PAR Laboratory Assignment

Lab 1: Experimental setup and tools

06/10/2021

Fall 2021-22

Index

0. Introduction

The scope of the first session of Parallelism laboratory is to introduce ourselves to the boada architecture, a multiprocessor server located at the Computer Architecture Department, composed of 9 nodes, named boada-1 to boada-9, equipped with different processor generations. In this course we are going to use only boada-1 to boada-4 nodes.

1. Experimental setup

In this section we are going to describe the architecture of boada server.

1.1. Node architecture and memory

In order to obtain information about the hardware of boada-1 (identical to the nodes of boada-2 to boada-4), we proceed to execute two commands:

- `lscpu`: this command gives us information about the CPU in text format, listing all details.
- `lstopo –of fig map.fig`: with this command and this specific options about this gives us a `.fig` file (a drawing of the architecture) that with `xfig` command we can visualise it.

The results obtained show us that:

	`boada-1 to boada-4`
Number of sockets per node	2 sockets/node
Number of cores per socket	6 cores/socket
Number of threads per core	2 threads/core
Maximum core frequency	2.395 GHz
L1-I cache size (per-core)	32 KB
L1-D cache size (per-core)	32 KB
L2 cache size (per-core)	256 KB
Last-level cache size (per-socket)	12 MB
Main memory size (per socket)	12 GB
Main memory size (per node)	24 GB

The next image shows the representation of boada-1 architecture. We can see all the features showed in the table in a more intuitive and descriptive diagram.

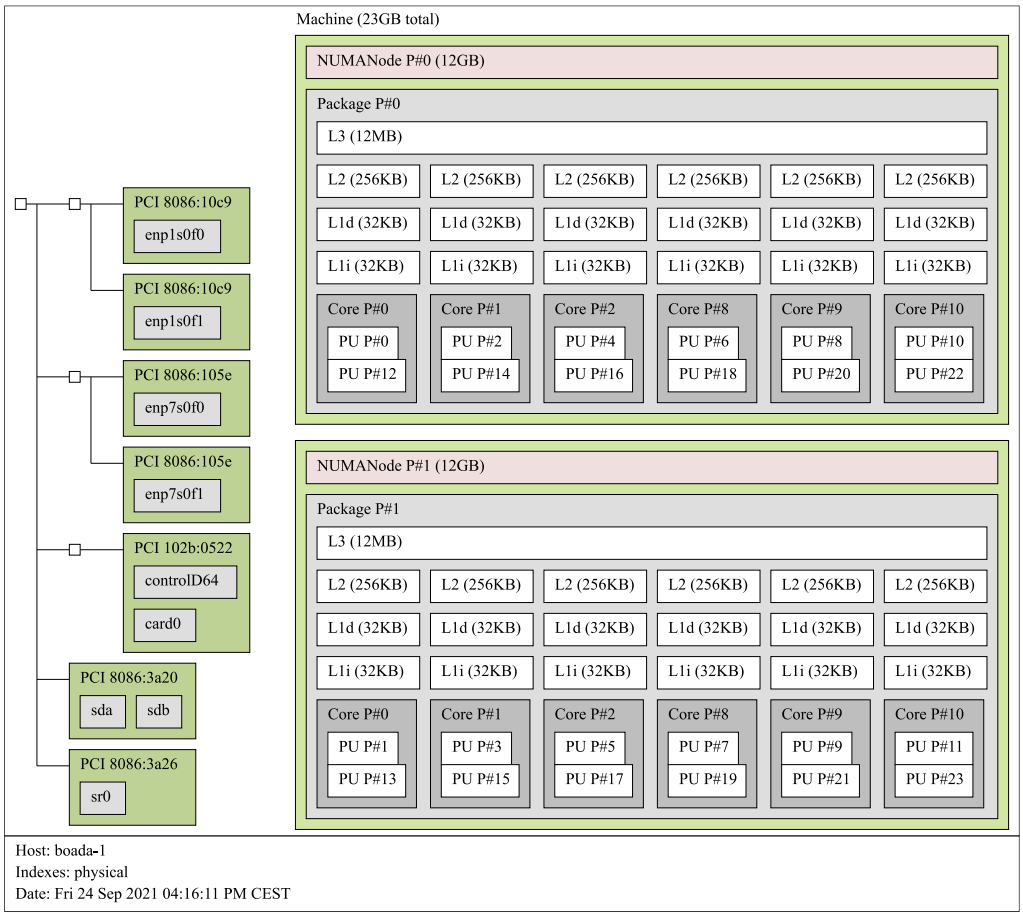

Figure 1.1: Architectural diagram

1.2. Strong vs. weak scalability

We have to ways to execute our programs in boada:

- Queued: the job will be executed in isolation and will use several processors in a node, because of that we will obtain reliable performance results.
- Interactive: execution will start immediately but will share resources with other programs and interactive jobs, consequently, will obtain worst performance than the previous option.

To see the differences between these two options we executed “pi_omp.c” in different scenarios:

# Threads	Interactive				Queued			
	user	system	elapsed	% of CPU	user	system	elapsed	% of CPU
1	3.94	0.0	0:03.94	99%	3.93	0.01	0:03.97	99%
2	7.97	0.0	0:03.99	199%	3.95	0.00	0:01.99	198%
4	7.94	0.08	0:04.02	199%	3.99	0.00	0:1.02	388%
8	7.96	0.05	0:04.01	199%	4.23	0.00	0:00.54	771%

We notice that the elapsed time in queued executions improves with the number of threads while it hangs around the same for interactive executions. As well we observed that the percentage of the CPU it remains the same for the interactive executions while for the queued ones are multiplied approximately by two, equal to the number of threads used.

The scalability is a measure of the capacity to effectively utilize an increasing number of processors in parallel. We have two types of scenarios considered:

- The *strong* scalability: the number of threads is changed with a fixed problem size. The parallelism reduces the execution time of the program.
- The *weak* scalability: the problem size is proportional to the number of threads. The problem size increases at the same time as the number of processors involved.

To analyse these scenarios, we used to scripts: “submit-strong-omp.sh” and “submit-weak-omp.sh”. These scripts allowed us to execute the parallel code using a settled number of threads.

Firstly, we used the strong scalability script with two different maximum threads. The results obtained were:

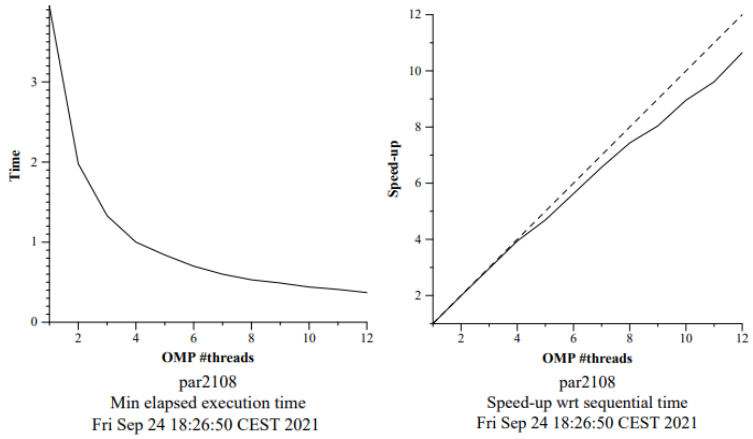

Figure 1.2.1: Strong scalability with 12 threads

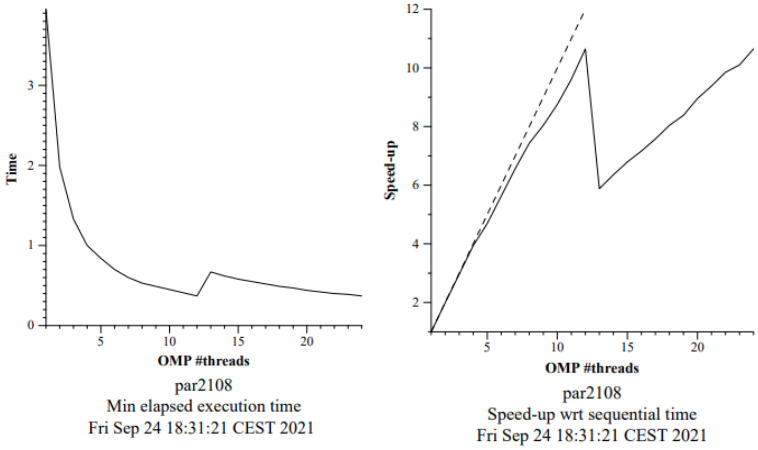

Figure 1.2.2: Strong scalability with 24 threads

In figure 1.2.2 we can see that the speed-up decreases when arrive at 11-12 threads, that's because the code can not be split in more tasks to improve the efficiency. Even if we could use an infinite number of threads, we would not obtain a better time execution.

To see more exactly what would happen if we increased the number of threads used, we change the NP_MAX to 64 instead of 24. And we obtained:

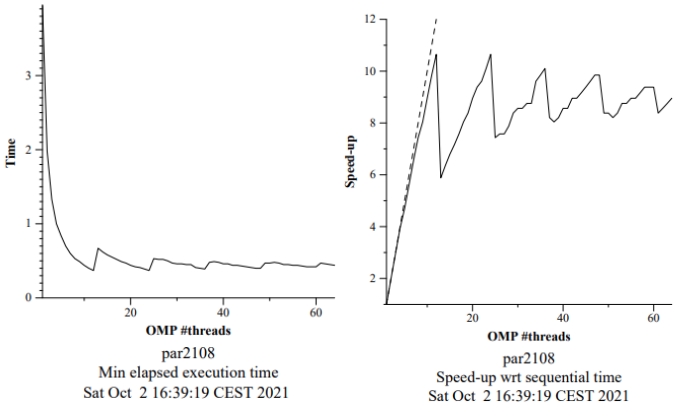

Figure 1.2.3: Strong scalability with 64 threads

We can conclude that increasing the number of threads will obtain a worst execution time and the speed up. Because of this, is so important this kind of analysis, to know how much we can improve our code using parallelism and not waste hardware performance on something that will get worse software performance.

And the result when we use weak scalability was:

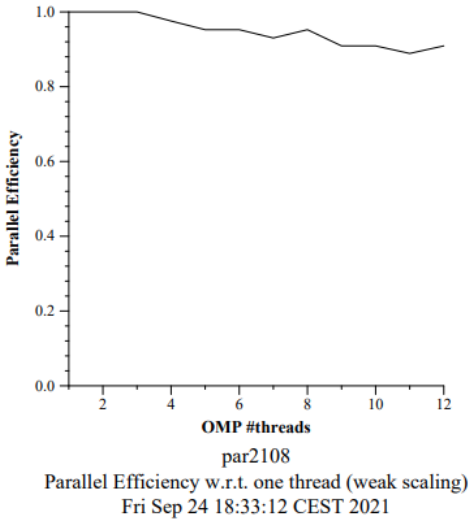

Figure 1.1.4: Weak scalability

As expected, the weak scalability remains at the same level of parallelism efficiency because the problem size adapts to the number of threads disponible.

2. Systematically analysing task decompositions with Tareador

The goal for this second session is to familiarize ourselves with the tool *Tareador* that:

1- Traces the execution of the program based on the specification of potential tasks to be run in parallel.
2- Records all static/dynamic data allocations and memory accesses in order to build the task dependence graph
3- Simulates the parallel execution of the tasks on certain number of processors in order to estimate the potential speed-up.

The only think that we need to do is to identify the tasks that we want to be evaluated.

2.1. Analysis of task decompositions for 3DFFT

For the purpose of discover more parallelism we will incrementally generates five new finer-grained task decompositions as of "3dfft_tar.c". For each new version will compute T_1, T_∞ and the potential parallelism (T_1/T_∞) from the task dependence graph generated by Tareador.

Version 1

For this version we replaced the task named "ffts1_and_transpositions" with a sequence of finer-grained tasks, one for each function invocation inside it.

The modification that we made in the code was:

```
...
START_COUNT_TIME;

tareador_start_task("ffts1_and_transpositions");
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);
tareador_end_task("ffts1_and_transpositions");

STOP_COUNT_TIME("Execution FFT3D");

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

→

```
...
START_COUNT_TIME;

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

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

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

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

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

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

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

```
tareador_end_task("transpose_xy_planes");

STOP_COUNT_TIME("Execution FFT3D");

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

What we have change in the code is that for each function that the main code calls, it generates a task.

The result graph dependency and the simulations with different processors:

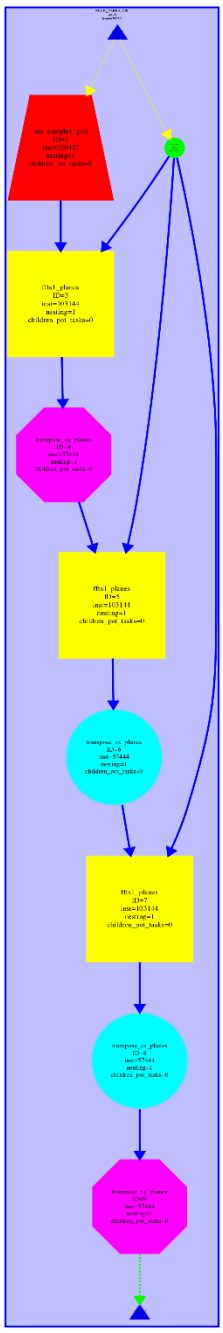

Figure 2.1: Dependence graph for 3dfft_tar_v1.c

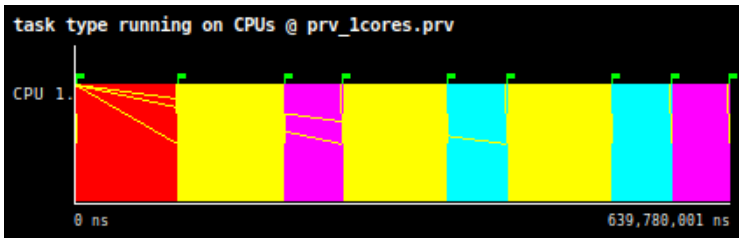

Figure 2.2: Simulation for 3dfft_tar_v1 with 1 processor

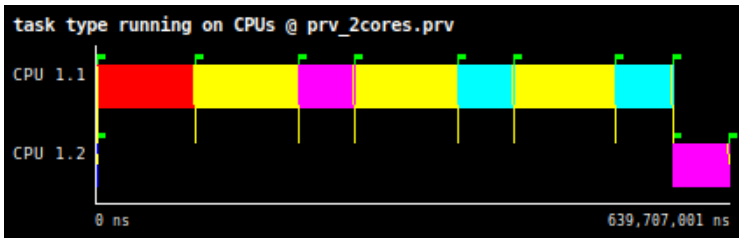

Figure 2.3: Simulation for 3dfft_tar_v1 with 2 processors

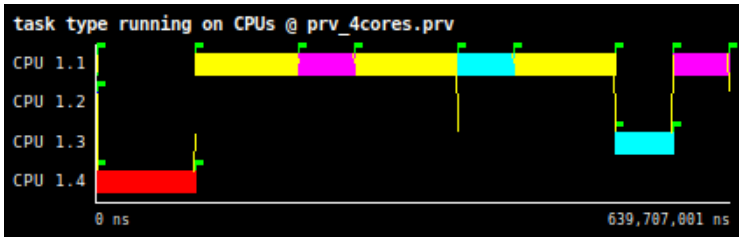

Figure 2.4: Simulation for 3dfft_tar_v1 with 4 processors

As of the different simulations we deduce that we can not improve the execution time from two processors in parallel. Because that $T_\infty = 639{,}707{,}001\ ns$ and we know that (because of the sequential time, time executed with 1 processor) $T_1 = 639{,}780{,}001\ ns$.

Version 2

Starting from the previous version we replace the “ffts1_planes” with a fine-grained task inside the function body and associated to individual iterations of the k loop.

The modification that we made in the code was:

```
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++)
       fftwf_execute_dft( p1d, (fftwf_complex *)in_fftw[k][j][0],
            (fftwf_complex *)in_fftw[k][j][0]);
    }
}

int main (int argc, char *argv[]) {
    ...

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

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

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

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

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

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

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

    STOP_COUNT_TIME("Execution FFT3D");
    ...
}
```

↓

```
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 (int argc, char *argv[]) {
    ...
    START_COUNT_TIME;

    ffts1_planes(p1d, in_fftw);

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

    ffts1_planes(p1d, tmp_fftw);

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

    ffts1_planes(p1d, in_fftw);

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

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

    STOP_COUNT_TIME("Execution FFT3D");
}
```

The change made will generate a task every iteration in the k loop when we are in the function “ffts1_planes”, not only when we call this function.

The result graph dependency and the simulations with different processors:

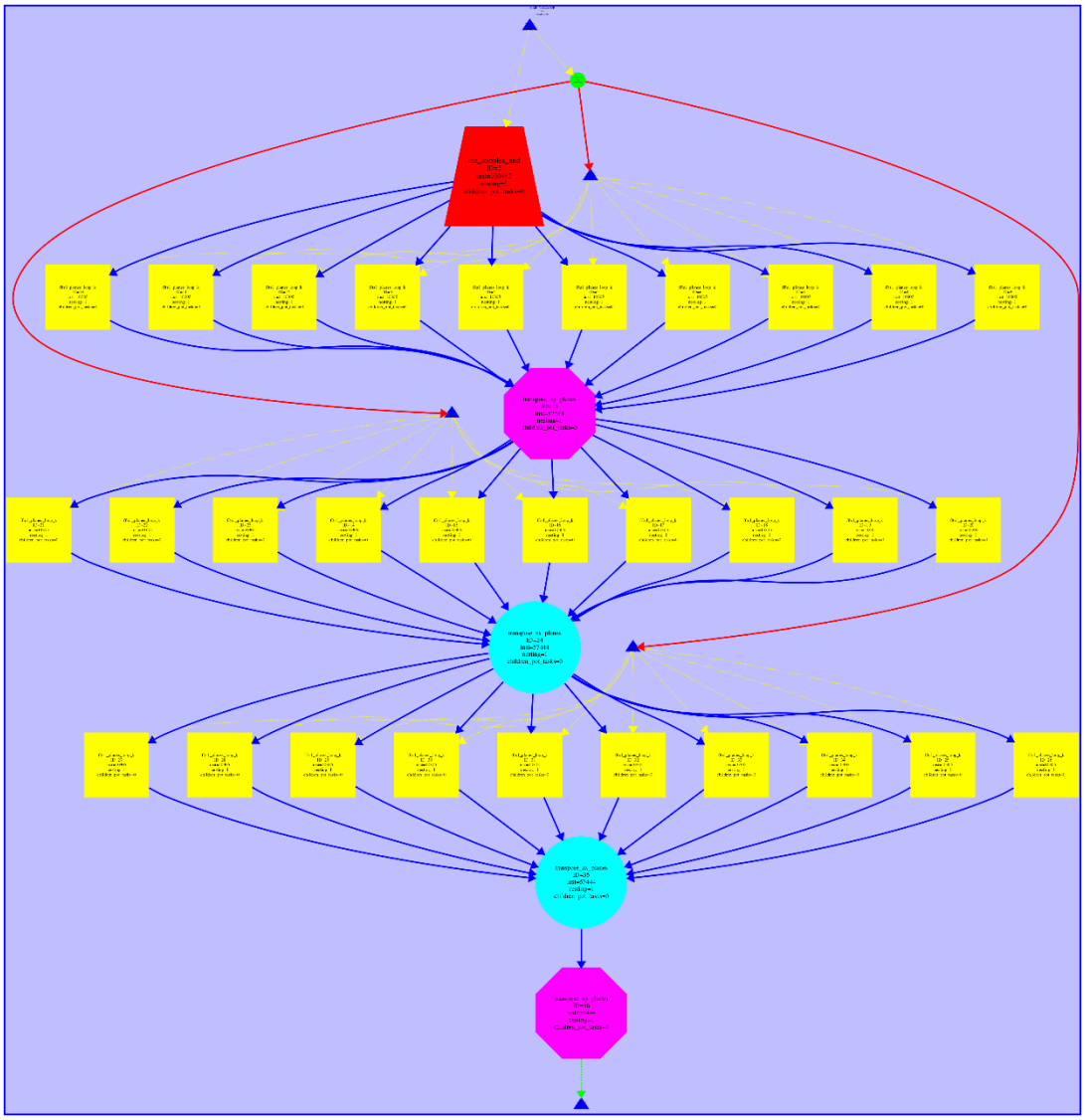

Figure 2.5: Dependence graph for 3dfft_tar_v2.c

Figure 2.6: Simulation for 3dfft_tar_v2 with 1 processor

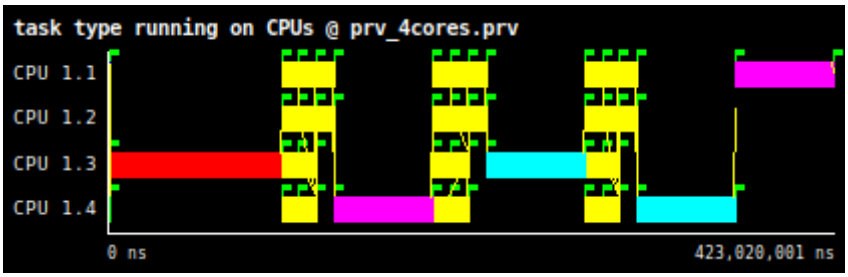

Figure 2.7: Simulation for 3dfft_tar_v2 with 4 processors

From the figure 2.6 we obtain that $T_1 = 639{,}780{,}001\ ns$ and as the previous version, what we did to obtain T_∞ is run different simulations by increasing the number of processors until the runtime stopped changing, and we obtain that $T_\infty = 361{,}190{,}001\ ns$.

Version 3

Starting from version 2, we replaced the function invocations: “transpose_xy_planes” and “transpose_zx_planes” with fine-grained tasks inside the first loop, as we did previously with “ffts1_planes”.

The modification that we made in the code was:

```
...
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++) {
     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];
       }
     }
    }
}

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++) {
     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];
       }
     }
    }
}
...

int main (int argc, char *argv[]) {
    ...
    START_COUNT_TIME;

    ffts1_planes(p1d, in_fftw);

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

    ffts1_planes(p1d, tmp_fftw);

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

```
    ffts1_planes(p1d, in_fftw);

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

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

    STOP_COUNT_TIME("Execution FFT3D");
    ...
}
```

↓

```
...
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 (int argc, char *argv[]) {
    ...
    START_COUNT_TIME;
```

```
    ffts1_planes(p1d, in_fftw);

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

    ffts1_planes(p1d, tmp_fftw);

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

    ffts1_planes(p1d, in_fftw);

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

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

    STOP_COUNT_TIME("Execution FFT3D");
    ...
}
```

In this case what we did is to change the call to both functions transpose to creates a task every iteration inside the first loop for each transpose function.

The result graph dependency and the simulations with different processors:

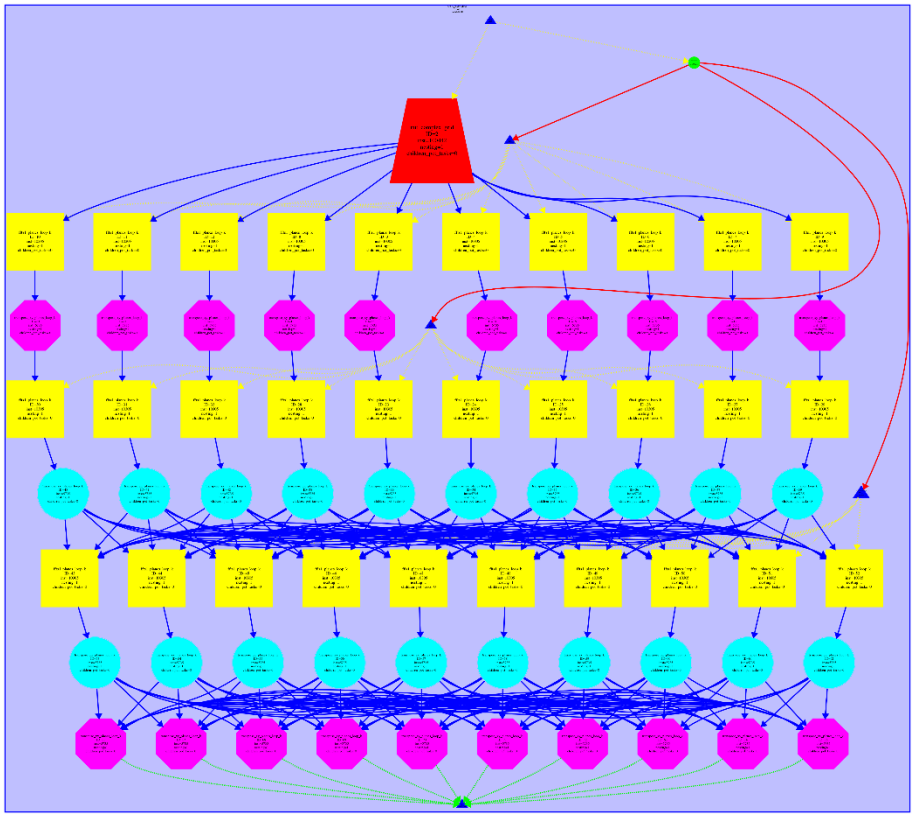

Figure 2.8: Dependence graph for 3dfft_tar_v3.c

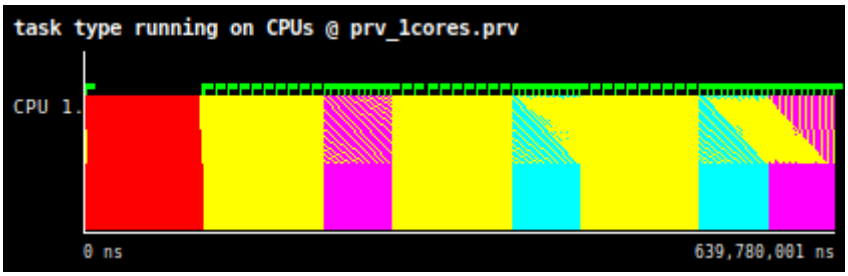

Figure 2.9: Simulation for 3dfft_tar_v3 with 1 processor

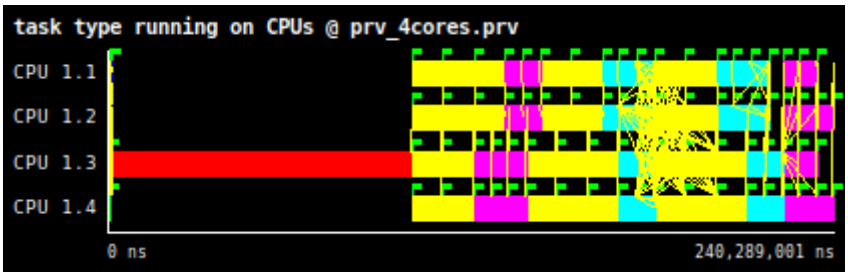

Figure 2.10: Simulation for 3dfft_tar_v3 with 4 processors

As the precursory version, we obtain that $T_1 = 639{,}780{,}001\,ns$ from the sequential execution (Figure 2.9) and running different simulations we obtain that $T_\infty = 154{,}354{,}001\,ns$

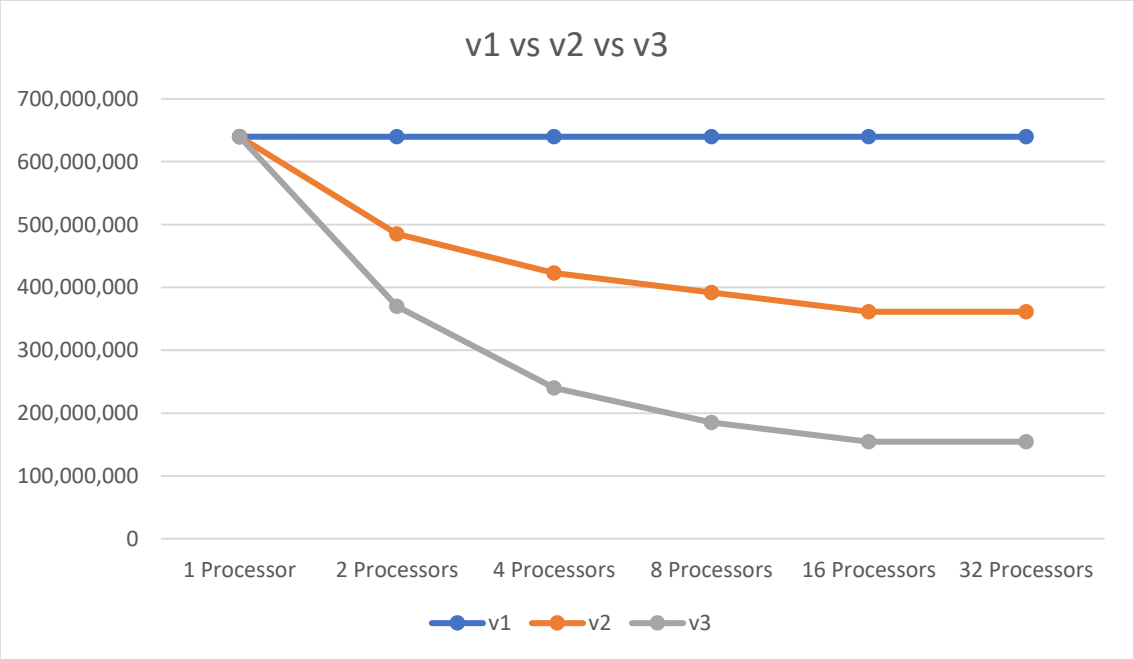

Figure 2.11: Graph comparing different versions with different number of processors running in parallel

Version 4

Starting from version 3, we replaced the definition for the “init_complex_grid” function with fine-grained tasks inside the body function.

The modification that we made in the code was:

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

  for (k = 0; k < N; 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
      }
    }
  }
}
...
```

↓

```
...
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");
    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");
  }
}
...
```

As we did with the prior version, we add a call inside the first loop of the function.

The result graph dependency and the simulations with different processors:

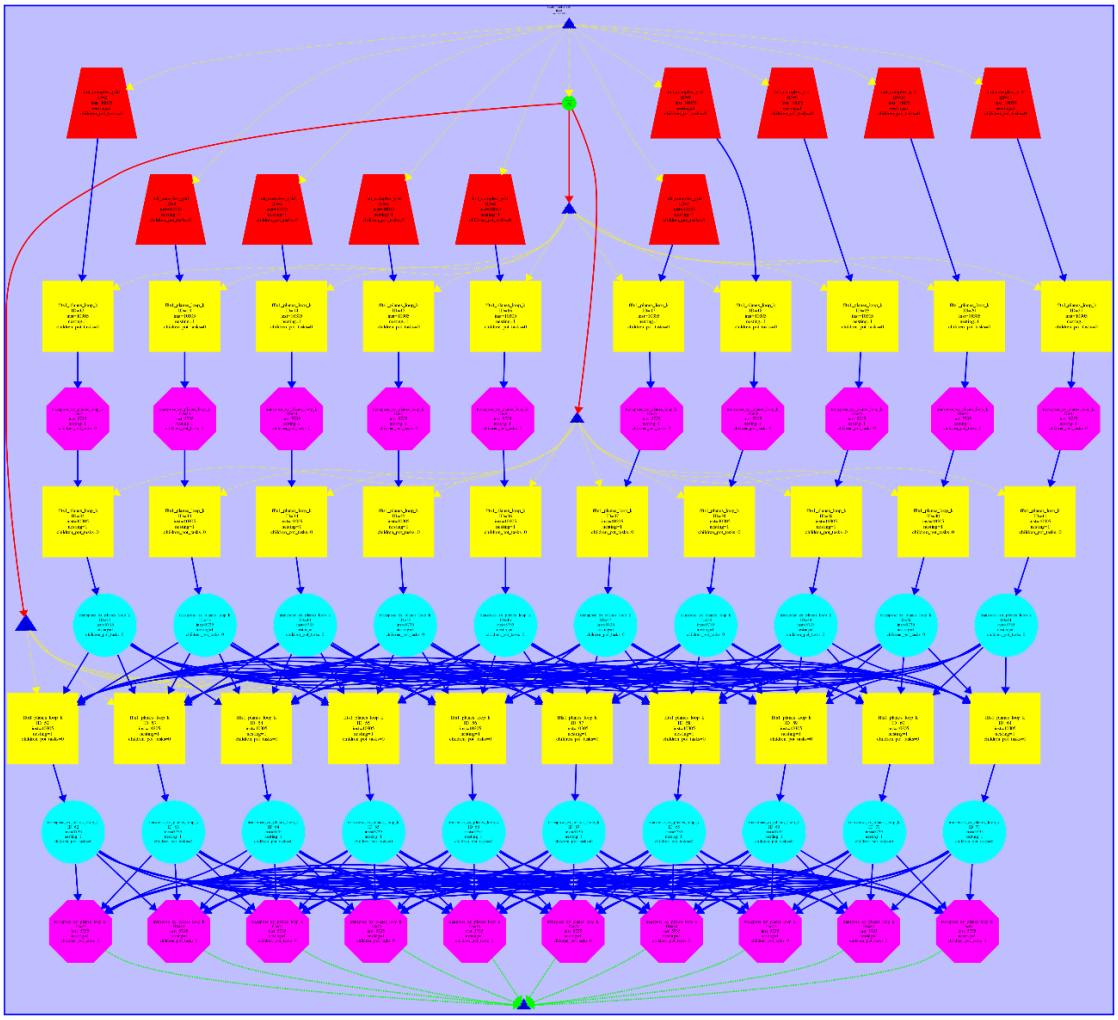

Figure 2.12: Dependence graph for 3dfft_tar_v4.c

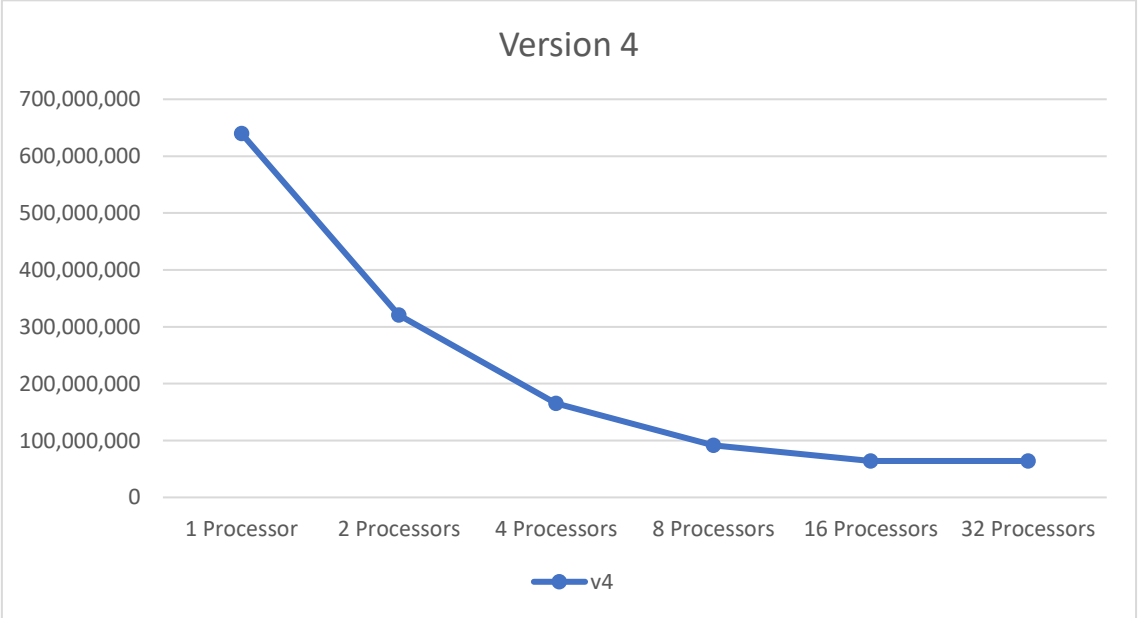

Figure 2.13: Graph comparing the 4th version with different number of processors running in parallel

Running few simulations, provides us with $T_1 = 639{,}780{,}001\ ns$ and $T_\infty = 64{,}018{,}001\ ns$

What limits the scalability of this 4th version is that no matter how many processors we assign to it, due of dependencies, it will not use all the resources for parallelization.

Version 5

For this final version we explored even a finer-grained tasks, to do that we changed the call task from k loop to j loop. The code that we changed:

```
...
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");
    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");
  }
}
...
```

↓

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

  for (k = 0; k < N; 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");
    }
  }
}
...
```

The result graph dependency and the simulations with different processors:

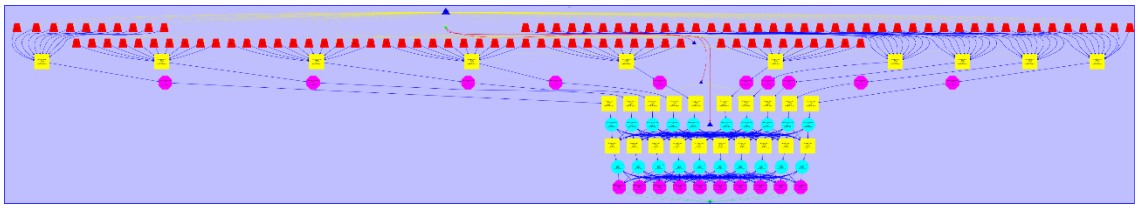

Figure 2.14: Dependence graph for 3dfft_tar_v5.c

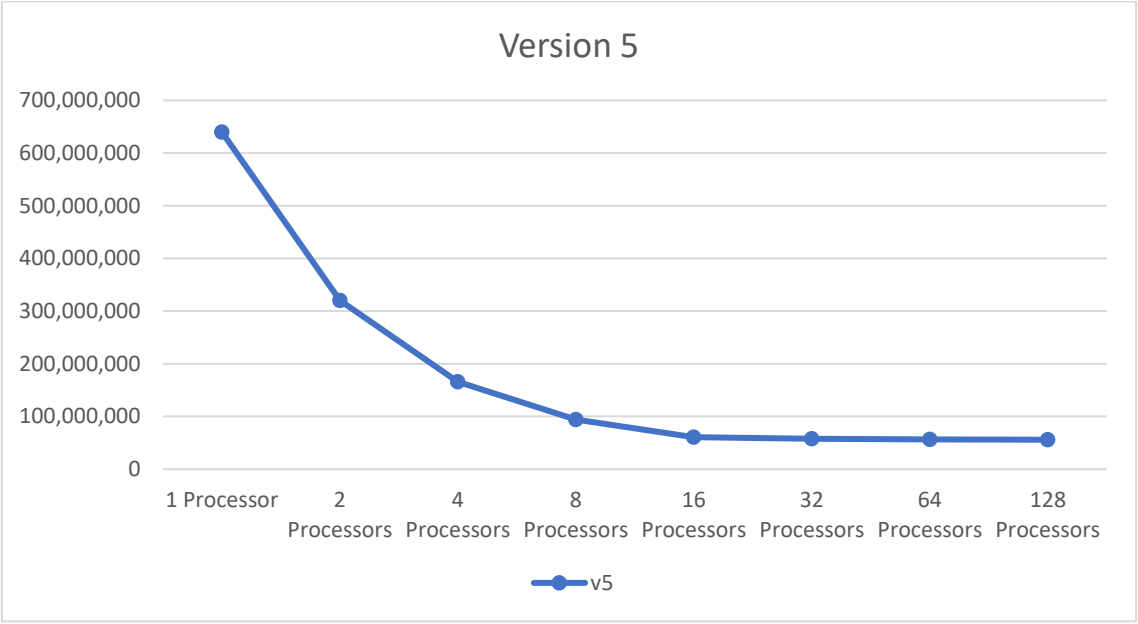

Figure 2.15: Graph comparing the 5th version with different number of processors running in parallel

Running few simulations, provides us with $T_1 = 639{,}780{,}001\ ns$ and for T_∞ we could not obtain an exact number because we cannot simulate more than 128 processors, but as the last ones vary very little, we can estimate that the last simulation (with 128 processors) is sufficiently accurate, then $T_\infty = 55{,}820{,}001$ ns.

Version 4 vs Version 5

If we compare the last two versions, we can see that the 5th version is a bit better than the 4th version, but the resources expended do not compensate for the improvement. We can notice in the figure 2.16 that if we double the number of processors to 64, the improvement is very low, even with 128 processors running in parallel remains low.

To conclude we could say that it is worth going to this level of granularity when the results obtained are in agreement with the amount of resources spent in it, in other words, the improvement is noticeable. Something to remark is that this type of programs helps us to decide when and how is better to do this type of improvements.

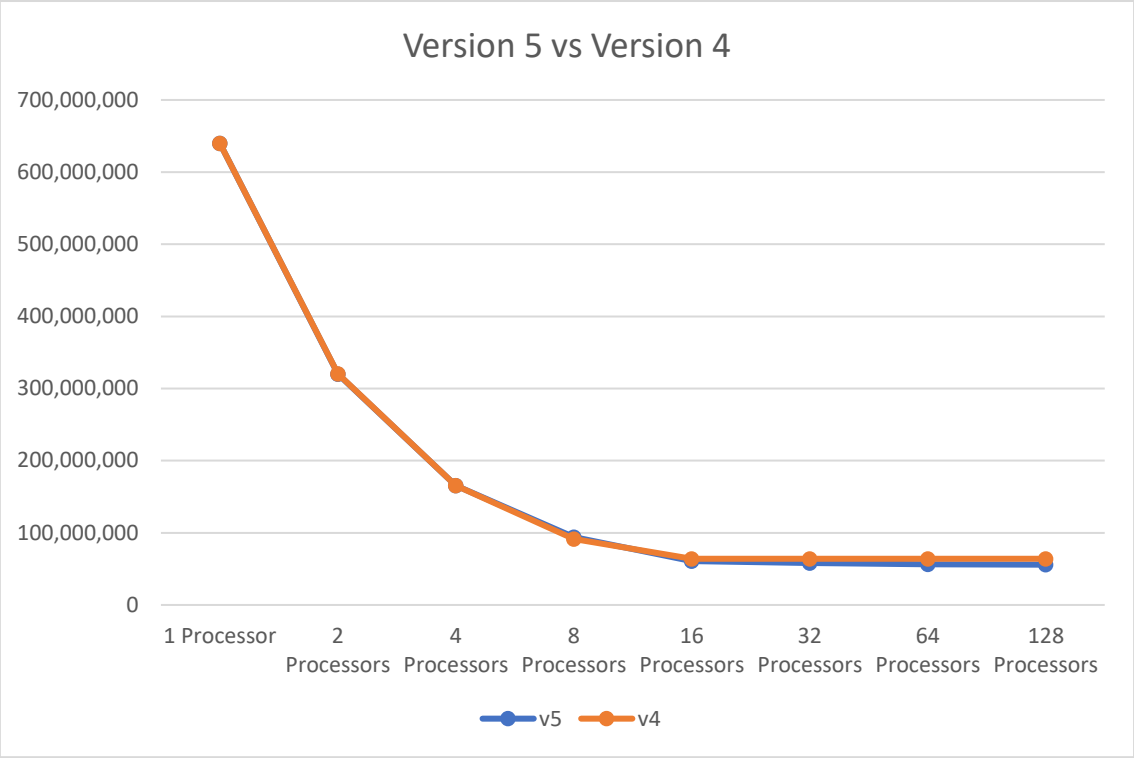

Figure 2.16: Graph comparing the versions 4 and 5 with different number of processors running in parallel

In the following table we summarize the evolution metrics:

Version	T_1	T_∞	*Parallelism*
Sequential version	$639{,}780{,}001\ ns$	$639{,}780{,}001\ ns$	1
Version 1	$639{,}780{,}001\ ns$	$639{,}707{,}001\ ns$	1.000114115
Version 2	$639{,}780{,}001\ ns$	$361{,}190{,}001\ ns$	1.771311496
Version 3	$639{,}780{,}001\ ns$	$154{,}354{,}001\ ns$	4.144887705
Version 4	$639{,}780{,}001\ ns$	$64{,}018{,}001\ ns$	9.993751617
Version 5	$639{,}780{,}001\ ns$	$55{,}820{,}001$ ns	11.461483152

We notice the first row $T_1 = T_\infty$, that's because is a sequential version, we do not parallelize any task. The version 1 almost not change anything when we parallelize tasks because the changes made to the code are not noticeable. In the second version we start to see a real improvement, almost doubling the parallelism, this is because we generate tasks in each iteration inside the first loop of the function "ffts1_planes". The 3rd version gets an even greater improvement than the prior version, here we do the same as the older version but instead of one function in two. The 4th and 5th versions are similar, and it doesn't make a big improvement from one to another, but are an improvement on their predecessors.

3. Understanding the execution of OpenMP programs

The objective of this session is to familiarize with the *Paraver* environment, which is composed of *Extrae* and *Paravear*:

-*Extrae* will help us collecting information about the status of each thread and different events related with the execution of the parallel program

-Then, the *Paraver* trace browser will be used to visualize the trace and analyze the execution of the program.

3.1. Initial version

Once we have the output from the execution of the program (using 1 thread), with the help of *Paravear* we obtain the following information:

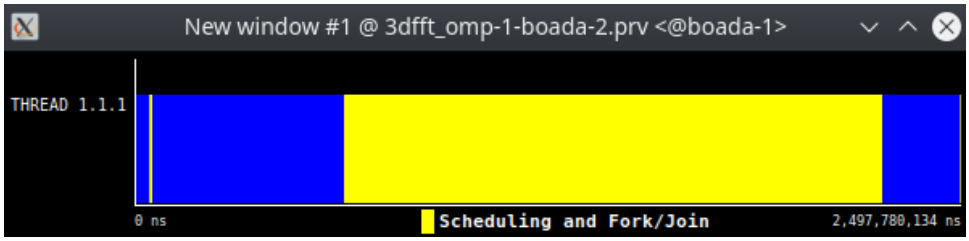

Figure 3.1: simulation for 3dfft_omp with 1 thread v1

Program execution time is approximately 2’5 seconds. Using a OMP parallel constructs configuration file, we can obtain the time spent by the program in the parallelized parts.

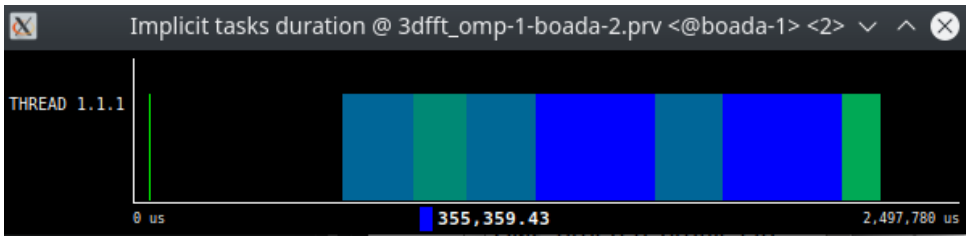

Figure 3.2: simulation with OMP parallel constructs configuration v1

The sum of the parallelizable parts gives a total of 1626.194 ms. With these data, we have T1=2497.780 ms and Tpar=1626.194 ms, and we can obtain Tseq with the following formula:

T1 = Tseq+Tpar => Tseq = T1-Tpar = 2497.780-1626.194 = 871.586 ms

and now we can calculate the parallel fraction: φ = Tpar ÷ (Tseq + Tpar)

φ = 1626.194 / 2497.780 = 0.65

For the execution of the program with 8 threads, we obtain the following information:

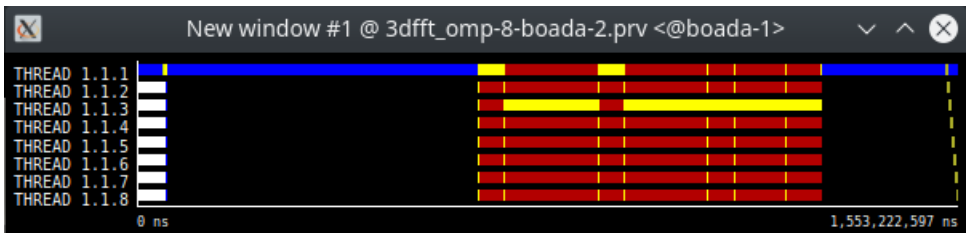

Figure 3.3: simulation for 3dfft_omp v1 with 8 threads

T8 = 1553.223 ms

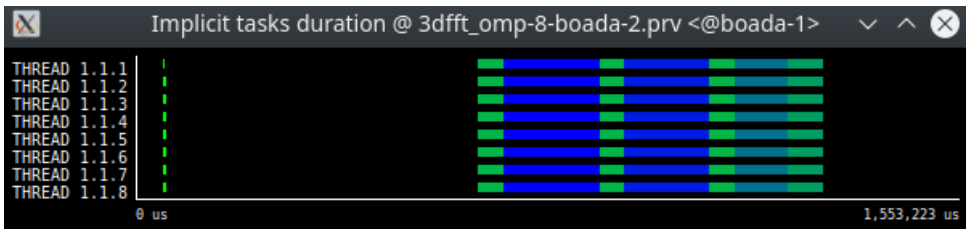

Figure 3.4: simulation with implicit tasks duration configuration v1

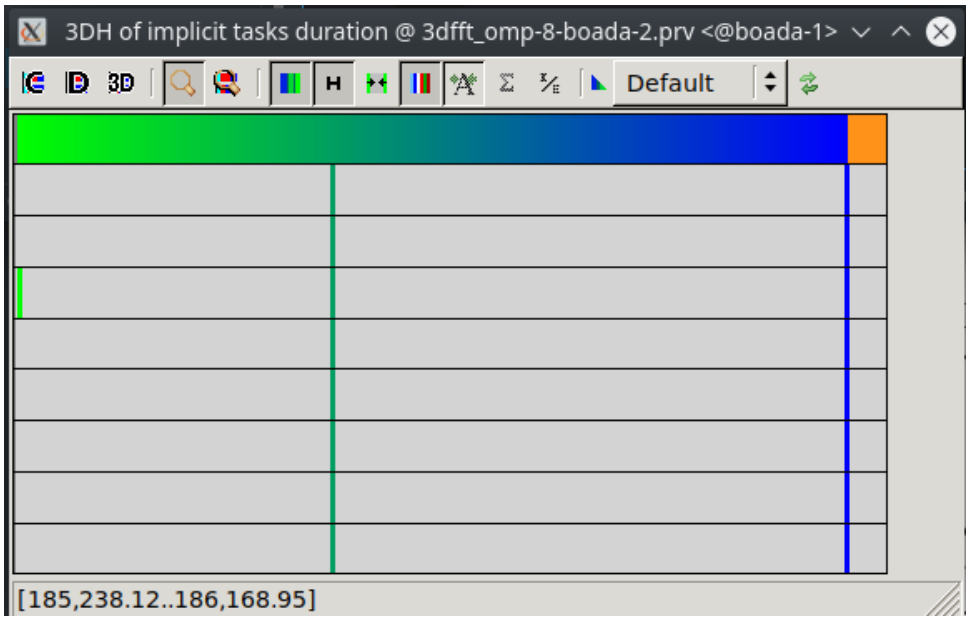

Figure 3.5: histograms of implicit tasks duration v1

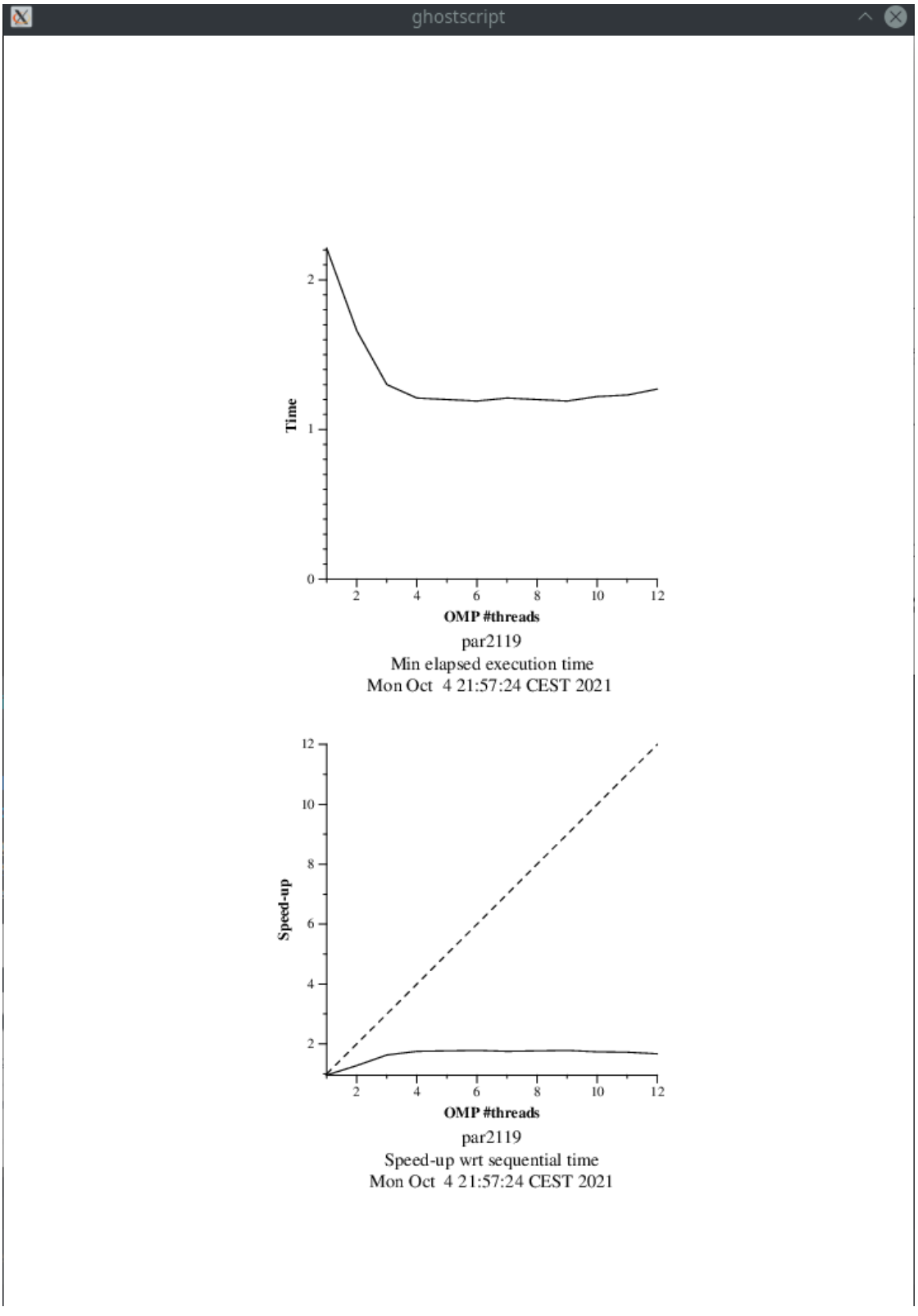

Figure 3.6: strong scalability plot v1

As in the initial version init_complex_grid is not parallelized, the efficiency of the program is very low.

3.2. Improving φ

In this version we allow the parallel execution of function init_complex_grid increasing the parallelizable part of the code and obtaining an improvement in φ.

After uncommenting the lines of code needed and executing the program with 1 and 8 threads, we obtain the following results:

T1 = 2777.87 ms

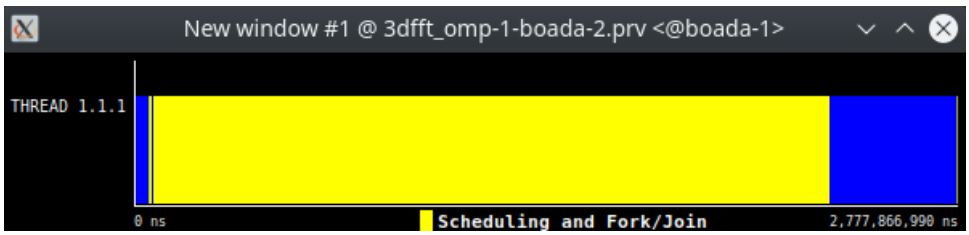

Figure 3.7: simulation for 3dfft_omp v2 with 1 thread

Figure 3.8: simulation with implicit tasks duration configuration v2

Tpar = 2279.31 ms

φ = 2279.31 / 2777.87 = 0.82

T8 = 1013.95 ms

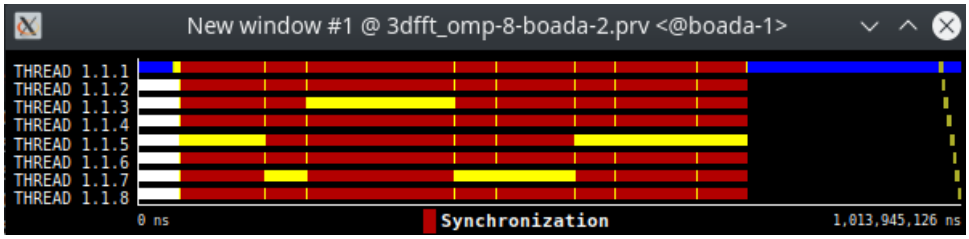

Figure 3.9: simulation for 3dfft_omp v2 with 8 threads

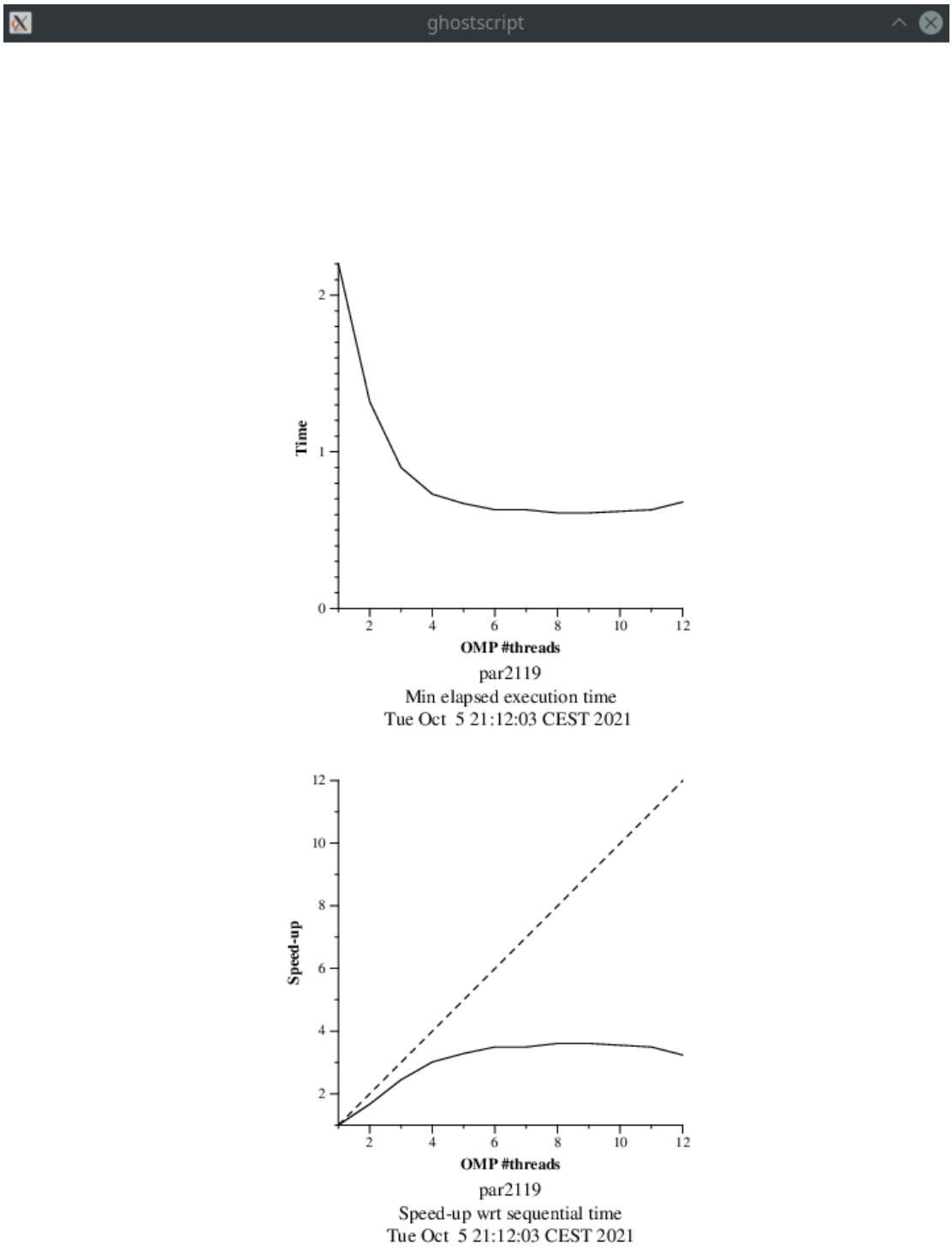

Figure 3.10: strong scalability plot v2

We can observe an improvement in the performance of the program compared to the first version, but due to overheads, the performance improvement stops at a point.

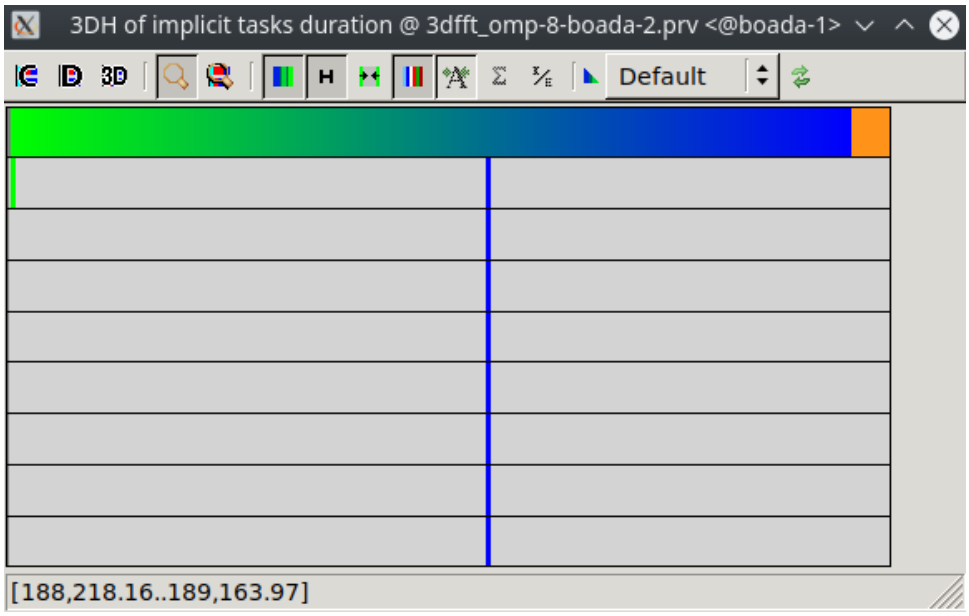

Figure 3.11: histograms of implicit tasks duration v2

2D thread state profile @ 3dfft_omp-8-boada-2.prv <@boada-1>

Default

	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	86.43 %	-	11.04 %	2.23 %	0.30 %	0.00 %
THREAD 1.1.2	76.24 %	6.49 %	16.87 %	0.00 %	0.40 %	-
THREAD 1.1.3	70.81 %	6.48 %	14.06 %	8.09 %	0.54 %	-
THREAD 1.1.4	76.00 %	6.49 %	17.05 %	0.00 %	0.45 %	-
THREAD 1.1.5	76.69 %	6.49 %	16.39 %	0.00 %	0.42 %	-
THREAD 1.1.6	76.20 %	6.50 %	16.86 %	0.00 %	0.45 %	-
THREAD 1.1.7	76.64 %	6.49 %	16.44 %	0.00 %	0.42 %	-
THREAD 1.1.8	76.16 %	6.52 %	16.92 %	0.00 %	0.39 %	-
Total	615.17 %	45.46 %	125.64 %	10.34 %	3.39 %	0.00 %
Average	76.90 %	6.49 %	15.71 %	1.29 %	0.42 %	0.00 %
Maximum	86.43 %	6.52 %	17.05 %	8.09 %	0.54 %	0.00 %
Minimum	70.81 %	6.48 %	11.04 %	0.00 %	0.30 %	0.00 %
StDev	4.04 %	0.01 %	1.98 %	2.67 %	0.06 %	0 %
Avg/Max	0.89	1.00	0.92	0.16	0.78	1

Figure 3.12: thread state profile v2

3.3. Reducing parallelisation overheads

In this latest version of the program we are going to parallelize outside the k loop of all functions, which will reduce overheads and improve the execution time of the program.

T1 = 2440.75 ms

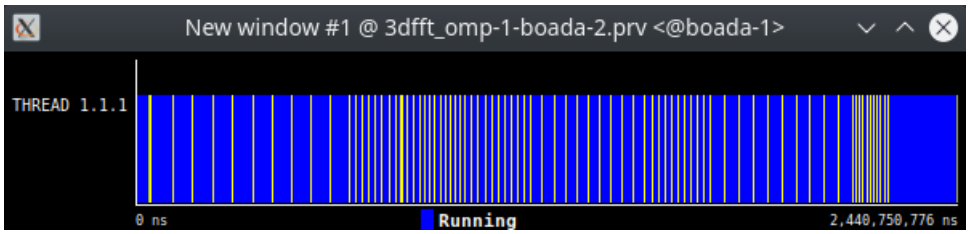

Figure 3.13: simulation for 3dfft_omp v3 with 1 thread

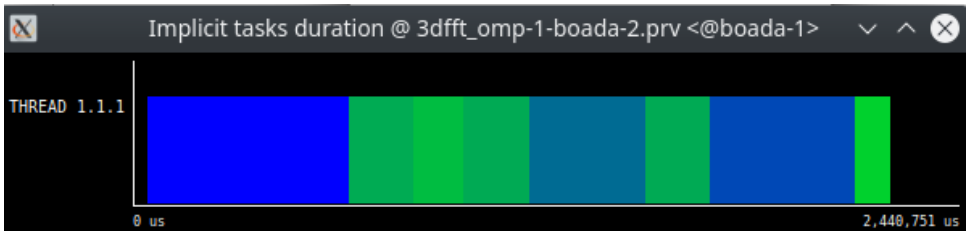

Figure 3.14: simulation with implicit tasks duration configuration v3

Tpar = 2165.27 ms

φ = 2165.27 / 2440.75 = 0.89

T8 = 641.04 ms

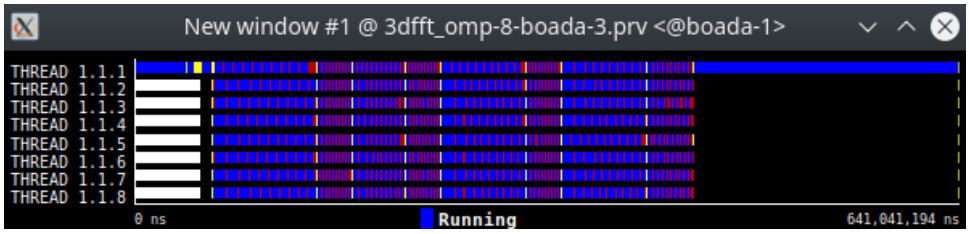

Figure 3.15: simulation for 3dfft_omp v3 with 8 threads

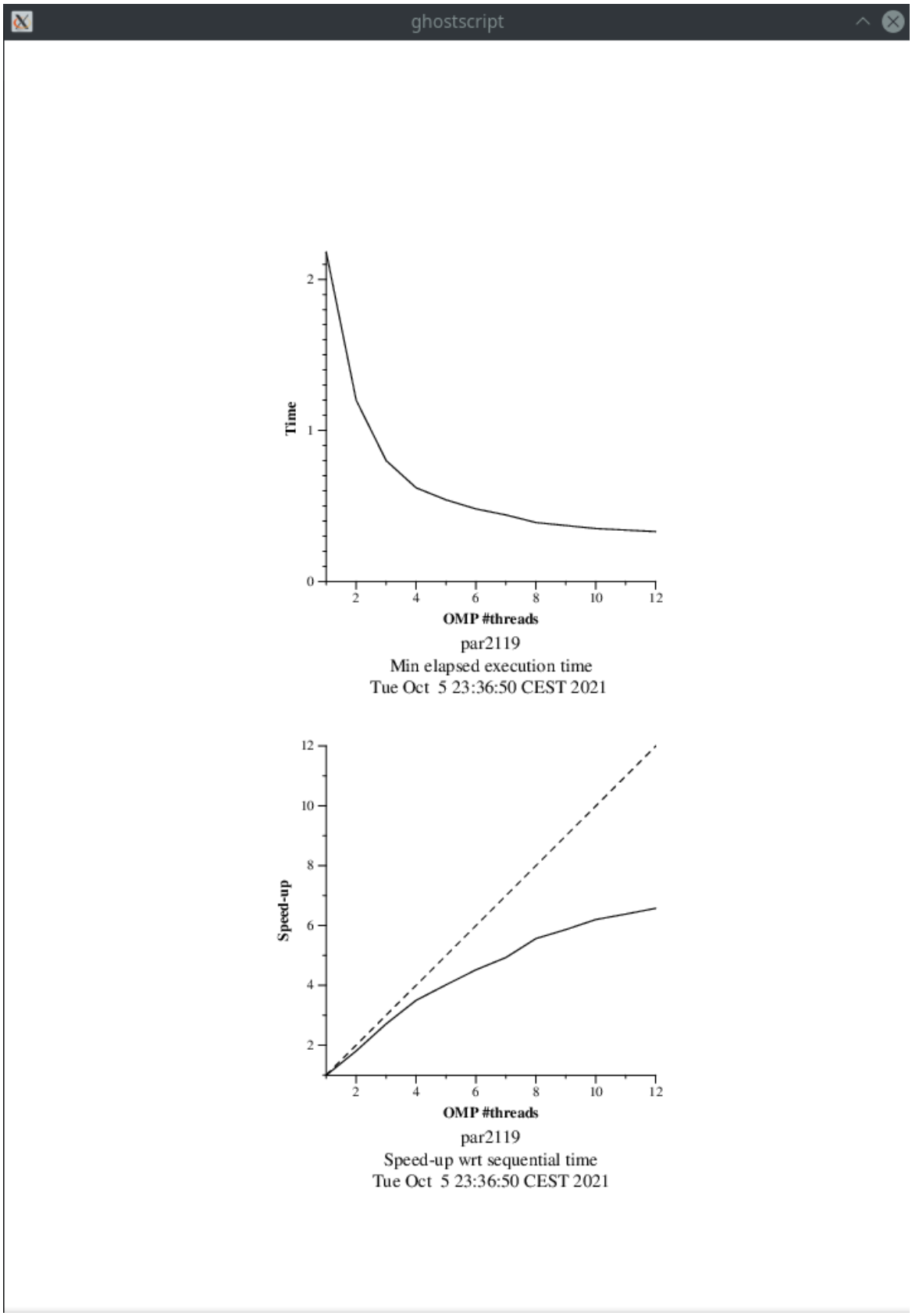

Figure 3.16: strong scalability plot v3

Analyzing the graphs we can see how in this latest version the deviation from the ideal case is slower and less than in previous cases, where the deviation moves away faster and becomes almost constant.

Now we only need to obtain the speed-up with the information that we have obtained throughout the session:

ideal S8 = 1 / (1- φ) and real S8 = T1 / T8

V1: ideal S8 = 1 / (1- 0.65) = 2.86, S8 = 2497.78 / 1553.22 = 1.61

V2: ideal S8 = 1 / (1- 0.82) = 5.55, S8 = 2777.87 / 1013.95 = 2.74

V3: ideal S8 = 1 / (1- 0.89) = 9.09, S8 = 2440.75 / 641.04 = 3.81

Version	φ	ideal S8	T1	T8	real S8
initial version in 3dfft omp.c	0.65	2.86	2.5s	1.5s	1.61
new version with improved φ	0.82	5.55	2.8s	1.0s	2.74
final version with reduced parallelisation overheads	0.89	9.09	2.4s	0.6	3.81