread. Painted parts represent $Tpar$.


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	1,332,373,547 ns	-	66,580,752 ns	1,736,729 ns	266,901 ns	2,725 ns
THREAD 1.1.2	542,675,658 ns	812,093,985 ns	24,604,872 ns	-	272,063 ns	-
THREAD 1.1.3	540,345,382 ns	812,093,898 ns	26,933,309 ns	-	238,520 ns	-
THREAD 1.1.4	536,808,404 ns	812,093,973 ns	30,470,281 ns	-	241,991 ns	-
THREAD 1.1.5	509,411,606 ns	812,077,486 ns	57,884,762 ns	-	225,534 ns	-
THREAD 1.1.6	503,372,164 ns	812,082,995 ns	63,919,197 ns	-	225,373 ns	-
THREAD 1.1.7	552,718,537 ns	812,113,074 ns	14,540,944 ns	-	240,045 ns	-
THREAD 1.1.8	504,549,251 ns	812,141,916 ns	62,682,291 ns	-	226,817 ns	-
Total	5,022,254,549 ns	5,684,697,327 ns	347,616,408 ns	1,736,729 ns	1,937,244 ns	2,725 ns
Average	627,781,818.62 ns	812,099,618.14 ns	43,452,051 ns	1,736,729 ns	242,155.50 ns	2,725 ns
Maximum	1,332,373,547 ns	812,141,916 ns	66,580,752 ns	1,736,729 ns	272,063 ns	2,725 ns
Minimum	503,372,164 ns	812,077,486 ns	14,540,944 ns	1,736,729 ns	225,373 ns	2,725 ns
StDev	266,909,133.94 ns	20,116.27 ns	19,888,469.04 ns	0 ns	17,020.92 ns	0 ns
Avg/Max	0.47	1.00	0.65	1	0.89	1

Figure 26: State config with (time) of the first version of 3dfft_omp.c with 8 threads

The results obtained are in the following table.
3dfft:
Tpar = 1523 ms
Tseq = 1372 ms

3dfft improving parallel fraction:
Tpar = 2087 ms
Tseq = 361 ms

3dfft reducing overhead:
Tpar = 2258 ms
Tseq = 500 ms

Version	ϕ	S_∞	T_1	T_8	S_8
initial version in *3dfft_omp.c*	0.53	2.12	2895 ms	1400 ms	2.07
new version with improved ϕ	0.85	6.67	2448 ms	960 ms	2.55
final version with reduced parallelisation overhead	0.82	5.56	2758 ms	1015 ms	2.72

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