UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Universidad Politécnica de Cataluña

Facultad de Informática de Barcelona

Laboratori 1

Experimental setup and tools

Izan Cordobilla Blanco (par1304)

Alejandro Salvat Navarro (par1120)

Grado en ingeniería informática

Cuadrimestre de primavera 2021-2022

ÍNDICE

1. SESIÓN 1

1.1. Introducción

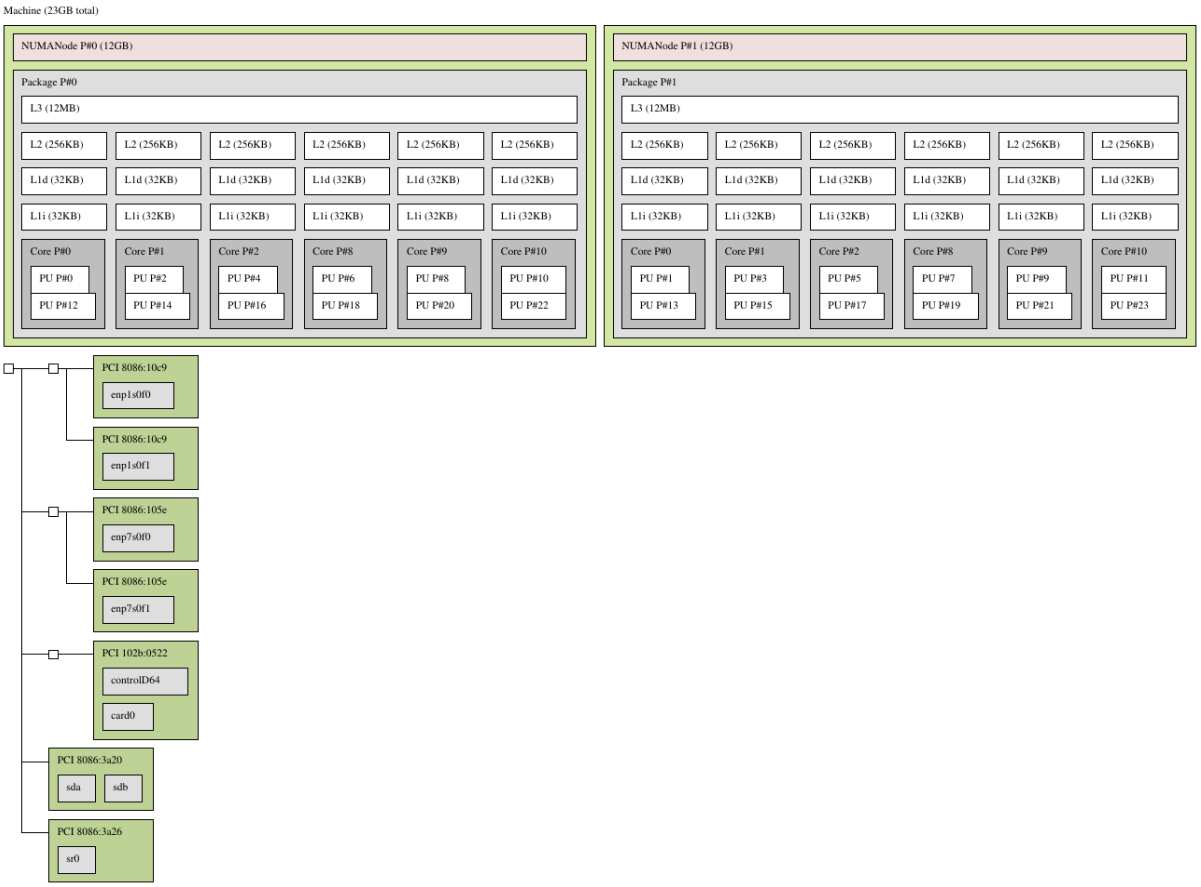
El objetivo de esta primera sesión es familiarizarnos con el entorno de trabajo que vamos a utilizar a partir de ahora para realizar las próximas sesiones de laboratorio. Trabajaremos con un servidor multiprocesador llamado ***Boada***.

Boada consta de nodos que utilizaremos para ejecutar nuestros programas: para la ejecución interactiva utilizaremos *Boada-1* y para utilizar el sistema de colas se podría utilizar de *Boada-2* a *Boada-8*, pero en este curso solo utilizaremos hasta *Boada-4*.

1.2. Node architecture and memory

Para empezar, investigaremos más a fondo la arquitectura dentro de los nodos disponibles de *Boada*. Para visualizar ésta arquitectura, hemos seguido el documento de laboratorio y hemos utilizado los comandos *lscpu,* que sirve para obtener algunos datos de la tabla 1 y *lstpo* (añadiendo la flag –of fig se obtiene la figura 1).

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



En ésta figura, podemos ver la arquitectura interna de un único nodo de *Boada*, pero todos son exactamente iguales.

1.3. Compilation and execution of OpenMP programs

Para obtener los resultados de la parte interactiva de la tabla 2, hemos ejecutado el ejecutable de ***pi_omp.c*** utilizando ***run-omp.sh***, es decir, de manera interactiva. Por un lado, vemos que el *User Time* se mantiene en aproximadamente 8 al utilizar más de un thread. También, podemos observar que aunque aumentemos el número de *threads* el *Elapsed Time* no disminuye y ésto es así porque únicamente se está ejecutando en un procesador.

En cambio, utilizando el sistema de colas podemos reducir el *Elapsed Time* gracias al aumento de *threads* utilizados. Como podemos observar, al doblar el número de *threads* utilizados se disminuye aproximadamente a la mitad el tiempo de ejecución. También, podemos observar que el *User Time* se mantiene aproximadamente en 4. Por último, si observamos el % de CPU utilizada nos damos cuenta que utilizando la manera interactiva éste valor solo crece al pasar de un único thread a dos, mientras que en el sistema de colas se dobla cada vez que doblamos el número de threads.

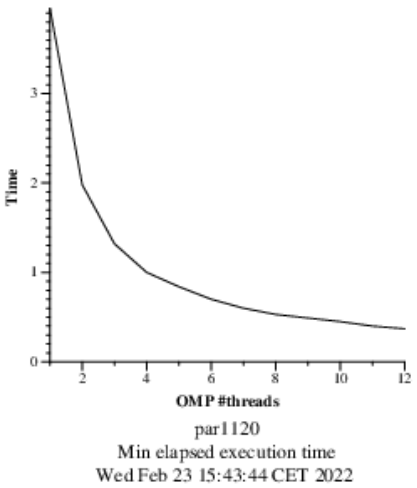
# threads	Interactive				Queued			
	User	System	Elapsed	% of CPU	User	System	Elapsed	% of CPU
1	3.97	0.00	0:03.94	99	3.94	0.00	0:03.97	99
2	7.97	0.00	0:03,99	199	3.95	0.00	0:01.99	198
4	7.93	0.07	0:04.01	199	3.99	0.01	0:01.02	391
8	8.00	0.04	0:04.03	199	4.23	0.00	0:00.55	762

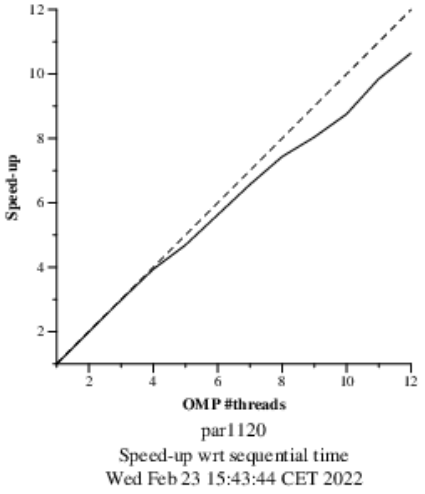
1.4. Strong vs Weak Scalability

En este punto vamos a explorar el concepto de escalabilidad en la versión paralela de **pi_omp.c.** Mediremos la escalabilidad calculando el ratio entre los tiempos de ejecución secuencial y paralela. Consideraremos 2 escenarios.

- *Strong scalability*: El número de threads varía, pero el tamaño del problema queda fijo. En este caso, el paralelismo se usa para reducir el tiempo de ejecución del programa.
- *Weak scalability:* El tamaño del problema es proporcional al número de threads. En este caso, el paralelismo se usa para incrementar el tamaño del problema del programa que se ejecuta.

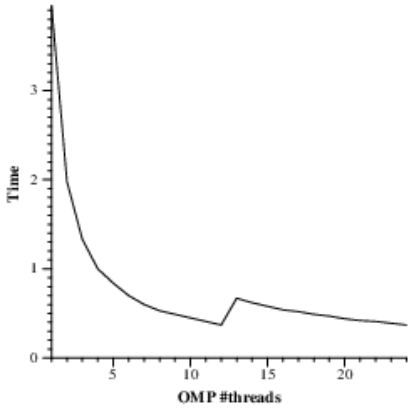
Usaremos los scripts **submit-strong-omp.sh** y **submit-weak-omp.sh** para analizar la escalabilidad. Estos *scripts* ejecutan el código paralelo usando de 1 (NP_min) a 12 (NP_max) threads. Ejecutando el script submit-strong-omp.sh obtenemos las dos siguientes figuras que podremos observarlas gracias al comando gs:

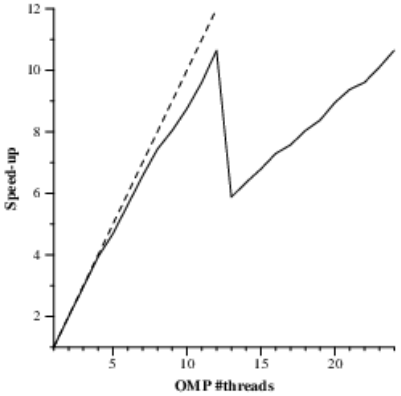




En el primer gráfico podemos observar que aumentando el número de threads disminuye el tiempo de ejecución de manera exponencial. A partir de los 8-10 threads no se nota tanto la disminución del tiempo debido a que los overheads y las sincronizaciones se empiezan a notar.

Ésto lo podemos ver también en la segunda gráfica, dónde se nos muestra cómo evoluciona el speed-up aumentando el número de threads. Hasta el thread número 4 el speed-up aumenta de manera lineal (caso ideal). A partir de aquí, se va desviando hasta llegar al thread #12, donde consigue el máximo speed-up.
Habiendo observado cómo se comporta el programa desde 1 a 12 threads, se nos pide que cambiemos el número de threads máximo (NP_max) de 12 a 24 threads. Volvemos a ejecutar el script submit-strong-omp.sh enviándolo al sistema de colas y obtenemos los siguientes dos gráficos:

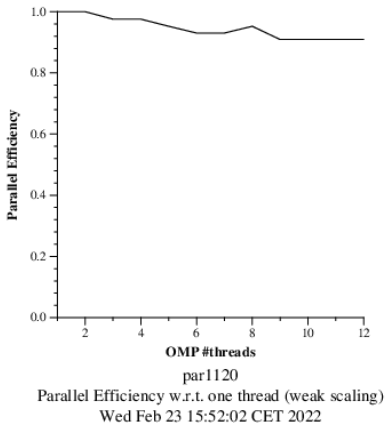




Estas dos figuras nos muestran la continuación del comportamiento de los anteriores dos gráficos con el doble de threads. Salta a la vista que, cuando se llega a los 12 threads, el programa tiene un comportamiento extraño. En la gráfica de tiempo, podemos observar que al llegar a este número de threads, el tiempo de ejecución sube y en el mismo punto del otro gráfico, el speed-up baja de manera drástica.

Ésto es debido a que el nodo de Boada dónde ejecutamos nuestro programa, aunque ponga que dispone de 24 cores solo tiene 12 reales. Por lo tanto, cuando sobrepasamos los 12 cores empieza el HyperThreading (cada core ejecuta más de un thread) y el speed-up baja.

Por último, ejecutamos el script submit-weak-omp.sh sin argumentos y obtenemos el siguiente gráfico:

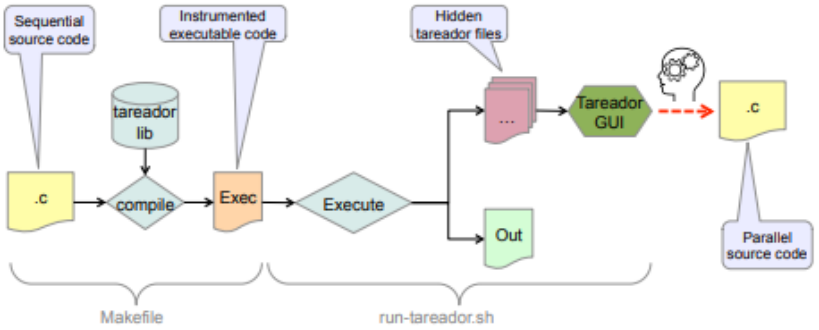


En él podemos ver como con el aumento de threads la paralelización no es ideal y la eficiencia empieza a bajar a causa de factores como los overheads y la sincronización entre threads que provoca el paralelismo.

2. SESIÓN 2

2.1. Introducción

En esta sesión aprenderemos a usar ***Tareador***, una herramienta que permite analizar el potencial paralelismo que podemos obtener aplicando una cierta estrategia de descomposición de tareas. Ésta herramienta solo necesita que el programador identifique las tareas que quiere evaluar dentro de una estrategia de descomposición de tareas. En la siguiente figura se puede observar el comportamiento de *Tareador*:



2.2. Task Decomposition for 3dfft

En este apartado analizaremos diferentes versiones del código 3dfft_tar.c usando Tareador. Visualizaremos los grafos de dependencia junto a simulaciones de ejecución de las distintas versiones. De esta manera, observaremos el tiempo de ejecución siguiendo la estrategia de paralelismo con un (T_1) e infinitos (T_∞) procesadores. Además de esto, también calcularemos el paralelismo obtenido con la fórmula correspondiente:

$$Paralelismo = \frac{T1}{T\infty}$$

Version	T_1	T_∞	Parallelism
seq	639780	639780	1
v1	639780	639760	1
v2	639780	361190	1,77
v3	639780	154354	4,14
v4	639780	64018	9,99
v5	639780	8255	77,50

2.2.1. Original Version (Seq)

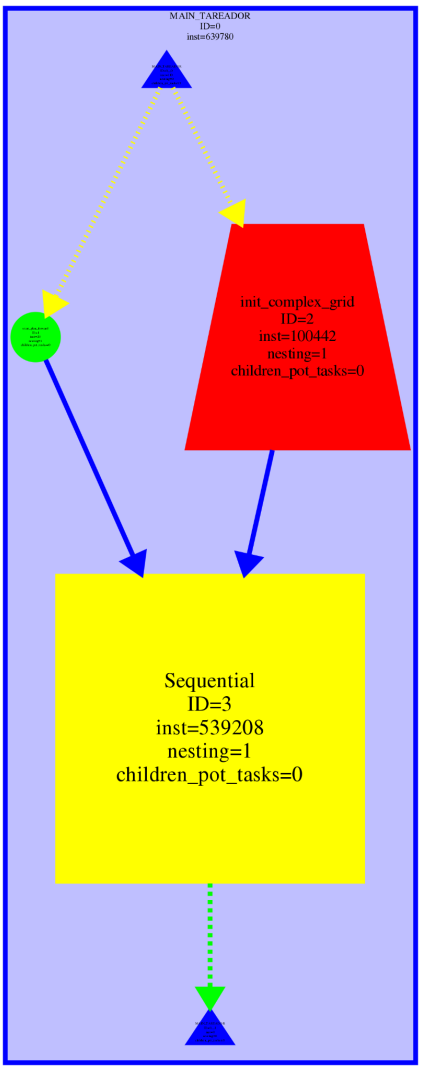
Al tratarse de una versión del código secuencial, el tiempo de ejecución con un procesador es igual al obtenido con procesadores infinitos, ya que la parte paralelizable del programa es inexistente. Por lo tanto, aplicando la anterior fórmula de paralelismo, obtenemos que:

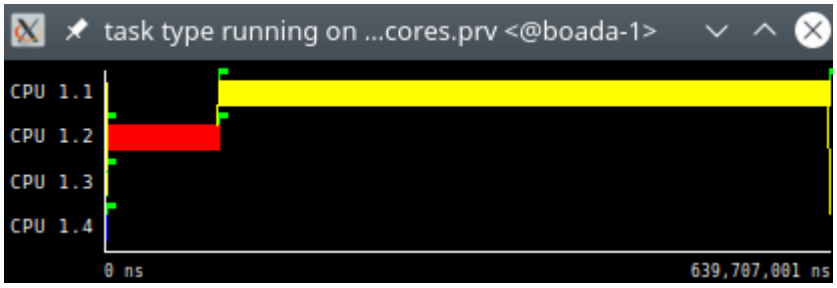
$$Paralelismo = \frac{T1}{T\infty} = \frac{639780}{639780} = 1$$

En las siguientes imágenes se puede observar el código que se ha utilizado para la ejecución, el Task Dependency Graph generado y visualizado con Tareador y una simulación con 4 threads de la ejecución (sería prácticamente lo mismo si hiciéramos la simulación con un único thread).

```
START_COUNT_TIME;

tareador_start_task("Sequential");

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("Sequential");

STOP_COUNT_TIME("Execution FFT3D");
```

3dfft_original.c





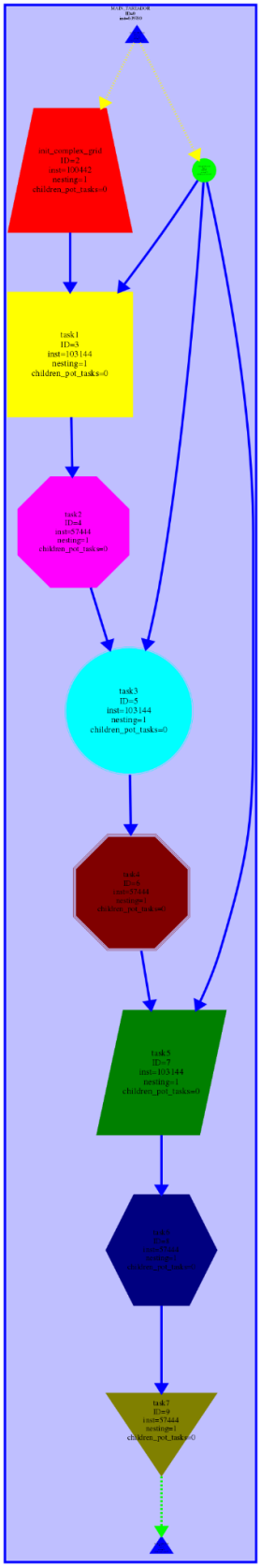
2.2.2. Version 1

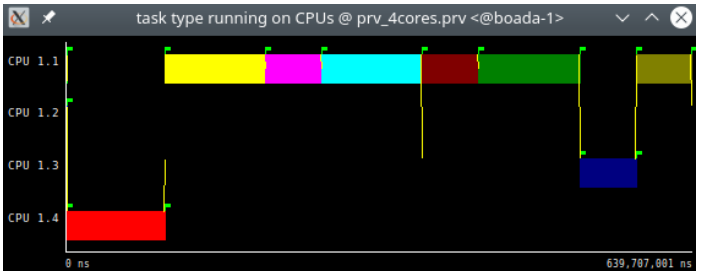
En esta primera versión, hemos reemplazado la tarea “ffts1_and_transpositions” por una tarea de Tareador por cada función del main. De esta manera, logramos descomponer el código en una secuencia de tareas más pequeñas. El problema es que, al depender las unas de las otras, aun siendo distintas tareas más pequeñas, la parte paralelizable del programa sigue siendo prácticamente nula, obteniendo el siguiente paralelismo:

$$Paralelismov1 = \frac{T1}{T\infty} = \frac{639780}{639760} = 1,0000312 \cong 1$$

En las siguientes imágenes se muestran los cambios realizados en el código, donde se puede ver que por cada invocación de una función dentro de lo que antes era la tarea “ffts1_and_transpositions”, se ha puesto una tarea de Tareador. También podemos ver el TDG y las dependencias de unas tareas sobre otras. Por último, se puede ver una simulación con 4 threads donde se aprecia que la ejecución es prácticamente secuencial.

```
START_COUNT_TIME;

tareador_start_task("task1");
ffts1_planes(p1d, in_fftw);
tareador_end_task("task1");

tareador_start_task("task2");
transpose_xy_planes(tmp_fftw, in_fftw);
tareador_end_task("task2");

tareador_start_task("task3");
ffts1_planes(p1d, tmp_fftw);
tareador_end_task("task3");

tareador_start_task("task4");
transpose_zx_planes(in_fftw, tmp_fftw);
tareador_end_task("task4");

tareador_start_task("task5");
ffts1_planes(p1d, in_fftw);
tareador_end_task("task5");

tareador_start_task("task6");
transpose_zx_planes(tmp_fftw, in_fftw);
tareador_end_task("task6");

tareador_start_task("task7");
transpose_xy_planes(in_fftw, tmp_fftw);
tareador_end_task("task7");

STOP_COUNT_TIME("Execution FFT3D");
```




task type running on CPUs @ prv_4cores.prv <@boada-1>
CPU 1.1
CPU 1.2
CPU 1.3
CPU 1.4
0 ns
639,707,001 ns

2.2.3. Version 2

En esta segunda versión, reemplazaremos la definición de las invocaciones a función asociadas al bucle de ffts1_planes, por tareas con granularidad más fina.

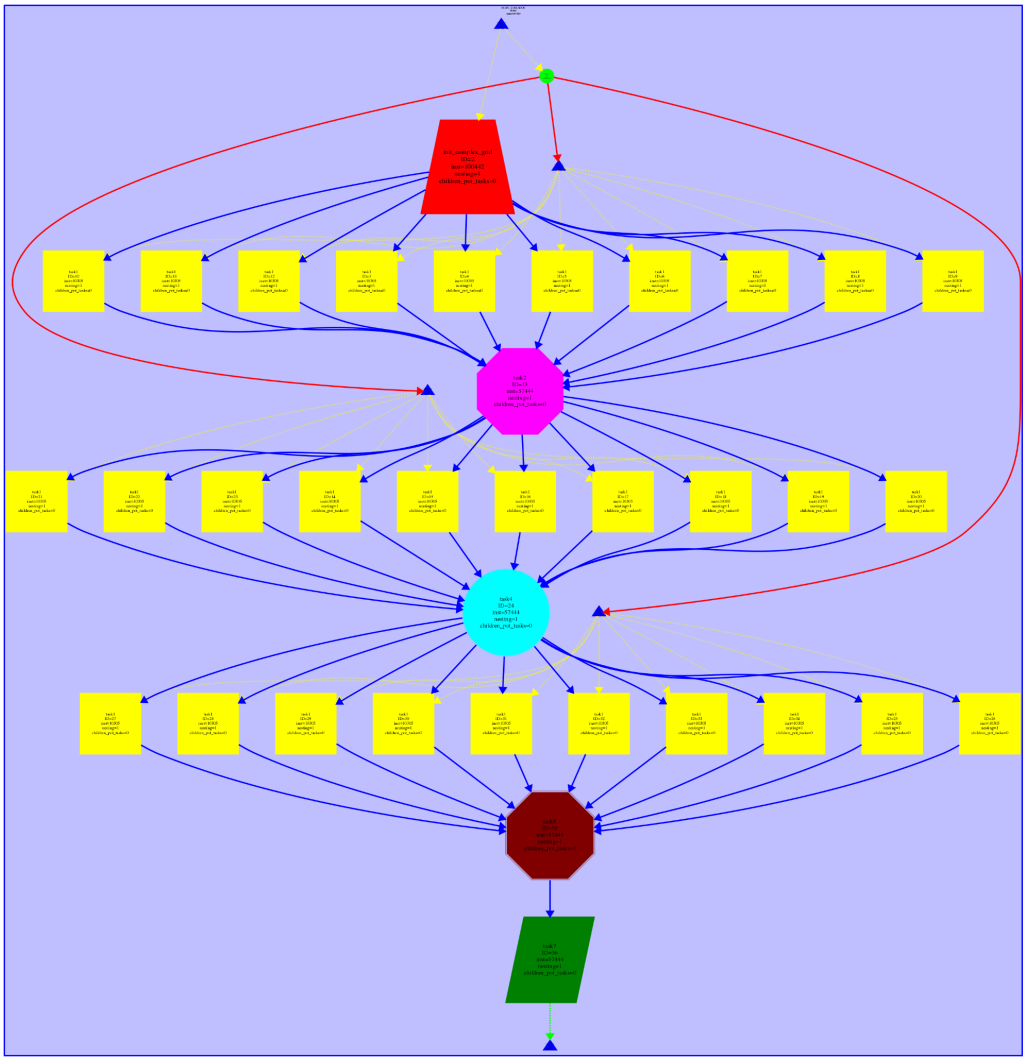
De esta manera, al separar el bucle anteriormente secuencial en diversas tareas más pequeñas sin dependencia entre ellas, conseguiremos que parte del programa sea paralelizable, obteniendo una mejora de tiempo de ejecución usando los procesadores necesarios. El paralelismo obtenido en esta versión de código es:

$$Paralelismov2 = \frac{T1}{T\infty} = \frac{639780}{361190} = 1,771311498 \cong 1,77$$

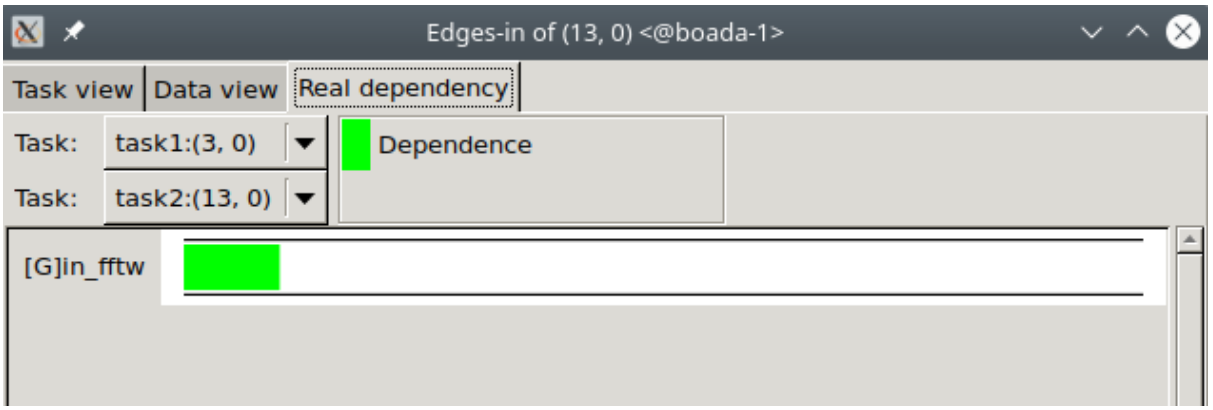
En las siguientes imágenes se muestran los cambios realizados en el código, donde cabe destacar la nueva localización de las tareas de Tareador llamadas “task1”. También, se puede observar que en el TDG aparecen tres grandes bloques horizontales de color amarillo. Ésto son las llamadas a la función **ftts_planes**, y aparecen segmentadas ya que es la parte que hemos paralelizado.

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

    for (k=0; k<N; k++) {
      tareador_start_task("task1");
        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("task1");
    }
}
```

```
    START_COUNT_TIME;

    ffts1_planes(p1d, in_fftw);

    tareador_start_task("task2");
    transpose_xy_planes(tmp_fftw, in_fftw);
    tareador_end_task("task2");

    ffts1_planes(p1d, tmp_fftw);

    tareador_start_task("task4");
    transpose_zx_planes(in_fftw, tmp_fftw);
    tareador_end_task("task4");

    ffts1_planes(p1d, in_fftw);

    tareador_start_task("task6");
    transpose_zx_planes(tmp_fftw, in_fftw);
    tareador_end_task("task6");

    tareador_start_task("task7");
    transpose_xy_planes(in_fftw, tmp_fftw);
    tareador_end_task("task7");

    STOP_COUNT_TIME("Execution FFT3D");
```



Clicando click derecho sobre un nodo, podemos ver edges-in para saber qué dependencias tiene ese nodo. Mirando las dependencias de la función ffts_planes (como se puede ver en la siguiente imagen), nos damos cuenta que las dependencias están generadas por la variable [G]in_fftw.



2.2.4. Version 3

En esta tercera versión deberemos cambiar la definición de tareas pertenecientes a los bucles de las funciones transpose_xy_planes y transpose_zx_planes de igual manera que hemos hecho en la versión anterior. Realizada esta configuración, el paralelismo del código es:

$$Paralelismov3 = \frac{T1}{T\infty} = \frac{639780}{154354} = 4,1448877 \cong 4,14$$

Como en la versión anterior, cabe destacar la nueva localización de las tareas de Tareador en las funciones transpose_xy_planes y transpose_zx_planes. En este caso, además de ver la descomposición de las funciones ftts_planes, también podemos observar la descomposición de transpose_xy_planes y transpose_zx_planes, que aparecen representadas por los colores magenta y cian respectivamente.

```
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");
        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");
    }
}

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");
        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");
    }
}
```

```
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);

STOP_COUNT_TIME("Execution FFT3D");
```

2.2.5. Version 4

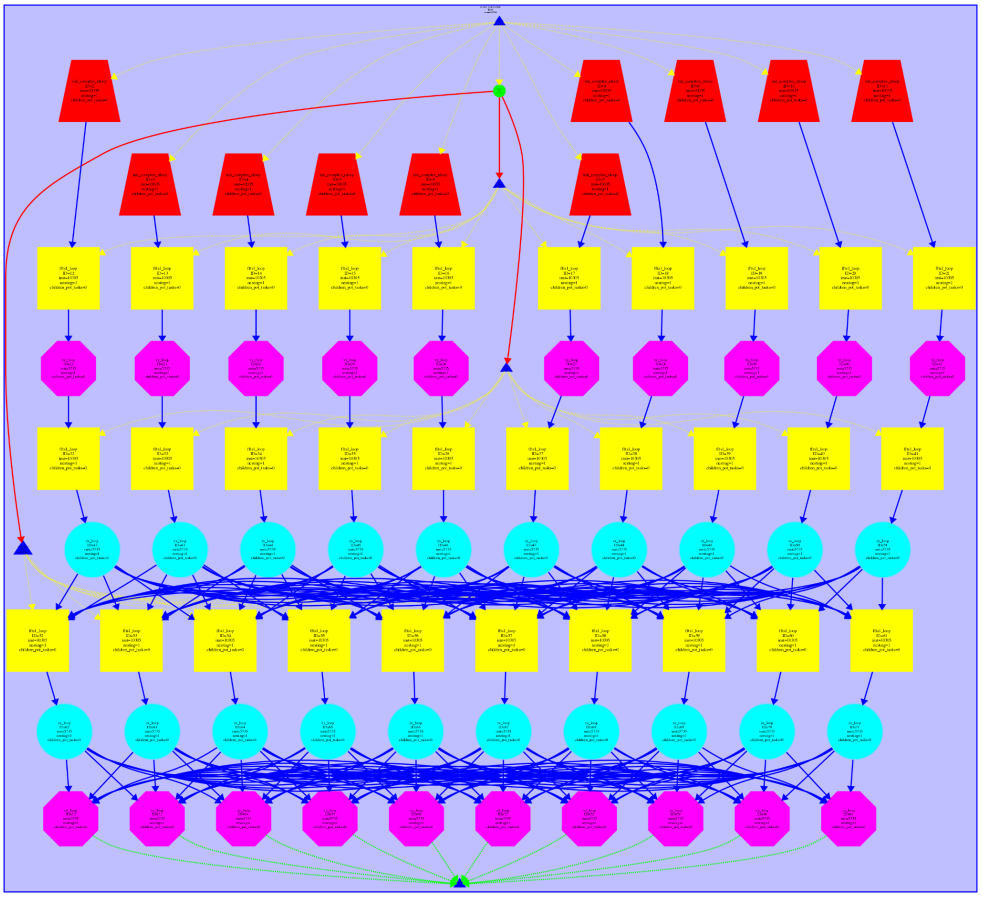
Como hemos podido observar en el TDG de la versión anterior, la única parte que no estaba paralelizada era la función init_complex_grid. Por eso, en esta versión hemos paralelizado las llamadas a función restantes del main, reemplazando la definición de tareas de la función init_complex_grid, tal y como se muestra en las siguientes figuras. Con esta configuración, obtendremos el siguiente paralelismo:

$$Paralelismov4 = \frac{T1}{T\infty} = \frac{639780}{64018} = 9,993751757 \cong 9{,}99$$

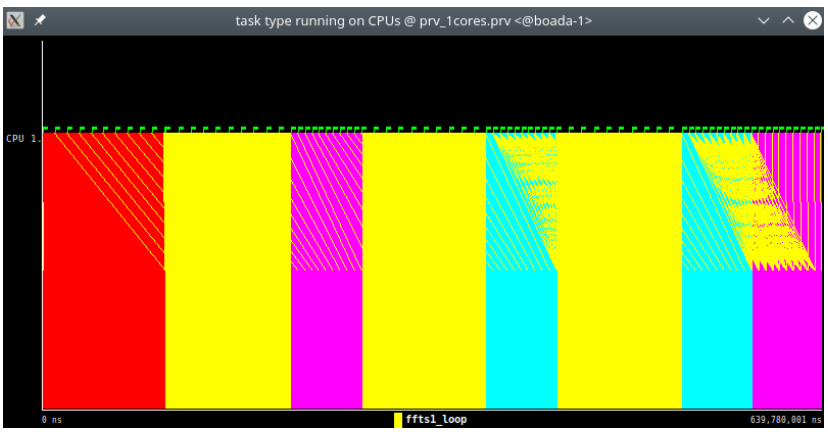
```
void init_complex_grid(fftwf_complex in_fftw[][N][N]) {
  int k,j,i;

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

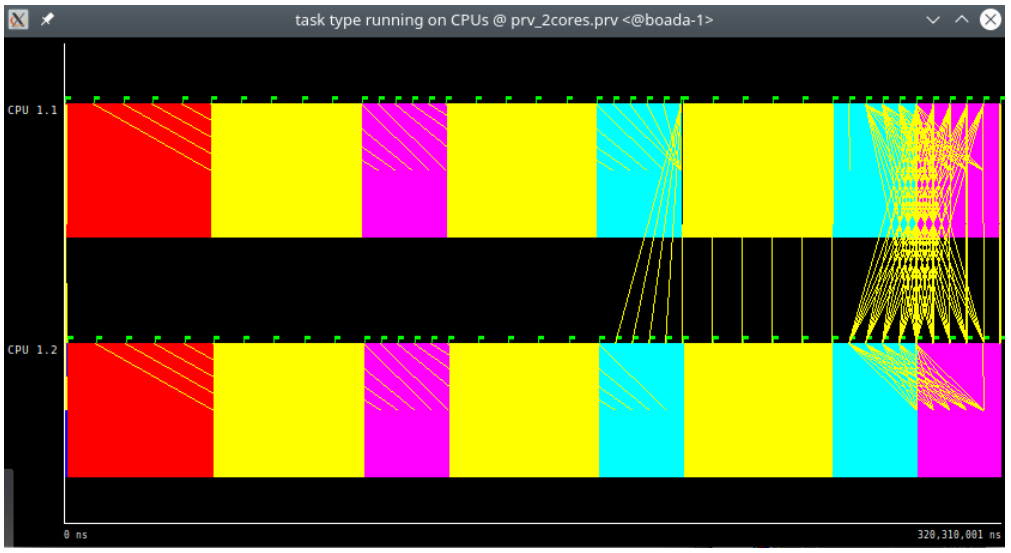
```
START_COUNT_TIME;

init_complex_grid(in_fftw);

STOP_COUNT_TIME("Init Complex Grid FFT3D");
```

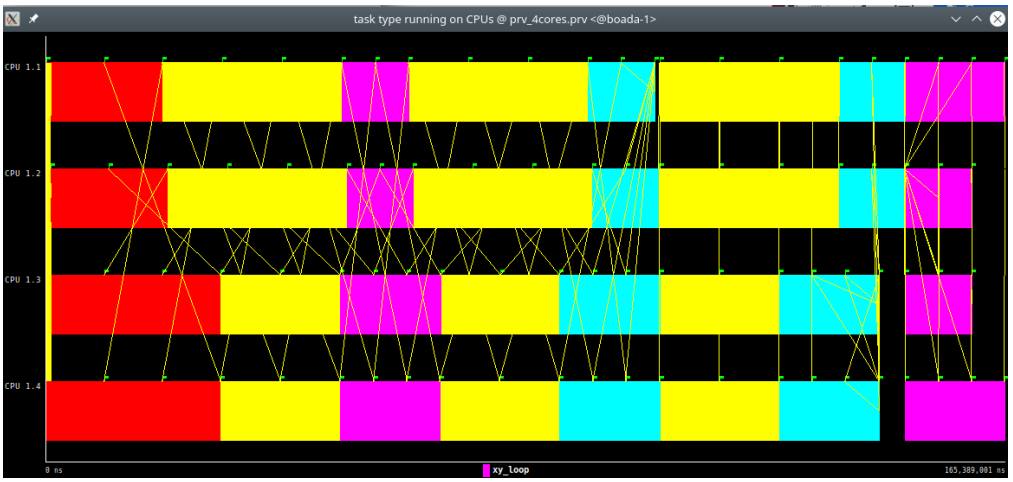
En esta versión, nos sale el siguiente Task Dependency Graph:

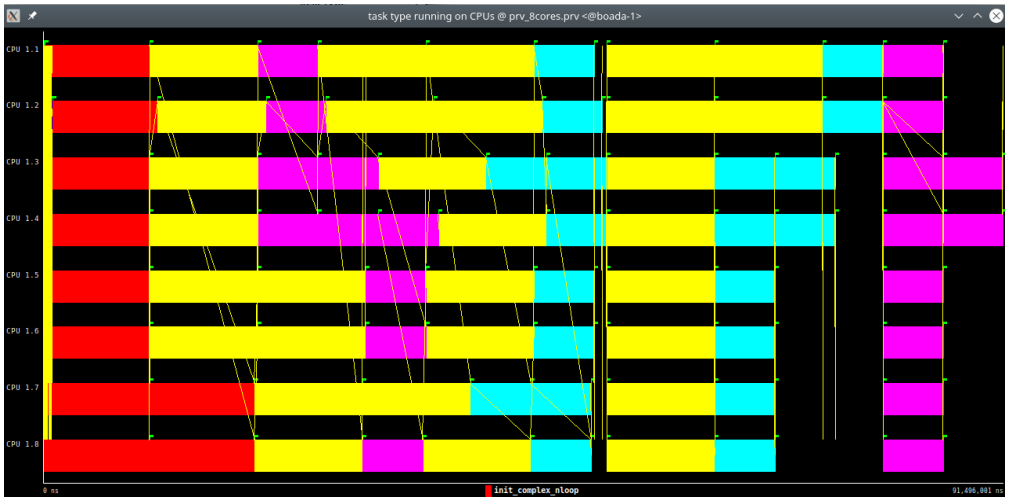


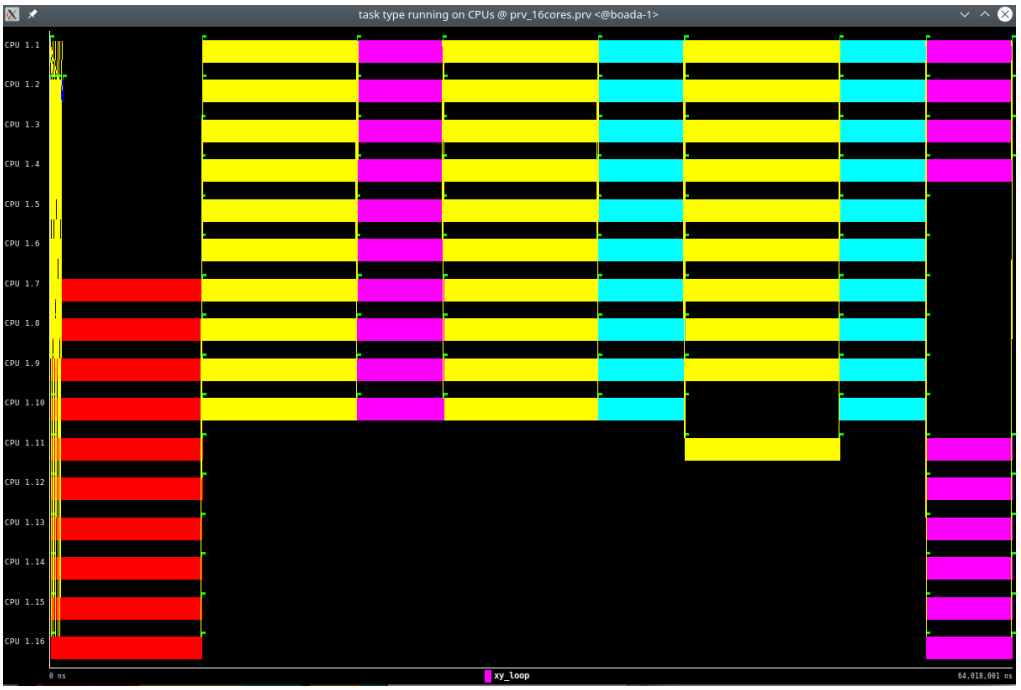
En él se puede observar que ya hemos segmentado la función init_complex_grid por tareas más pequeñas, y, aunque todavía se puede paralelizar el código en tareas todavía más pequeñas, ya que, al incluir el bucle for interno entero dentro de la definición de tarea, no estamos haciendo el programa todo lo paralelizable que podemos, haciendo que las dependencias eviten al programa aproveche todos los procesadores para hacer la paralelización (lo veremos en la siguiente versión), hemos obtenido un paralelismo bastante importante. A continuación se podrán ver las simulaciones hechas con 1, 2, 4, 8, 16 y 32 threads.

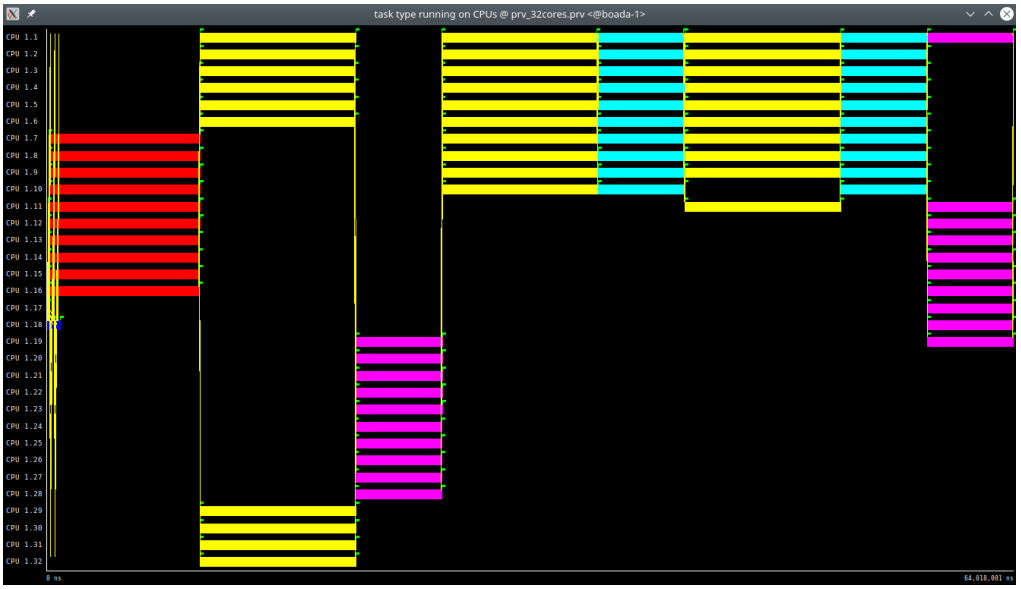



task type running on CPUs @ prv_2cores.prv <@boada-1>
CPU 1.1
CPU 1.2
0 ns
320,310,001 ns


task type running on CPUs @ prv_4cores.prv <@boada-1>
CPU 1.1
CPU 1.2
CPU 1.3
CPU 1.4
0 ns
xy_loop
165,389,001 ns


task type running on CPUs @ prv_8cores.prv <@boada-1>
CPU 1.1
CPU 1.2
CPU 1.3
CPU 1.4
CPU 1.5
CPU 1.6
CPU 1.7
CPU 1.8
0 ns
init_complex_nloop
91,466,001 ns

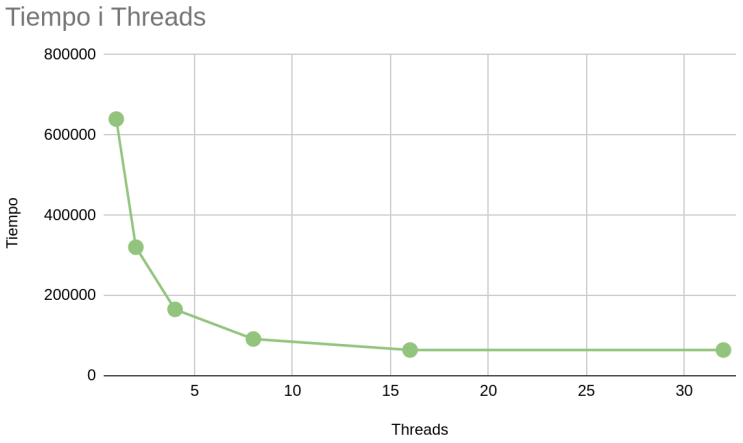

task type running on CPUs @ prv_16cores.prv <@boada-1>
CPU 1.1
CPU 1.2
CPU 1.3
CPU 1.4
CPU 1.5
CPU 1.6
CPU 1.7
CPU 1.8
CPU 1.9
CPU 1.10
CPU 1.11
CPU 1.12
CPU 1.13
CPU 1.14
CPU 1.15
CPU 1.16
0 ns
xy_loop
64,018,001 ns


task type running on CPUs @ prv_32cores.prv <@boada-1>
CPU 1.1
CPU 1.2
CPU 1.3
CPU 1.4
CPU 1.5
CPU 1.6
CPU 1.7
CPU 1.8
CPU 1.9
CPU 1.10
CPU 1.11
CPU 1.12
CPU 1.13
CPU 1.14
CPU 1.15
CPU 1.16
CPU 1.17
CPU 1.18
CPU 1.19
CPU 1.20
CPU 1.21
CPU 1.22
CPU 1.23
CPU 1.24
CPU 1.25
CPU 1.26
CPU 1.27
CPU 1.28
CPU 1.29
CPU 1.30
CPU 1.31
CPU 1.32
0 ns
64,018,001 ns

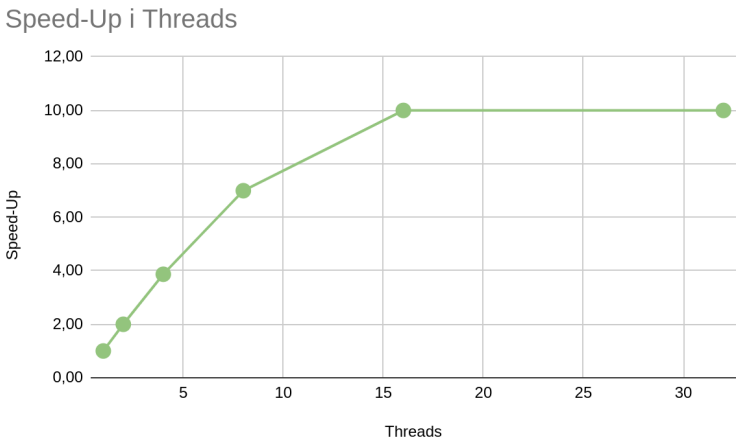
Gracias a estas simulaciones, podemos observar que a medida que aumentamos el número de procesadores el tiempo de ejecución disminuye hasta que, cuando llegamos a los 16, para de disminuir ya que no se usan todos los procesadores.

También, podemos extraer las siguientes dos tablas y gráficas que muestran más fácilmente cómo evoluciona el tiempo y el speed-up con el incremento de procesadores.

Threads	Tiempo
1	639780
2	320310
4	165389
8	91496
16	64018
32	64018



Threads	Speed-Up
1	1,00
2	2,00
4	3,87
8	6,99
16	9,99
32	9,99



En ellas, se puede observar que al llegar a los 16 procesadores, tanto el tiempo de ejecución como el speed-up se quedan estancados.

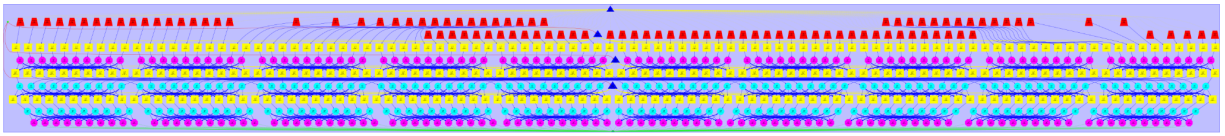
2.2.6. Versión 5

En esta quinta y última versión de nuestro código, exploraremos tareas con una granularidad todavía más baja. Para conseguirlo, hemos definido dichas tareas en bucles más internos. Con esta configuración, obtenemos el siguiente paralelismo:

$$Paralelismov5 = \frac{T1}{T\infty} = \frac{639780}{8255} = 77,502119927 \cong 77{,}50$$

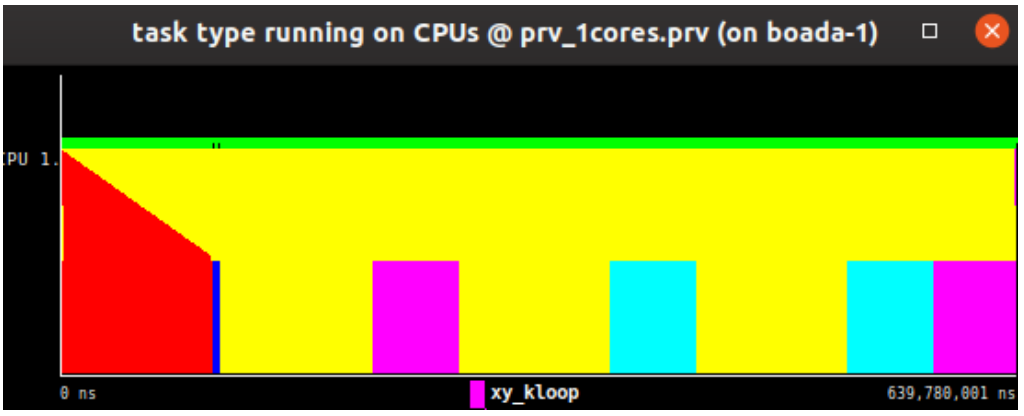
Estas modificaciones se pueden ver reflejadas en el siguiente código, donde hemos metido las tareas de Tareador en el segundo bucle de las funciones. Al explorar este código con Tareador, obtenemos el TDG siguiente:

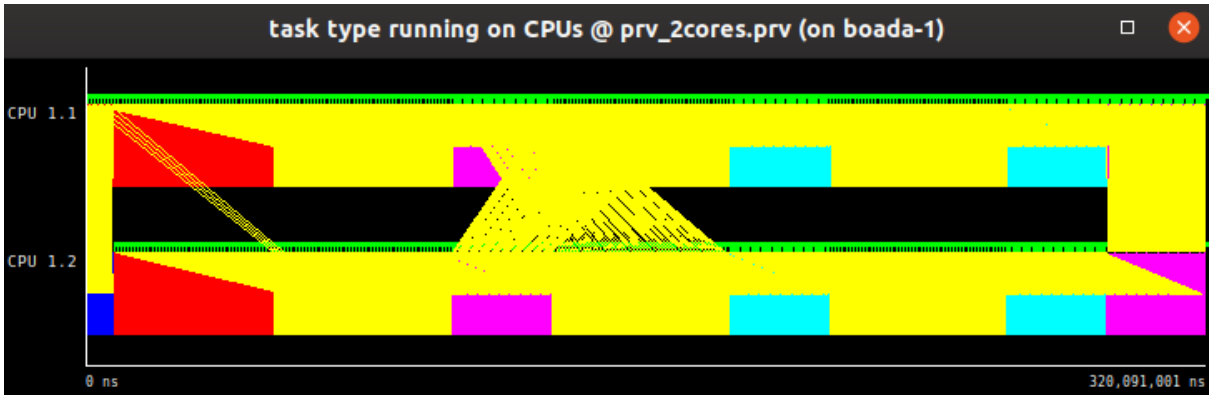
```
void init_complex_grid(fftwf_complex in_fftw[][N][N]) {
  int k,j,i;

  for (k = 0; k < N; k++) {
      for (j = 0; j < N; j++) {
        tareador_start_task("init_complex_kloop");
        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_kloop");
    }
  }
}

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++) {
            tareador_start_task("xy_kloop");
            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_kloop");
        }
    }
}

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++) {
            tareador_start_task("zx_kloop");
            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_kloop");
        }
    }
}

void ffts1_planes(fftwf_plan pld, 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_kloop");
            fftwf_execute_dft( pld, (fftwf_complex *)in_fftw[k][j][0],
                               (fftwf_complex *)in_fftw[k][j][0]);
            tareador_end_task("ffts1_kloop");
        }
    }
}
```

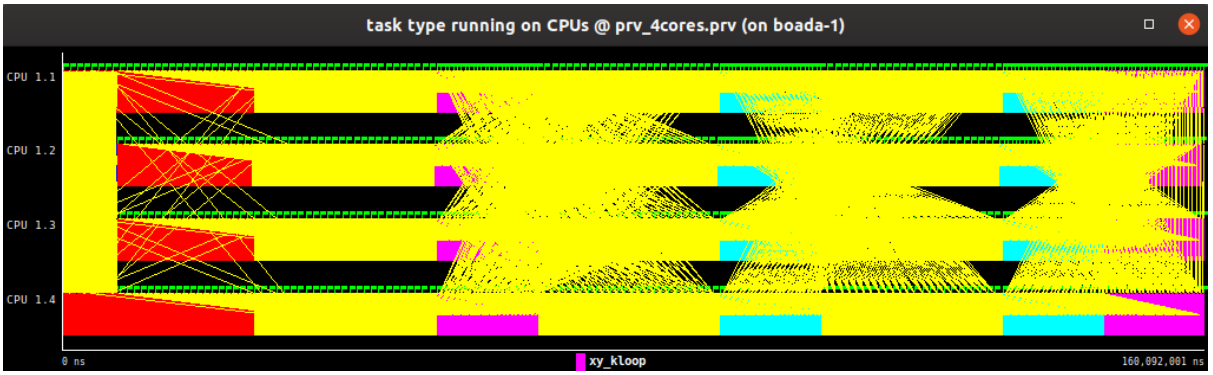


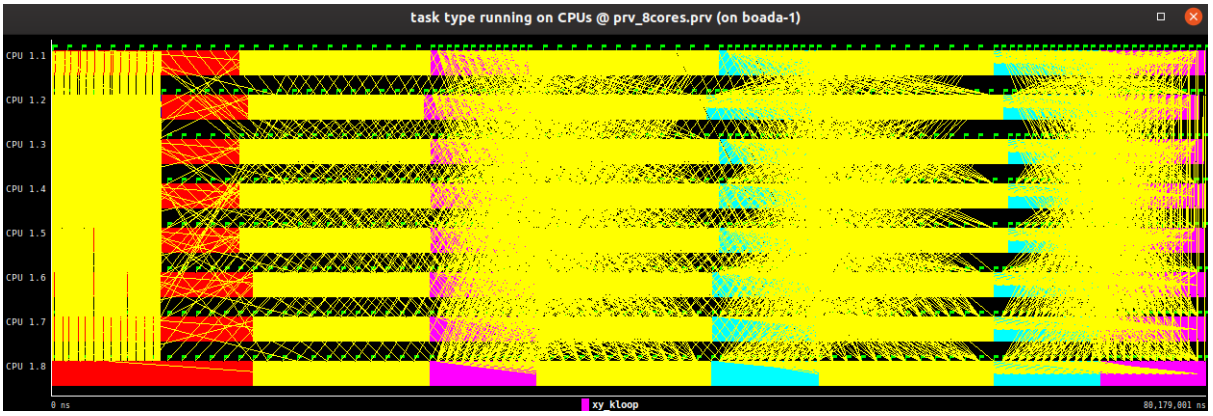
En este TDG vemos que, a diferencia de en la versión anterior, gracias a colocar la definición de tarea en el bucle más interno, podemos crear más tareas con una granularidad mucho más baja. De esta manera, aprovechamos mucho mejor la cantidad de threads para paralelizar el proceso, haciendo que el efecto de la dependencia sea casi negligible y, como resultado, obteniendo un programa con un paralelismo muy superior.

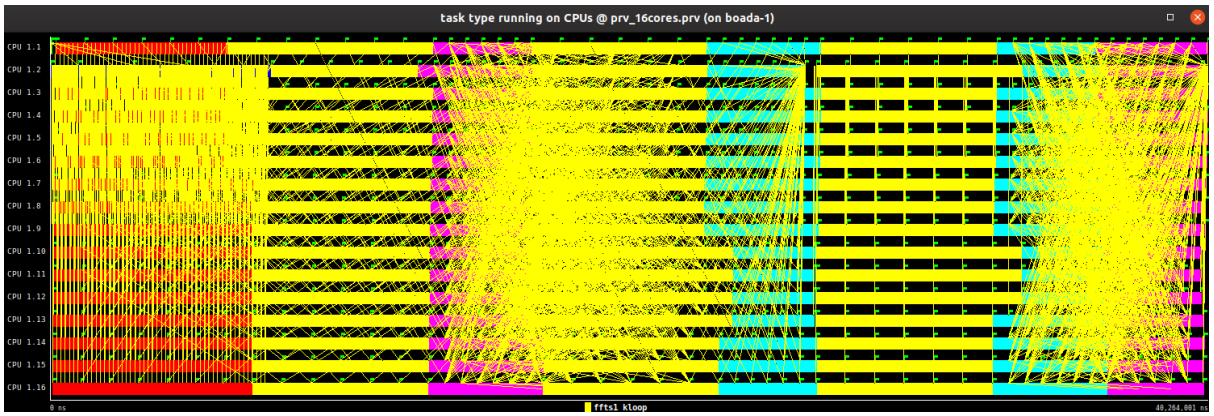
Además, al igual que hemos hecho con la versión 4, hemos hecho simulaciones utilizando 1, 2, 4, 8, 16 y 32 procesadores. Éstas se pueden observar en las siguientes figuras:

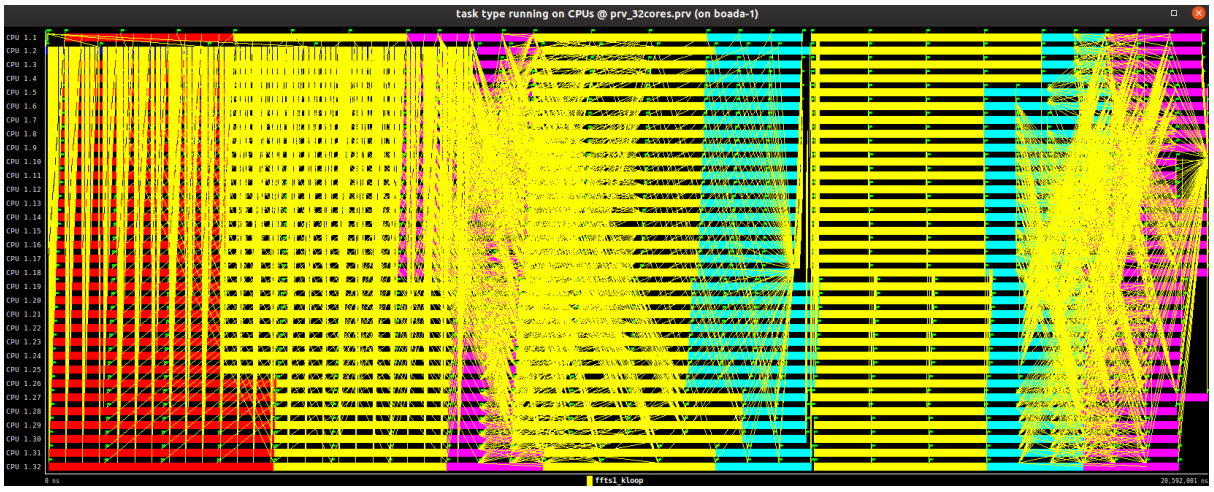




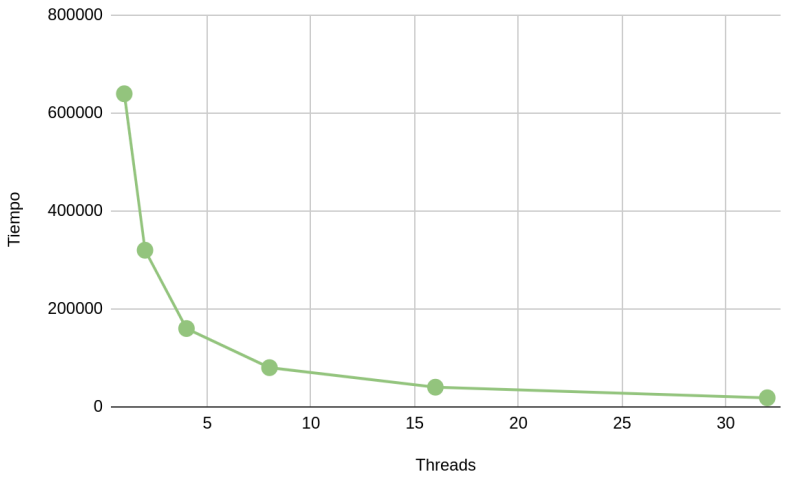



task type running on CPUs @ prv_8cores.prv (on boada-1)
CPU 1.1
CPU 1.2
CPU 1.3
CPU 1.4
CPU 1.5
CPU 1.6
CPU 1.7
CPU 1.8
0 ns
xy_kloop
80,179,001 ns


task type running on CPUs @ prv_16cores.prv (on boada-1)
CPU 1.1
CPU 1.2
CPU 1.3
CPU 1.4
CPU 1.5
CPU 1.6
CPU 1.7
CPU 1.8
CPU 1.9
CPU 1.10
CPU 1.11
CPU 1.12
CPU 1.13
CPU 1.14
CPU 1.15
CPU 1.16
0 ns
fft1_kloop
48,264,001 ns

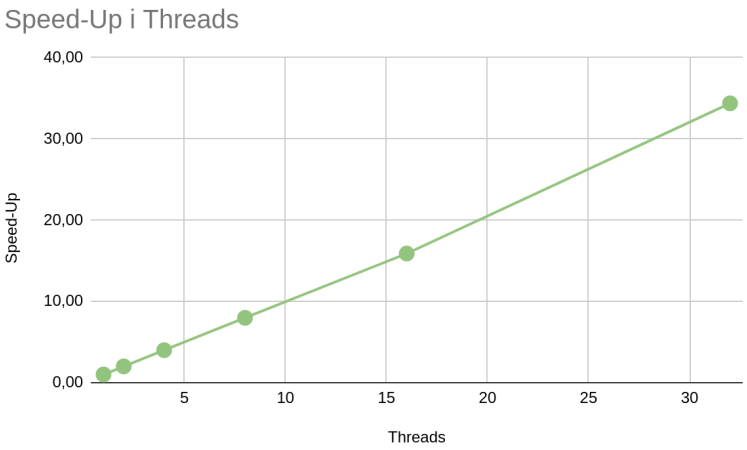

task type running on CPUs @ prv_32cores.prv (on boada-1)
0 ns
fft1_kloop
20,592,001 ns

Threads	Tiempo
1	639780
2	320091
4	160092
8	80179
16	40264
32	18607



A diferencia de la versión 4, en ésta podemos ver que el aumento de threads disminuye el tiempo de ejecución ya que al tener un paralelismo tan alto, se aprovechan todos los procesadores. Por tanto, el speed-up aumenta desde 1 a 32 threads, no se queda estancado como podemos ver en el siguiente gráfico:

Threads	Speed-Up
1	1,00
2	2,00
4	4,00
8	7,98
16	15,89
32	34,38



3. SESIÓN 3

3.1. Introducción

En esta sesión de laboratorio, nos familiarizaremos con el entorno Paraver, el cual nos permite obtener información y visualizar simulaciones de la ejecución paralela de un programa en OpenMP. Este entorno también se compone por Extrae y Paraver. Extrae proporciona una API para definir manualmente, en el código fuente, puntos donde emitir eventos. De todas maneras, Extrae lo usaremos para obtener información sobre el estado de cada thread y los diferentes eventos relacionados con la ejecución paralela del programa. Por otra parte, Paraver nos permite visualizar gráficos de tiempo, de manera que podamos analizar la ejecución del programa.

3.2. Obtaining parallelisation metrics for 3DFFT using modelfactors

3.2.1. Initial Version

Empezaremos trabajando con el fichero 3dfft_omp.c. Ejecutaremos el fichero con el script **submit_strong_extrae.sh** de manera que se simulara la ejecución del programa con 1, 2, 4, 6, 8 y 12 threads. De todas estas simulaciones, obtendremos información sobre el tiempo invertido en la ejecución, el speedup y la eficiencia. Vemos en las tablas, que la fracción paralela del programa es el 71,39%. Además, como podemos observar en las gráficas, la escalabilidad del programa no es apropiada, debido a que no se ve una mejora significativa del speedup según se aumenta el número de threads. Además, se puede ver que según aumenta el número de threads, la eficiencia va cayendo en picado, por lo tanto, la eficiencia de las regiones paralelizadas tampoco es apropiada. El factor que nos está causando más problemas se trata de init_complex_grid.

Overview of whole program execution metrics:

Number of processors	1	2	4	6	8	10	12
Elapsed time (sec)	2.26	1.86	1.28	1.26	1.26	1.28	1.30
Speedup	1.00	1.21	1.76	1.79	1.79	1.77	1.73
Efficiency	1.00	0.61	0.44	0.30	0.22	0.18	0.14

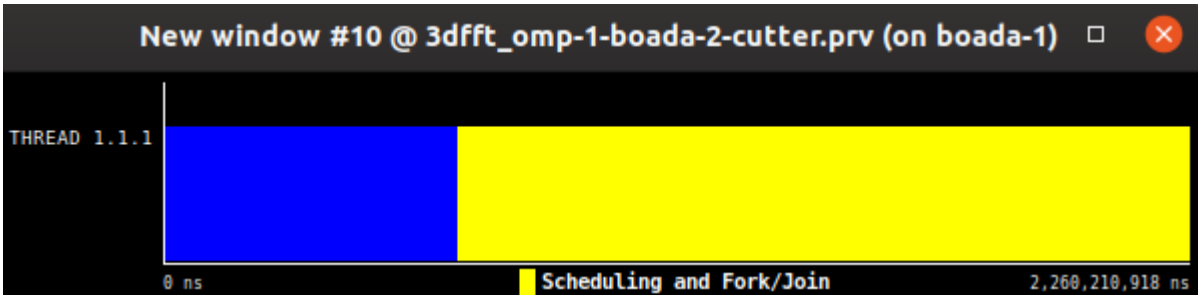
Overview of the Efficiency metrics in parallel fraction:

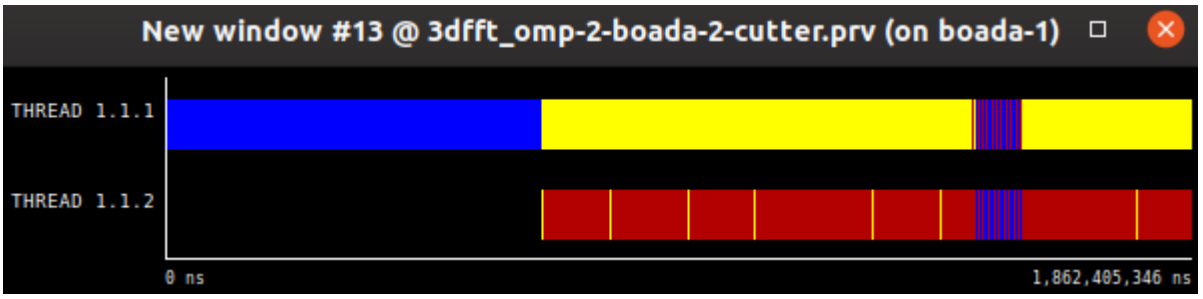
Number of processors	1	2	4	6	8	10	12
Parallel fraction	71.39%						
Global efficiency	95.83%	65.55%	55.40%	38.49%	28.92%	22.67%	18.25%
-- Parallelization strategy efficiency	95.83%	90.27%	84.45%	82.85%	81.41%	81.19%	79.83%
-- Load balancing	100.00%	99.07%	98.61%	98.41%	98.29%	98.33%	98.19%
-- In execution efficiency	95.83%	91.12%	85.64%	84.19%	82.82%	82.56%	81.30%
-- Scalability for computation tasks	100.00%	72.61%	65.59%	46.45%	35.52%	27.92%	22.86%
-- IPC scalability	100.00%	87.73%	74.42%	59.93%	49.88%	44.72%	39.22%
-- Instruction scalability	100.00%	99.46%	98.45%	97.46%	96.48%	95.53%	94.59%
-- Frequency scalability	100.00%	83.22%	89.53%	79.54%	73.82%	65.37%	61.63%

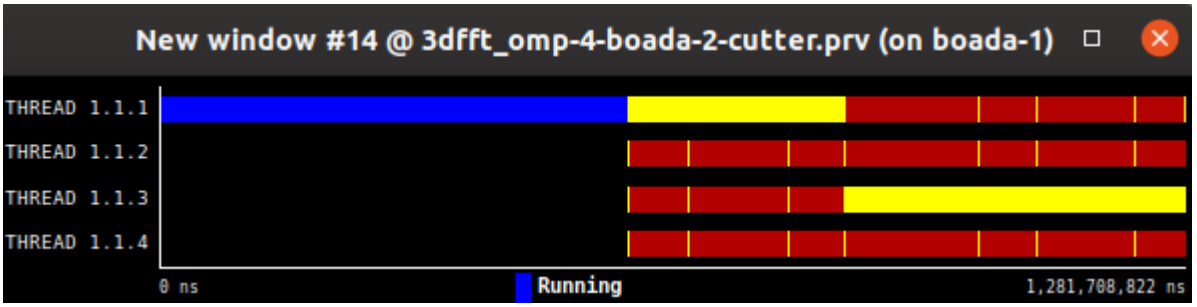
Statistics about explicit tasks in parallel fraction

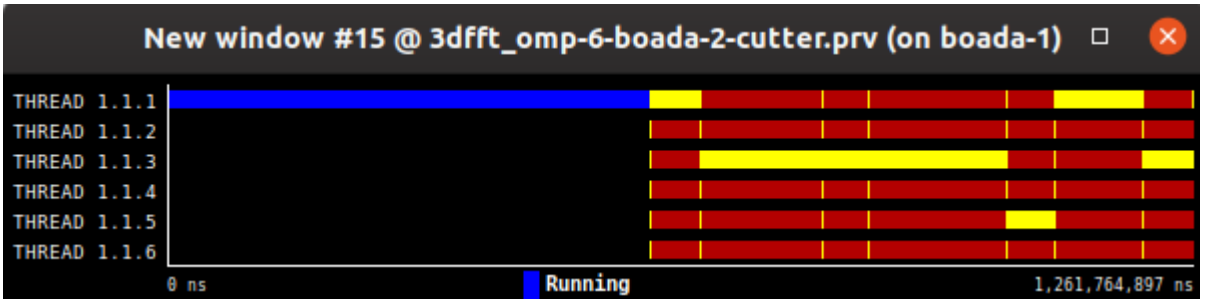
Number of processors	1	2	4	6	8	10	12
Number of explicit tasks executed (total)	17920.0	35840.0	71680.0	107520.0	143360.0	179200.0	215040.0
LB (number of explicit tasks executed)	1.0	0.96	0.95	0.93	0.93	0.89	0.9
LB (time executing explicit tasks)	1.0	0.99	0.98	0.98	0.98	0.98	0.98
Time per explicit task (average)	85.93	59.23	32.79	30.89	30.31	30.86	31.41
Overhead per explicit task (synch %)	0.52	9.54	16.82	19.11	21.09	21.4	23.33
Overhead per explicit task (sched %)	3.85	1.25	1.61	1.59	1.73	1.75	1.91
Number of taskwait/taskgroup (total)	1792.0	1792.0	1792.0	1792.0	1792.0	1792.0	1792.0

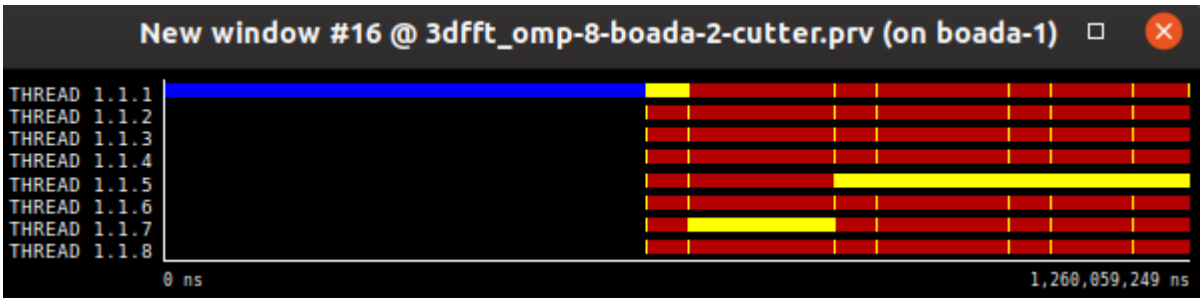
Esta figura nos muestra la tabla resultante de ejecutar 3dfft_omp.c con el script **submit_strong_extrae.sh.** Dicha tabla nos muestra los tiempos de ejecución del programa con distintos threads, el speedup y la eficiencia. Cabe resaltar que el speedup no evoluciona de manera notable a partir de los 4 threads, además de que la eficiencia cae en picado según aumenta el número de threads

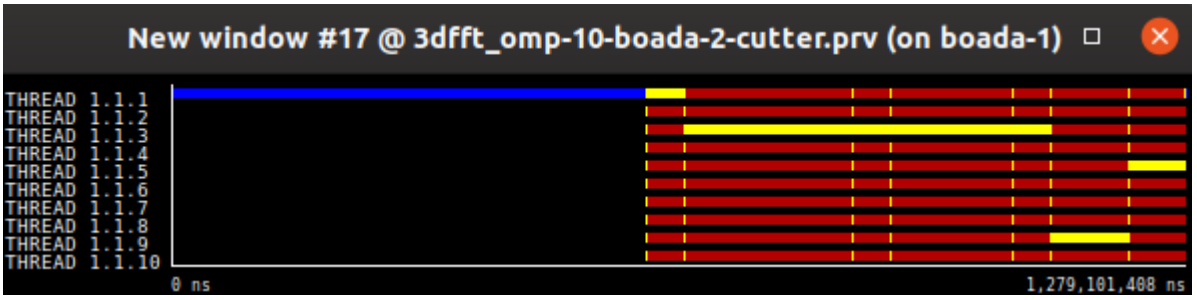


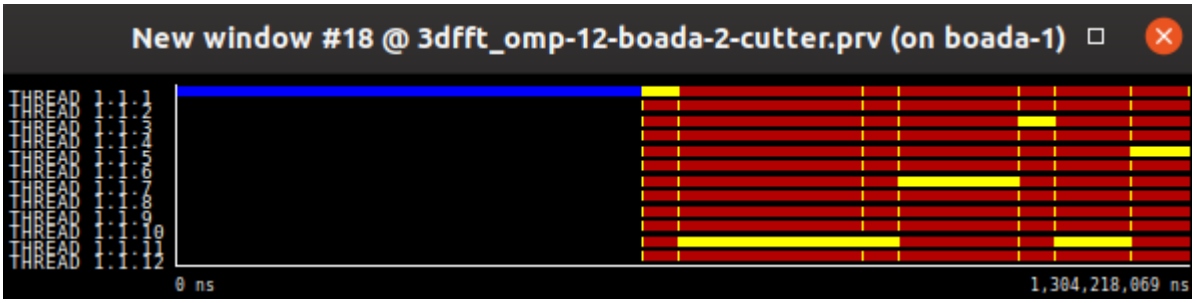




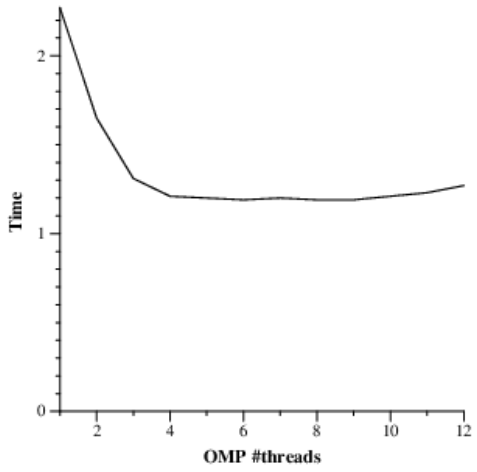


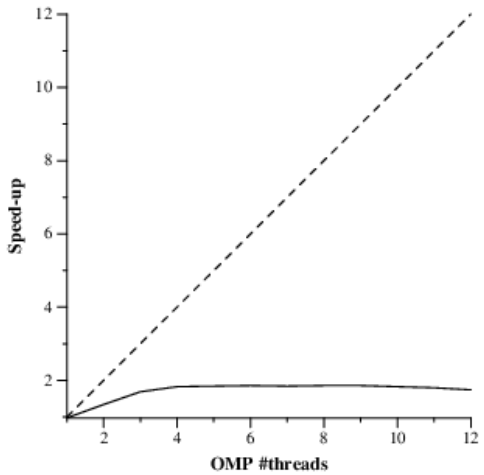






En estas figuras podemos ver la ejecución del programa con 1, 2, 4, 6, 8, 10 y 12 threads. Se puede observar que, al igual que podemos observar en la tabla, a partir de los 4 threads no se percibe prácticamente ninguna mejora por mucho que aumentemos el número de los mismos. Esto se debe a que, por muchos threads con los que trabajemos, lo único que paralelizaremos serán más procesos de sincronización y scheduling.





En estas 2 gráficas podemos ver la evolución del tiempo de ejecución y del speedup según aumentamos el número de threads. Una vez más, como veíamos en la anterior tabla, se aprecia en ambas gráficas un codo llegados a los 4 threads, donde el tiempo deja de bajar y el speedup deja de aumentar, convirtiéndose en una línea casi horizontal.

Si calculamos el Speedup real con 8 threads, obtenemos 1,79, mientras que el ideal sería de 3.495. Por lo tanto, solo conseguimos la mitad de lo que sería el speedup ideal con 8 procesadores.

3.2.2. Improving ϕ

Como hemos podido visualizar en el apartado anterior, la función que más problemas estaba causando a la hora de paralelizar era init_complex_grid. Por eso, hemos descomentado los pragmas parallel, single y el taskloop del bucle más interno para intentar paralelizarla (como se muestra en la siguiente imagen).

```
void init_complex_grid(fftwf_complex in_fftw[][N][N]) {

  #pragma omp parallel
  #pragma omp single
  //#pragma omp taskloop
  for (int k = 0; k < N; k++)
    #pragma omp taskloop firstprivate(k)
    for (int j = 0; j < N; j++)
      for (int 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
      }
}
```

Overview of whole program execution metrics:

Number of processors	1	2	4	6	8	10	12
Elapsed time (sec)	2.30	1.41	0.82	0.73	0.69	0.71	0.72
Speedup	1.00	1.63	2.79	3.15	3.33	3.26	3.17
Efficiency	1.00	0.82	0.70	0.52	0.42	0.33	0.26

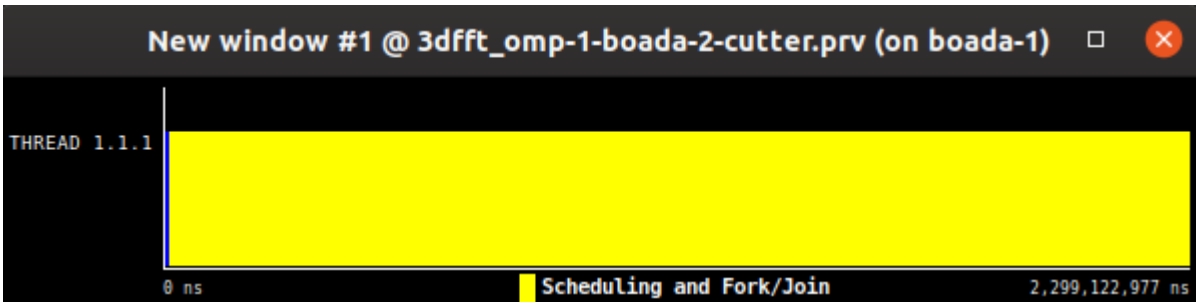
Overview of the Efficiency metrics in parallel fraction:

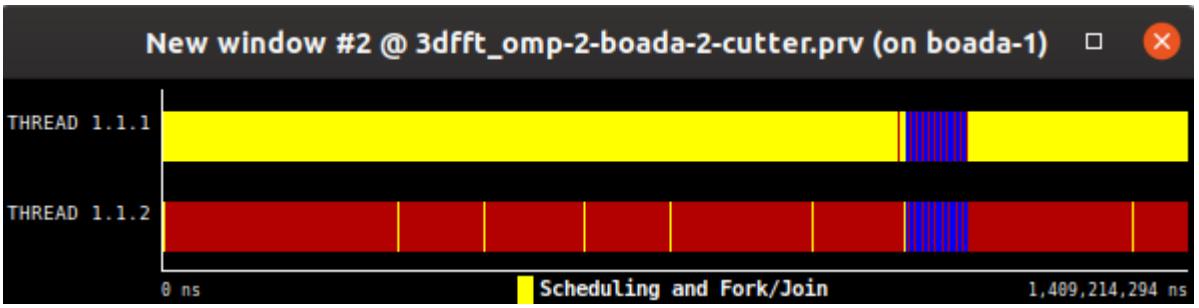
Number of processors	1	2	4	6	8	10	12
Parallel fraction	99.55%						
Global efficiency	96.63%	78.50%	67.10%	50.49%	40.01%	31.39%	25.45%
-- Parallelization strategy efficiency	96.63%	91.25%	84.97%	82.33%	80.72%	79.66%	78.17%
-- Load balancing	100.00%	98.84%	98.58%	97.77%	98.21%	98.03%	98.08%
-- In execution efficiency	96.63%	92.32%	86.20%	84.21%	82.19%	81.26%	79.70%
-- Scalability for computation tasks	100.00%	86.03%	78.97%	61.33%	49.57%	39.41%	32.56%
-- IPC scalability	100.00%	97.16%	89.10%	79.63%	69.16%	63.30%	56.66%
-- Instruction scalability	100.00%	99.59%	98.81%	98.04%	97.29%	96.54%	95.81%
-- Frequency scalability	100.00%	88.92%	89.70%	78.56%	73.67%	64.49%	59.97%

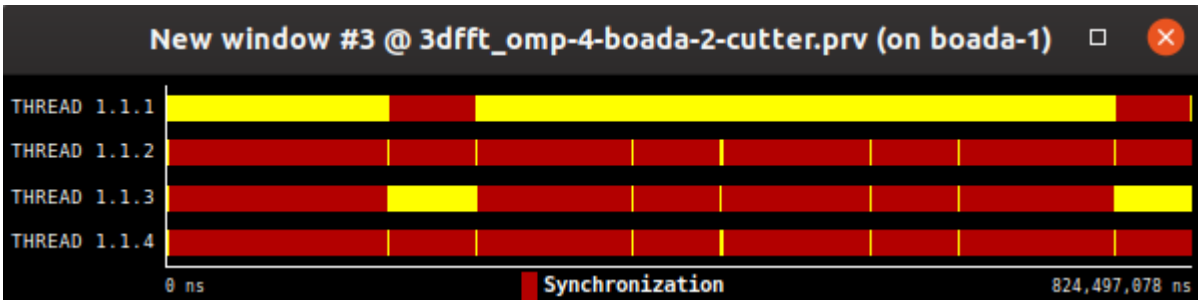
Statistics about explicit tasks in parallel fraction

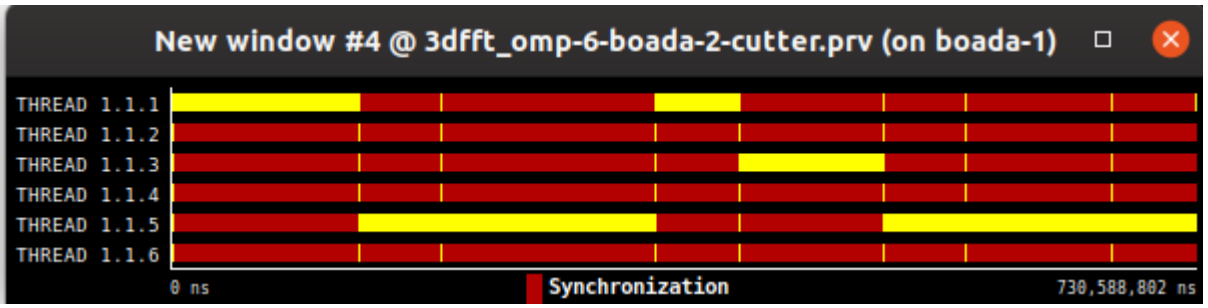
Number of processors	1	2	4	6	8	10	12
Number of explicit tasks executed (total)	20480.0	40960.0	81920.0	122880.0	163840.0	204800.0	245760.0
LB (number of explicit tasks executed)	1.0	1.0	0.97	0.97	0.97	0.94	0.98
LB (time executing explicit tasks)	1.0	0.99	0.98	0.98	0.99	0.98	0.98
Time per explicit task (average)	107.62	62.57	34.1	29.28	27.18	27.36	27.61
Overhead per explicit task (synch %)	0.42	8.47	16.18	19.82	22.06	23.61	25.83
Overhead per explicit task (sched %)	3.07	1.14	1.53	1.66	1.83	1.94	2.1
Number of taskwait/taskgroup (total)	2048.0	2048.0	2048.0	2048.0	2048.0	2048.0	2048.0

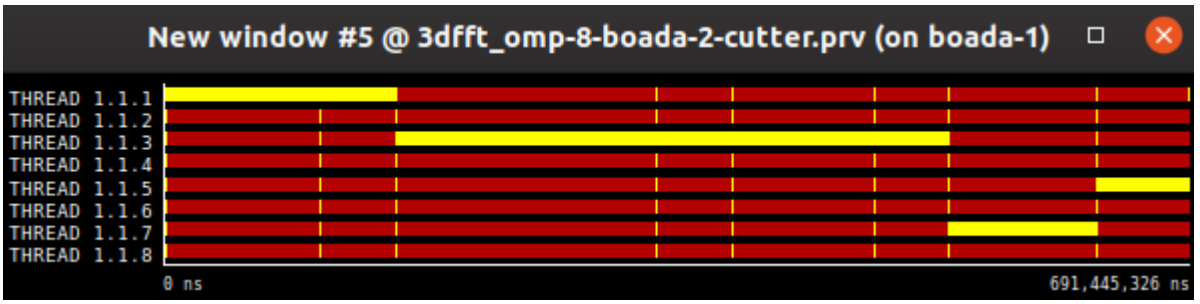
Una vez habilitado el código para ejecutar paralelamente el código dentro de la función **init_complex_grid**, y ejecutar el fichero con el mismo script, obtendremos esta tabla con los datos de ejecución del programa. Esta vez, vemos que la parte paralelizable ha aumentado, ya que ha pasado de un 71.39% a un 99.55% y que, por consecuencia, el speedup aumenta considerablemente esta vez hasta los 8 threads, una vez llegado, podemos notar que el speedup va decreciendo, mostrando que cuantos más threads por encima de ese número, más perjudicamos a la ejecución del programa. Por último, la eficiencia del programa cae con menor intensidad que en la versión anterior del programa.

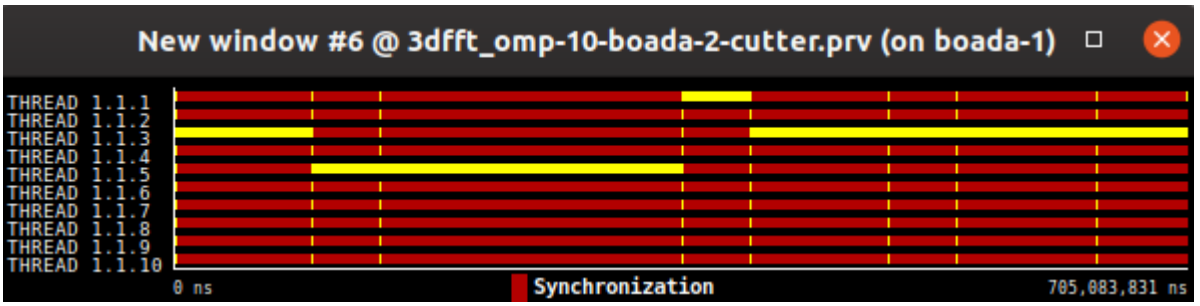

New window #1 @ 3dfft_omp-1-boada-2-cutter.prv (on boada-1)
THREAD 1.1.1
0 ns
Scheduling and Fork/Join
2,299,122,977 ns

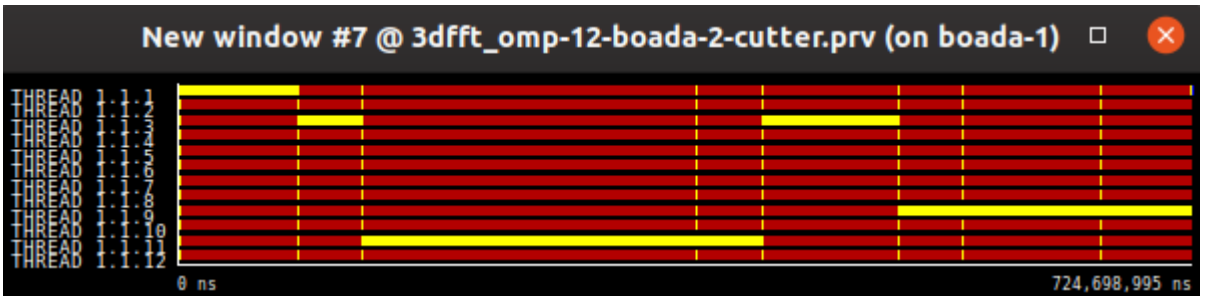

New window #2 @ 3dfft_omp-2-boada-2-cutter.prv (on boada-1)
THREAD 1.1.1
THREAD 1.1.2
0 ns
Scheduling and Fork/Join
1,409,214,294 ns


New window #3 @ 3dfft_omp-4-boada-2-cutter.prv (on boada-1)
THREAD 1.1.1
THREAD 1.1.2
THREAD 1.1.3
THREAD 1.1.4
0 ns
Synchronization
824,497,078 ns

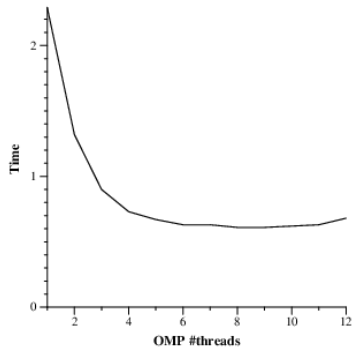

New window #4 @ 3dfft_omp-6-boada-2-cutter.prv (on boada-1)
THREAD 1.1.1
THREAD 1.1.2
THREAD 1.1.3
THREAD 1.1.4
THREAD 1.1.5
THREAD 1.1.6
0 ns
Synchronization
730,588,802 ns


New window #5 @ 3dfft_omp-8-boada-2-cutter.prv (on boada-1)
THREAD 1.1.1
THREAD 1.1.2
THREAD 1.1.3
THREAD 1.1.4
THREAD 1.1.5
THREAD 1.1.6
THREAD 1.1.7
THREAD 1.1.8
0 ns
691,445,326 ns

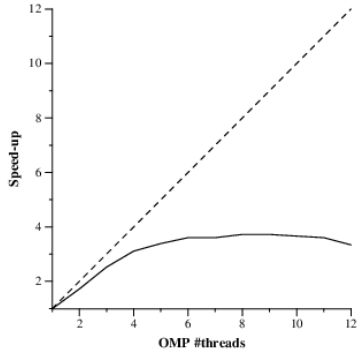

New window #6 @ 3dfft_omp-10-boada-2-cutter.prv (on boada-1)
THREAD 1.1.1
THREAD 1.1.2
THREAD 1.1.3
THREAD 1.1.4
THREAD 1.1.5
THREAD 1.1.6
THREAD 1.1.7
THREAD 1.1.8
THREAD 1.1.9
THREAD 1.1.10
0 ns
Synchronization
705,083,831 ns


New window #7 @ 3dfft_omp-12-boada-2-cutter.prv (on boada-1)
THREAD 1.1.1
THREAD 1.1.2
THREAD 1.1.3
THREAD 1.1.4
THREAD 1.1.5
THREAD 1.1.6
THREAD 1.1.7
THREAD 1.1.8
THREAD 1.1.9
THREAD 1.1.10
THREAD 1.1.11
THREAD 1.1.12
0 ns
724,698,995 ns

Las anteriores figuras nos muestran los gráficos de la ejecución del fichero con diferentes números de threads. En dichas figuras vemos que, llegados a los 8 threads, el tiempo de ejecución crece, debido a que tenemos que dedicar más tiempo de ejecución a los overheads, haciendo que tener más de 8 threads provoque un peor resultado en el tiempo de ejecución de nuestro programa.



par1304
Min elapsed execution time
Thu Mar 10 09:57:23 CET 2022



par1304
Speed-up wrt sequential time
Thu Mar 10 09:57:23 CET 2022

Gracias a estas gráficas, podemos apreciar visualmente la evolución tanto del tiempo de ejecución del programa, como del speedup según el número de threads. Tal y como hemos observado en las anteriores figuras de manera numérica, podemos apreciar que a partir de los 8 threads, el tiempo de ejecución del programa empieza a crecer exponencialmente, a la vez que el speedup empieza a decrecer de la misma manera. Concluyendo así que el número óptimo de threads para el programa es 8.

3.2.3. Reducing parallelisation overheads

Para cerrar la sesión 3 del primer laboratorio, probaremos a incrementar la granularidad de las tareas comentando los taskloop más internos en cada región paralela, y descomentando los más externos.

```
void init_complex_grid(fftwf_complex in_fftw[][N][N]) {

  #pragma omp parallel
  #pragma omp single
  #pragma omp taskloop
  for (int k = 0; k < N; k++)
    //#pragma omp taskloop firstprivate(k)
    for (int j = 0; j < N; j++)
      for (int 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 transpose_xy_planes(fftwf_complex  tmp_fftw[][N][N], fftwf_complex in_fftw[][N][N]) {

  #pragma omp parallel
  #pragma omp single
  #pragma omp taskloop
  for (int k=0; k<N; k++)
    //#pragma omp taskloop firstprivate(k)
    for (int j=0; j<N; j++)
      for (int 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]) {

  #pragma omp parallel
  #pragma omp single
  #pragma omp taskloop
  for (int k=0; k<N; k++)
    //#pragma omp taskloop firstprivate(k)
    for (int j=0; j<N; j++)
      for (int 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];
      }
}

void ffts1_planes(fftwf_plan p1d, fftwf_complex in_fftw[][N][N]) {

  #pragma omp parallel
  #pragma omp single
  #pragma omp taskloop
  for (int k=0; k<N; k++)
    //#pragma omp taskloop firstprivate(k)
    for (int j=0; j<N; j++)
      fftwf_execute_dft( p1d, (fftwf_complex *)in_fftw[k][j][0], (fftwf_complex *)in_fftw[k][j][0]);
}
```

Como resultado, vemos que tanto el speedup como la eficiencia han mejorado considerablemente respecto a las versiones anteriores. Además, también vemos en la tabla

que la eficiencia de las regiones paralelas del programa consiguen un desempeño apropiado.

Overview of whole program execution metrics:

Number of processors	1	2	4	6	8	10	12
Elapsed time (sec)	2.27	1.25	0.61	0.46	0.38	0.34	0.32
Speedup	1.00	1.82	3.72	4.91	6.05	6.60	7.22
Efficiency	1.00	0.91	0.93	0.82	0.76	0.66	0.60

Overview of the Efficiency metrics in parallel fraction:

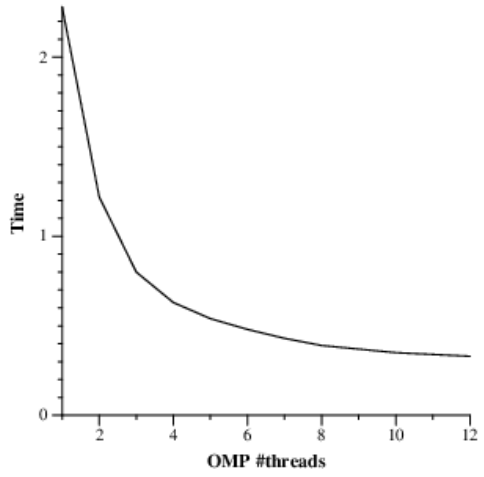
Number of processors	1	2	4	6	8	10	12
Parallel fraction	99.61%						
Global efficiency	99.90%	90.44%	92.62%	81.50%	75.32%	65.73%	59.98%
-- Parallelization strategy efficiency	99.90%	98.52%	97.17%	97.69%	97.66%	96.91%	96.47%
-- Load balancing	100.00%	98.87%	97.74%	98.19%	98.44%	97.44%	97.03%
-- In execution efficiency	99.90%	99.65%	99.41%	99.49%	99.21%	99.46%	99.42%
-- Scalability for computation tasks	100.00%	91.79%	95.31%	83.42%	77.13%	67.82%	62.17%
-- IPC scalability	100.00%	100.00%	96.64%	88.35%	82.41%	75.42%	69.57%
-- Frequency scalability	100.00%	91.79%	98.63%	94.44%	93.60%	89.94%	89.39%

Statistics about explicit tasks in parallel fraction

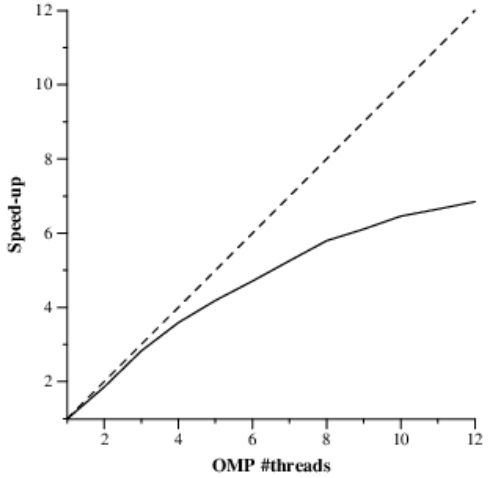
Number of processors	1	2	4	6	8	10	12
Number of explicit tasks executed (total)	80.0	160.0	320.0	480.0	640.0	800.0	960.0
LB (number of explicit tasks executed)	1.0	1.0	0.94	0.96	0.96	0.84	0.94
LB (time executing explicit tasks)	1.0	0.99	0.98	0.99	0.99	0.99	0.98
Time per explicit task (average)	28288.92	15408.48	7419.37	5650.88	4583.96	4170.3	3790.8
Overhead per explicit task (synch %)	0.01	1.47	2.86	2.31	2.33	3.1	3.57
Overhead per explicit task (sched %)	0.09	0.02	0.02	0.03	0.03	0.03	0.04
Number of taskwait/taskgroup (total)	8.0	8.0	8.0	8.0	8.0	8.0	8.0

Una vez configurado el programa como hemos explicado anteriormente, obtendremos esta tabla de datos sobre la ejecución del programa. En esta versión, podemos comprobar que el speedup aumenta constantemente según aumenta el número de threads, sin llegar a ningún codo ni ningún punto de inflexión. Podemos notar un comportamiento parecido con la eficiencia del programa, la cual baja constantemente, pero a un ritmo bajo, por lo que, aún trabajando sobre un gran número de threads, tenemos una eficiencia mayor o igual al 60%.
Además, al aumentar la granularidad de la región paralela, el paralelismo crece un poco, llegando al 99.61% de paralelismo.

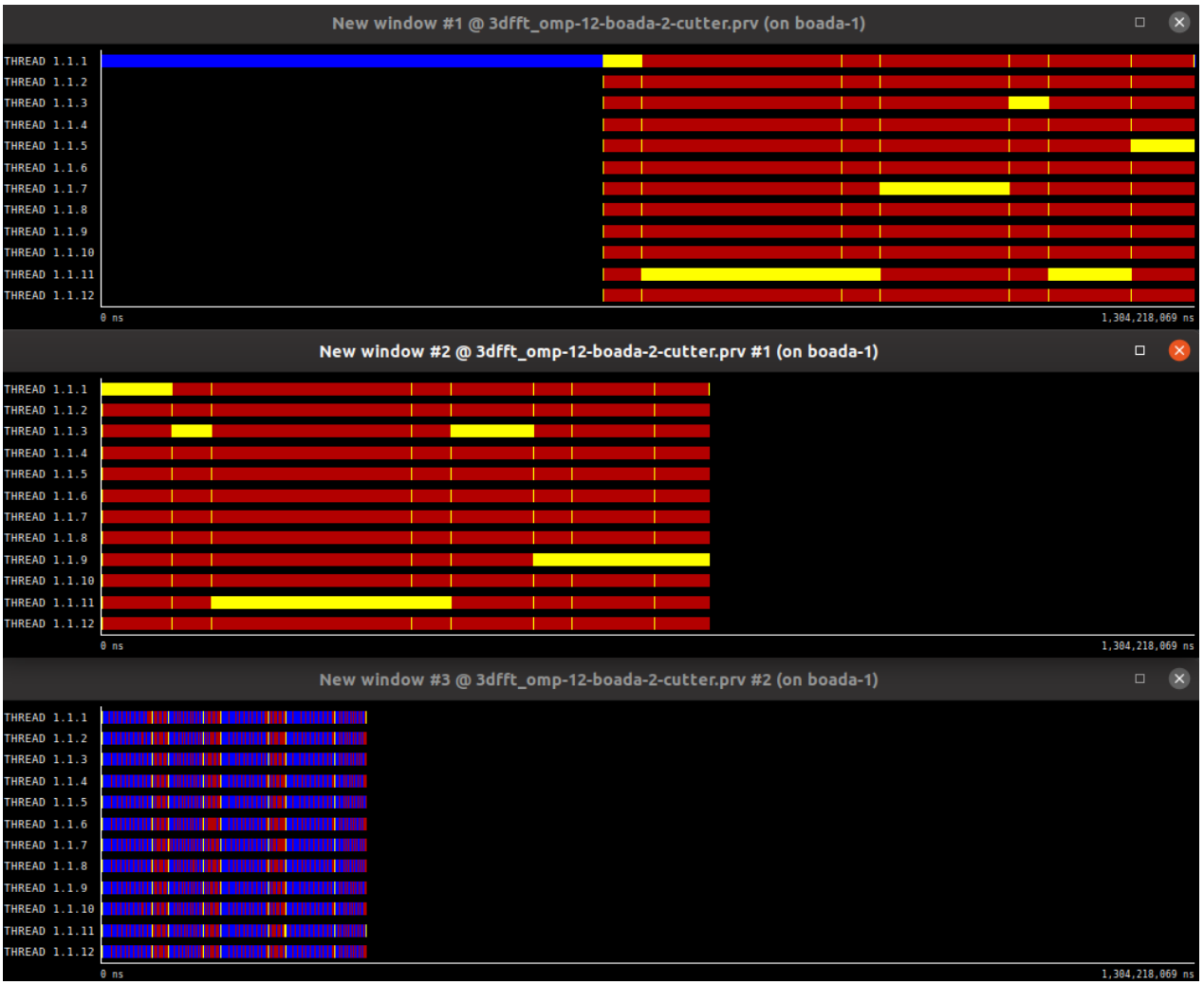
En las siguientes gráficas, podremos visualizar los datos numéricos de la tabla de manera más sencilla. Podemos observar que el tiempo de ejecución baja siempre que aumentamos el número de threads, por lo tanto, en esta versión, el problema del tiempo de ejecución de los overheads no nos perjudica al aumentar el número de threads. Por otra parte, podemos ver que el speedup aumenta considerablemente según aumenta el número de procesadores, comprobando así el mismo hecho, que gracias a que no nos vemos afectados por los tiempos de ejecución de los overheads, no nos perjudica aumentar el número de threads.



par1304
Min elapsed execution time
Thu Mar 10 10:11:40 CET 2022



par1304
Speed-up wrt sequential time
Thu Mar 10 10:11:40 CET 2022



En esta figura podemos observar gráficamente el comportamiento de las tres versiones del programa utilizando la misma escala de tiempo. Vemos como el principal problema de la versión 1 (función init_complex_grid) se soluciona en la siguiente versión. En ésta, se puede observar claramente el principal problema (demasiados overheads y sincronizaciones) y como se soluciona en la versión tres, donde los threads pasan de estar prácticamente todo el rato sincronizandose a hacer trabajo.

Ésta última parte se puede apreciar mejor si miramos las siguientes imágenes:

	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	78.14 %	19.80 %	2.06 %
THREAD 1.1.2	79.13 %	20.86 %	0.00 %
THREAD 1.1.3	76.29 %	18.42 %	5.29 %
THREAD 1.1.4	79.04 %	20.95 %	0.00 %
THREAD 1.1.5	80.37 %	19.63 %	0.00 %
THREAD 1.1.6	79.86 %	20.13 %	0.00 %
THREAD 1.1.7	80.33 %	19.66 %	0.00 %
THREAD 1.1.8	79.74 %	20.25 %	0.00 %
THREAD 1.1.9	73.15 %	16.69 %	10.15 %
THREAD 1.1.10	79.77 %	20.23 %	0.00 %
THREAD 1.1.11	76.79 %	17.74 %	5.47 %
THREAD 1.1.12	79.82 %	20.18 %	0.00 %
Total	942.45 %	234.56 %	22.99 %
Average	78.54 %	19.55 %	1.92 %
Maximum	80.37 %	20.95 %	10.15 %
Minimum	73.15 %	16.69 %	0.00 %
StDev	2.06 %	1.23 %	3.18 %
Avg/Max	0.98	0.93	0.19

	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	94.20 %	5.67 %	0.12 %
THREAD 1.1.2	95.45 %	4.54 %	0.01 %
THREAD 1.1.3	97.88 %	2.12 %	0.01 %
THREAD 1.1.4	94.97 %	5.02 %	0.01 %
THREAD 1.1.5	98.86 %	1.13 %	0.01 %
THREAD 1.1.6	94.86 %	5.14 %	0.01 %
THREAD 1.1.7	98.84 %	1.06 %	0.10 %
THREAD 1.1.8	95.36 %	4.63 %	0.01 %
THREAD 1.1.9	98.05 %	1.94 %	0.01 %
THREAD 1.1.10	96.45 %	3.54 %	0.01 %
THREAD 1.1.11	98.59 %	1.24 %	0.17 %
THREAD 1.1.12	94.73 %	5.26 %	0.01 %
Total	1,158.24 %	41.29 %	0.47 %
Average	96.52 %	3.44 %	0.04 %
Maximum	98.86 %	5.67 %	0.17 %
Minimum	94.20 %	1.06 %	0.01 %
StDev	1.72 %	1.74 %	0.05 %
Avg/Max	0.98	0.61	0.23

En la primera imagen, podemos observar el trabajo realizado por los threads de la versión con la mejora de phi, mientras que la segunda nos muestra el trabajo de los threads para la última versión. Podemos ver claramente que en la última versión los threads ejecutan mucho más porcentaje de trabajo ya que se tienen que invertir menos tiempo realizando sincronizaciones y scheduling and fork joins. Por lo tanto, hemos conseguido el objetivo que queríamos con la última versión.

En resumen, podemos extraer esta tabla:

Version	ϕ	Ideal S_8	T_1	T_8	Real S_8
Initial Version of 3dfft_omp.c	71,39	3.495	2,26	1,26	1,79
New version with improved ϕ	99,55	222.2	2,3	0,69	3,33
Final version with reduced parallelisation overheads	99,61	256.41	2,27	0,38	6,05

Podemos ver la fracción de programa paralelizable, el speedup ideal, speedup real y tiempos de ejecución de las diferentes versiones del programa con 1 y 8 threads. Podemos ver que aunque de la versión 2 a la 3, la fracción paralelizable casi no aumenta, el speedup real es prácticamente el doble, debido a que al poseer una granularidad mucho mayor, evita el problema de tiempo de los overheads.

4. CONCLUSIONES

En este primer laboratorio hemos visto que la arquitectura sobre la que estamos trabajando importa mucho en la ejecución de un programa. Además hemos aprendido diferentes maneras de medir el rendimiento de un programa.

En la segunda sesión, hemos trabajado sobre la estructura de un mismo programa, y como la posición de las definiciones de tareas puede afectar significativamente al mismo, ya que reduce o incrementa la granularidad, teniendo un impacto directo sobre la ejecución de este.

Por último, en la última sesión, hemos trabajado diversos pequeños cambios que se pueden aplicar al código que, aun siendo pequeños, provocan considerables mejoras en speedup y escalabilidad.

En general, hemos aprendido a usar las herramientas que usaremos de aquí en adelante en el laboratorio de la asignatura, y la importancia que tiene la correcta paralelización de los programas.