UNIVERSITAT POLITÈCNICA DE CATALUNYA

Paralelismo

Laboratorio 1: Experimental setup

Haopeng Lin Ye

Joan Sales de Marcos

ÍNDICE

Sesión 1 - Experimental setup

1.1 - Node architecture and memory
1.2 - Serial compilation and execution
1.3 - Compilation and execution of OpenMP programs
1.4 - Strong vs. Weak scalability

Sesión 2 - Systematically analyzing task decompositions with Tareador

2.1 - Exploring new task decompositions for 3DFFT
    2.1.1 - Versión original
    2.1.2 - Versión 1
    2.1.3 - Versión 2
    2.1.4 - Versión 3
    2.1.5 - Versión 4
    2.1.6 - Versión 5
2.2 - Análisis de los resultados

Sesión 3 - Understanding the execution of OpenMP programs

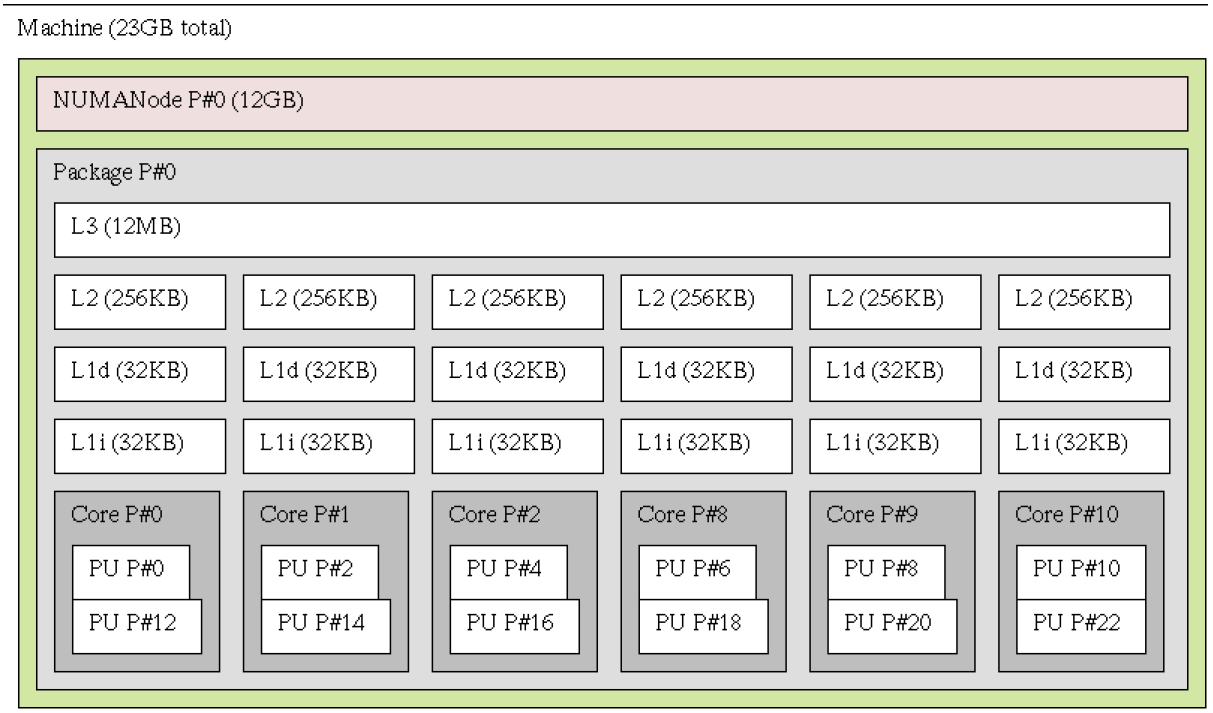
3.1 - Versión inicial
3.2 - Improving Φ
3.3 - Reducing parallelisation overheads
3.4 - Conclusión

Sesión 1: Experimental setup

El objetivo de esta primera sesión es familiarizarnos con el entorno que trabajaremos durante este curso. En nuestro setup trabajaremos con un servidor multiprocesador llamado Boada, que es un cluster dividido en 8 nodos con diferentes arquitecturas. Para ejecutar nuestros programas , lo haremos interactivamente (boada-1) o con el sistema de colas (boada-2 a boada-4).

1.1 - Node architecture and memory

Primero visualizamos la arquitectura de nodos con los comandos lscpu(en forma de texto) i lstopo (la figura 1.1 se obtiene añadiendo el flag –of fig).
La arquitectura de los nodos boada-1 hasta boada-4 és exactamente igual.



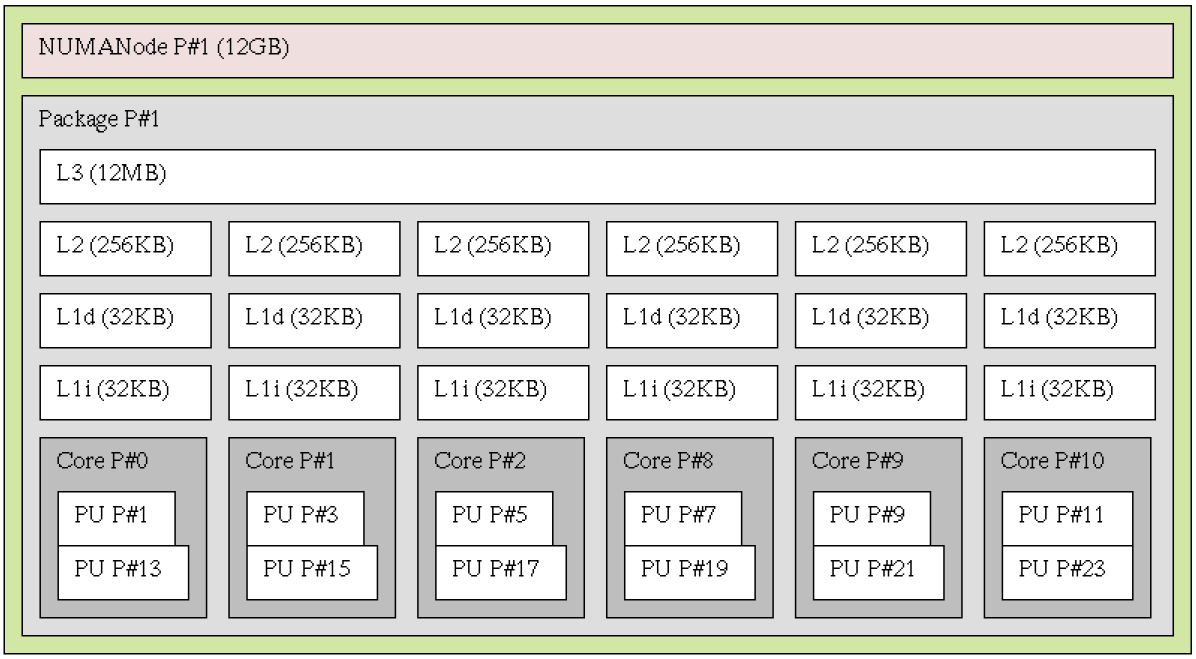


Figura 1.1:Arquitectura de memoria de boada-1 a boada-4. (Boada-1)

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

1.2 - Serial compilation and execution

En esta parte probaremos a ejecutar el código pi_seq.c con los diferentes módulos de boada. Para ello, primero ejecutaremos el código interactivamente y después lo pondremos en la cola de ejecución de boada-2 y boada-3.

Los tiempos de ejecución obtenidos son los siguientes:

Ejecución	T_{exe}	% CPU
interactiva (boada-1)	3,94 s	99
boada-2	3,96 s	99
boada-3	3,96 s	99

Teóricamente las ejecuciones en boada-2 y boada-3 deberían haber sido más rápidas al ser ejecuciones aisladas (a diferencia de boada-1 donde todos los grupos interactúan simultáneamente), sin embargo han sido aproximadamente 0,02s más lentas, suponemos que por factores concretos del hardware, como pueden ser el estado de la caché, el calor de la máquina o el estado puntual de la CPU en ese momento.

En las siguientes figuras se puede observar de dónde hemos sacado los datos:

```
par2311@boada-1:~/lab1/pi$ ./run-seq.sh pi_seq 1000000000
Number pi after 1000000000 iterations = 3.141592653589768
Execution time (secs.): 3.940106
3.93user 0.00system 0:03.94elapsed 99%CPU (0avgtext+0avgdata 2192maxresident)k
0inputs+0outputs (0major+85minor)pagefaults 0swaps
```

```
par2311@boada-1:~/lab1/pi$ cat time-pi_seq-boada-2
3.94user 0.00system 0:03.96elapsed 99%CPU (0avgtext+0avgdata 2172maxresident)k
0inputs+8outputs (0major+86minor)pagefaults 0swaps
par2311@boada-1:~/lab1/pi$ cat time-pi_seq-boada-3
3.94user 0.00system 0:03.96elapsed 99%CPU (0avgtext+0avgdata 2216maxresident)k
72inputs+8outputs (1major+89minor)pagefaults 0swaps
```

Figuras 1.2 y 1.3: Terminal con los datos de las ejecuciones de pi_seq

1.3 - Compilation and execution of OpenMP programs

T_{user} y T_{system} son dos valores derivados del tiempo de ejecución que suelen ser más realistas ya que tienen en cuenta otras variables:

T_{user}: es la cantidad de tiempo de CPU empleado en el código de modo de usuario (fuera del kernel) dentro del proceso. Este es solo el tiempo de CPU real utilizado para ejecutar el proceso. Otros procesos y el tiempo que el proceso pasa bloqueado no cuentan para esta cifra.

T_{system}: es la cantidad de tiempo de CPU invertido en el kernel dentro del proceso. Esto significa el T_{exe} de CPU dedicado a las llamadas al sistema dentro del núcleo, a diferencia del código de la biblioteca, que aún se ejecuta en el espacio del usuario. Al igual que T_{user}, este es solo el tiempo de CPU utilizado por el proceso.

Después de ejecutar pi_omp con 1.000.000.000 iteraciones y 1, 2, 4, y 8 threads hemos podido apreciar que lo que hace el programa es básicamente calcular el número pi hasta unos cuantos decimales. La diferencia es la información adicional que se puede ver en la terminal: El tiempo de ejecución y el porcentaje de CPU usado en el mismo.

Como se puede observar en la figura 1.4 (más abajo), el tiempo de ejecución con los distintos threads no es del todo diferente, mientras que el consumo de CPU con más de 1 thread es significativamente superior a la ejecución con uno solo.

Ejecución interactiva

nº de threads	T_{user}	T_{system}	T_{exe}	% CPU
1	5,55 s	0,00 s	5,557784 s	99
2	11,28 s	0,00 s	5,640298 s	199
4	11,28 s	0,02 s	5,656298 s	199
8	11,16 s	0,05 s	5,609682 s	199

```
par2311@boada-1:~/lab1/pi$ ./run-omp.sh pi_omp 10000000000 1
Number pi after 1410065408 iterations = 3.141592653589974
Execution time (secs.): 5.557784
5.55user 0.00system 0:05.56elapsed 99%CPU (0avgtext+0avgdata 4084maxresident)k
0inputs+0outputs (0major+248minor)pagefaults 0swaps
par2311@boada-1:~/lab1/pi$ ./run-omp.sh pi_omp 10000000000 2
Number pi after 1410065408 iterations = 3.141592653589820
Execution time (secs.): 5.640298
11.28user 0.00system 0:05.64elapsed 199%CPU (0avgtext+0avgdata 4284maxresident)k
0inputs+0outputs (0major+351minor)pagefaults 0swaps
par2311@boada-1:~/lab1/pi$ ./run-omp.sh pi_omp 10000000000 4
Number pi after 1410065408 iterations = 3.141592653589799
Execution time (secs.): 5.656298
11.28user 0.02system 0:05.66elapsed 199%CPU (0avgtext+0avgdata 4316maxresident)k
0inputs+0outputs (0major+386minor)pagefaults 0swaps
par2311@boada-1:~/lab1/pi$ ./run-omp.sh pi_omp 10000000000 8
Number pi after 1410065408 iterations = 3.141592653589825
Execution time (secs.): 5.609682
11.16user 0.05system 0:05.61elapsed 199%CPU (0avgtext+0avgdata 4456maxresident)k
0inputs+0outputs (0major+475minor)pagefaults 0swaps
```

Figura 1.4: Terminal con los datos de las diferentes ejecuciones de pi_omp (Ejecución interactiva)

La conclusión que sacamos de los resultados es que este código al ser puesto a prueba secuencialmente no se aprovecha del todo el hecho de utilizar diferentes threads. Además de que eso parece ser malo ya que se está usando el doble de CPU para tardar prácticamente el mismo tiempo.

Usando el comando cat para ver el archivo de texto que nos da la ejecución en cola de pi_omp la terminal nos muestra el tiempo que ha tardado en finalizar el programa y el porcentaje de CPU por tiempo usado en éstas (Figura 1.5, más abajo). Los mismos datos que nos daba la ejecución interactiva pero en un documento a parte.

Se puede apreciar un cambio significativo en los resultados.

Ejecución en cola

nº de threads	**T_{user}**	**T_{system}**	**T_{exe}**	**% CPU**
1	5,55 s	0,00 s	5,58 s	99
2	5,57 s	0,00 s	2,80 s	198
4	5,62 s	0,00 s	1,42 s	394
8	5,93 s	0,01 s	0,76 s	775

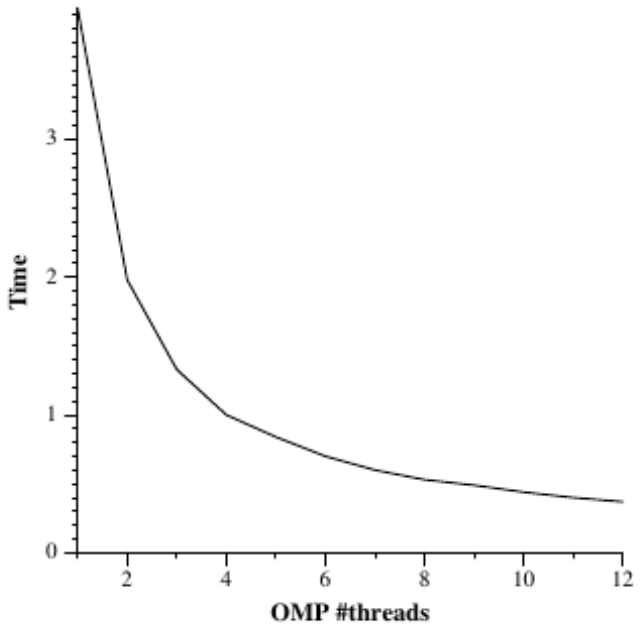
```
par2311@boada-1:~/lab1/pi$ cat time-pi_omp-*
5.55user 0.00system 0:05.58elapsed 99%CPU (0avgtext+0avgdata 4196maxresident)k
0inputs+8outputs (0major+251minor)pagefaults 0swaps
5.57user 0.00system 0:02.80elapsed 198%CPU (0avgtext+0avgdata 4264maxresident)k
0inputs+8outputs (0major+360minor)pagefaults 0swaps
5.62user 0.00system 0:01.42elapsed 394%CPU (0avgtext+0avgdata 4324maxresident)k
0inputs+8outputs (0major+409minor)pagefaults 0swaps
5.93user 0.01system 0:00.76elapsed 775%CPU (0avgtext+0avgdata 4476maxresident)k
0inputs+8outputs (0major+327minor)pagefaults 0swaps
```

Figura 1.5: Terminal con los datos de las diferentes ejecuciones de pi_omp (Ejecución en cola)

Claramente la ejecución en cola es mucho más rápida cuantos más threads se usen. En concreto, el speedup de 8 threads respecto a 1 es de 5,58 / 0,76 = 7,34 veces más rápido. Esto es una buena señal de que el programa es paralelizable.

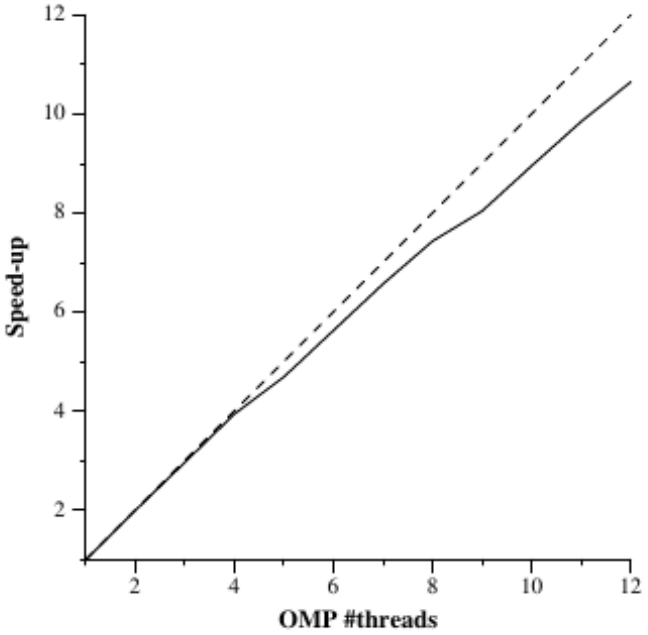
1.4 - Strong vs. weak scalability

En esta sección se explorará la escalabilidad del código pi_omp.c cuando el número de threads varia. Primero analizaremos el código de strong-omp para la escalabilidad fuerte



par2311
Min elapsed execution time
Fri Feb 18 13:54:50 CET 2022

Como se puede ver en el gráfico de arriba, el tiempo de ejecución del programa según los threads disminuye siguiendo una curvatura, esto nos dice que a partir de cierto punto no es demasiado recomendable seguir aumentando el número de threads. Por lo que nos muestra la imagen, calculamos que este valor es aproximadamente entre 4 o 6.



par2311
Speed-up wrt sequential time
Fri Feb 18 13:54:50 CET 2022

También se puede observar en la gráfica de abajo un decremento en la pendiente del speedup a medida que los threads aumentan, como se podría haber deducido de la primera gráfica. Esto significa que a partir de un punto la ganancia de tiempo al añadir threads será muy pequeña.

Como se puede intuir, si se sigue aumentando el número de threads hasta 24, en el gráfico de arriba habrá una “colilla” muy plana después de la curva ya que por el comportamiento de la misma se puede seguir la trayectoria. Lo mismo pasa con el speedup, la pendiente cada vez será más plana e incluso llegará un punto donde deje de aumentar.

En la siguiente figura se muestra la eficiencia paralela a medida que el número de threads aumenta. Se puede apreciar cómo claramente no le está haciendo un favor al programa ya que el gráfico está en pendiente hacia abajo.

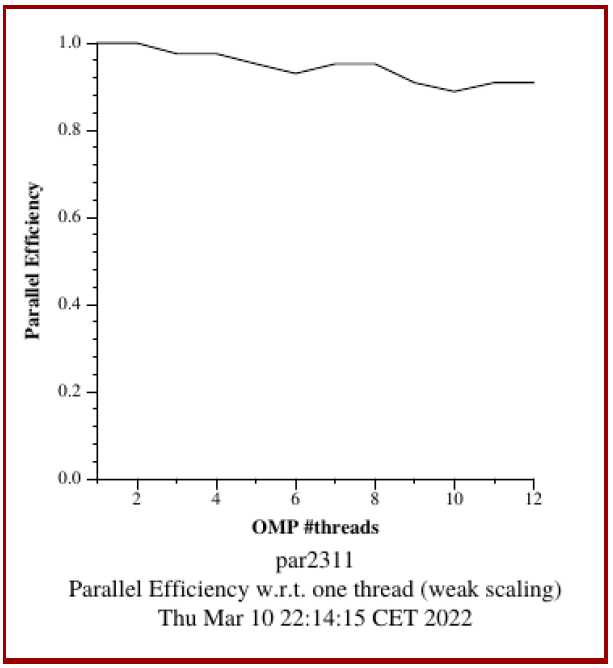


Figura 1.6: Gráfico de la eficiencia del programa a medida que aumentan los threads

Sesión 2: Systematically analyzing task decompositions with Tareador

El objetivo de esta sesión es aprender los conceptos, comandos y herramientas básicas de *tareador*, un software para analizar el paralelismo de un programa, además de usarlo para hacer lo ya mencionado con diferentes versiones de un mismo código y así averiguar de qué modificación es más paralelizable.

2.1 - Exploring new task decompositions for 3DFFT

Para esta sección de laboratorio trabajaremos con un código llamado 3dfft_tar.c y le aplicaremos diferentes cambios para ver hasta qué punto se puede paralelizar.

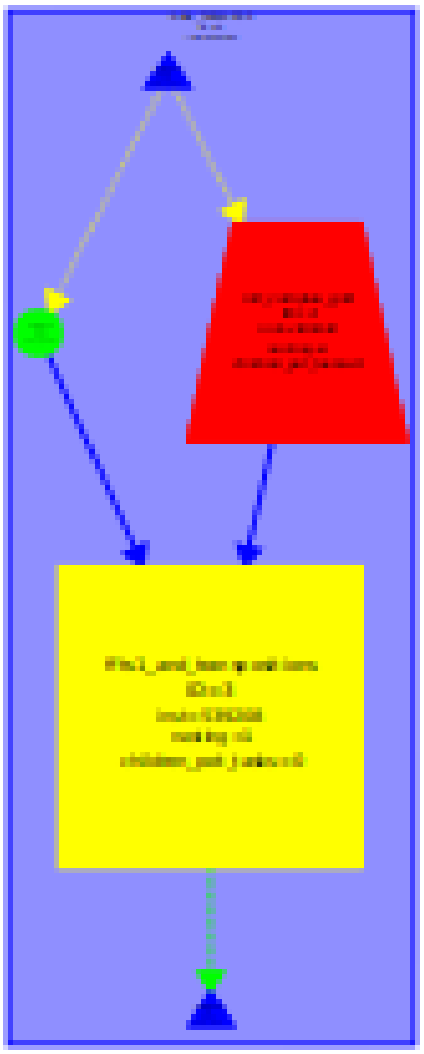
2.1.1 - Versión original

En la versión original del código, solo hay un par de tareas diferenciadas por el tareador, así que el paralelismo no es del todo deseable

En la figura 2.1 se puede observar el TDG que nos da tareador y en la 2.2 el código de dónde proviene (Figuras más abajo).

Para calcular el paralelismo, necesitamos el tiempo que tarda la ejecución del programa secuencialmente, lo que tardaría con recursos infinitos y el número de instrucciones del programa, cosas que nos da el propio tareador y la terminal al ejecutar el programa. Los datos de todas las versiones se encuentran en la sección 2.2.

Como todas las versiones del programa están siendo ejecutadas en el mismo nodo de boada, el tiempo secuencial para calcular el paralelismo será igual en las futuras versiones de 3dfft_tar.



```
/* Initialize Tareador analysis */
tareador ON ();
START_COUNT_TIME;

tareador_start_task("start_plan_forward");
start_plan_forward(in_fftw, &p1d);
tareador_end_task("start_plan_forward");

STOP_COUNT_TIME("3D FFT Plan Generation");

START_COUNT_TIME;

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

STOP_COUNT_TIME("Init Complex Grid FFT3D");

START_COUNT_TIME;

tareador_start_task("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");
```

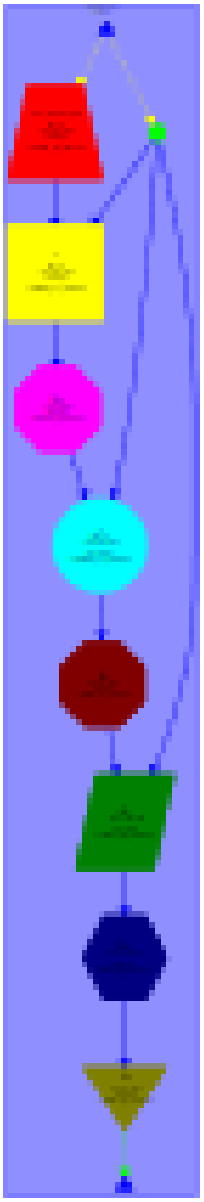
Figuras 2.1 y 2.2: TDG y código enfocado a tareador de la v0 de 3dfft_tar.c

2.1.2 - Versión 1

Para esta versión se pedía desenglobar una tarea del tareador en muchas más pequeñas para cada línea de código dentro de la misma, así se puede aprovechar un poco más la posible independencia entre líneas de código.

Como bien se ha explicado en la versión anterior, el tiempo secuencial para calcular el paralelismo será igual. Los datos de todas las versiones se encuentran en la sección 2.2.

A continuación están la figura 2.3 y 2.4: el Task Dependency Graph y el código de esta versión:



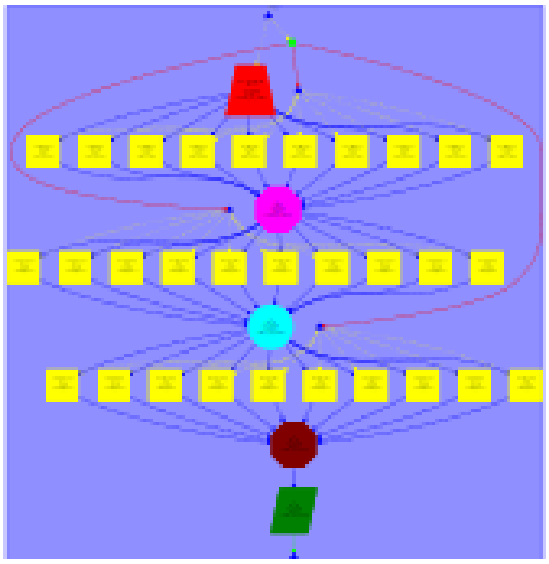
```
/* Initialize Tareador analysis */
tareador ON ();
START_COUNT_TIME;

tareador start task("start plan forward");
start_plan_forward(in_fftw, &p1d);
tareador_end_task("start_plan_forward");

STOP_COUNT_TIME("3D FFT Plan Generation");

START COUNT TIME;

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

STOP_COUNT_TIME("Init Complex Grid FFT3D");

START_COUNT_TIME;

tareador_start_task("1");

ffts1_planes(p1d, in_fftw);
tareador_end_task("1");
tareador_start_task("2");
transpose_xy_planes(tmp_fftw, in_fftw);
tareador_end_task("2");
tareador_start_task("3");
ffts1_planes(p1d, tmp_fftw);
tareador_end_task("3");
tareador_start_task("4");
transpose_zx_planes(in_fftw, tmp_fftw);
tareador_end_task("4");
tareador_start_task("5");
ffts1_planes(p1d, in_fftw);
tareador_end_task("5");
tareador_start_task("6");
transpose_zx_planes(tmp_fftw, in_fftw);
tareador_end_task("6");
tareador_start_task("7");
transpose_xy_planes(in_fftw, tmp_fftw);
tareador_end_task("7");

STOP_COUNT_TIME("Execution FFT3D");
```

Figuras 2.3 y 2.4: TDG y código enfocado a tareador de la v1 de 3dfft_tar.c

2.1.3 - Versión 2

La diferencia que se hacía en esta versión consistía en sustituir una tarea grande llamada originalmente como “ffts1_planes“ por una tarea más “pequeña” en cada iteración del bucle de dentro de la función llamándola “"ffts1_planes_loop_k”.

Esto provoca la creación de múltiples tareas potencialmente independientes y paralelizables que pueden ayudar al tiempo de ejecución del programa. Los datos de todas las versiones se encuentran en la sección 2.2.



```
/* Initialize Tareador analysis */
tareador ON ();
START_COUNT_TIME;

tareador start task("start plan forward");
start plan forward(in fftw, &p1d);
tareador end task("start plan forward");

STOP COUNT TIME("3D FFT Plan Generation");

START COUNT TIME;

tareador start task("init complex grid");
init complex grid(in fftw);
tareador end task("init complex grid");

STOP_COUNT_TIME("Init Complex Grid FFT3D");
START_COUNT_TIME;
ffts1_planes(p1d, in_fftw);
tareador_start_task("2");
transpose_xy_planes(tmp_fftw, in_fftw);
tareador_end_task("2");
ffts1_planes(p1d, tmp_fftw);
tareador_start_task("4");
transpose_zx_planes(in_fftw, tmp_fftw);
tareador_end_task("4");
ffts1_planes(p1d, in_fftw);
tareador_start_task("6");
transpose_zx_planes(tmp_fftw, in_fftw);
tareador_end_task("6");
tareador_start_task("7");
transpose_xy_planes(in_fftw, tmp_fftw);


tareador_end_task("7");

STOP_COUNT_TIME("Execution FFT3D");
```

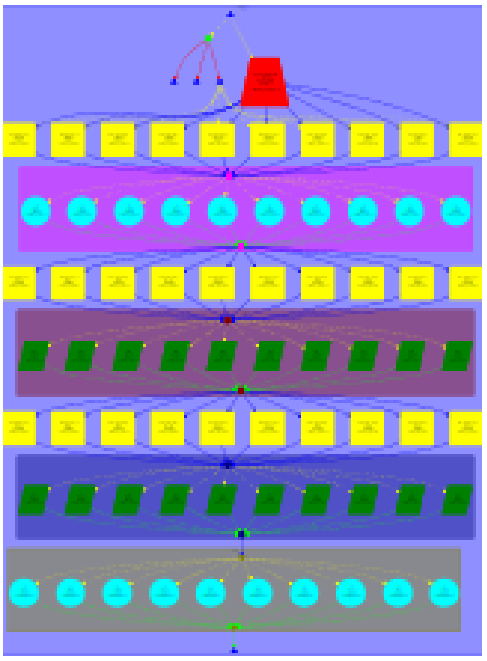
Figuras 2.5 y 2.6: TDG y código enfocado a tareador de la v2 de 3dfft_tar.c

No se adjunta la parte de dentro de la función ffts_planes porque la modificación es idéntica a la que sale en el enunciado.

2.1.4 - Versión 3

Para esta versión había que seguir con la misma dinámica que en la versión 2 pero aplicarla a más partes del código, haciendo que más fragmentos de éste se puedan paralelizar de una forma más clara, obteniendo mejores resultados.

Los datos de todas las versiones se encuentran en la sección 2.2.



```
/* Initialize Tareador analysis */
tareador ON ();
START_COUNT_TIME;

tareador start task("start plan forward");
start_plan_forward(in_fftw, &p1d);
tareador_end_task("start_plan_forward");

STOP_COUNT_TIME("3D FFT Plan Generation");

START COUNT TIME;

tareador_start_task("init_complex_grid");
init complex grid(in fftw);
tareador_end_task("init_complex_grid");

STOP COUNT TIME("Init Complex Grid FFT3D");

START_COUNT_TIME;

ffts1_planes(p1d, in_fftw);
tareador_start_task("2");
transpose_xy_planes(tmp_fftw, in_fftw);
tareador_end_task("2");
ffts1_planes(p1d, tmp_fftw);
tareador_start_task("4");
transpose_zx_planes(in_fftw, tmp_fftw);
tareador_end_task("4");
ffts1_planes(p1d, in_fftw);
tareador_start_task("6");
transpose_zx_planes(tmp_fftw, in_fftw);
tareador_end_task("6");
tareador_start_task("7");
transpose_xy_planes(in_fftw, tmp_fftw);
tareador_end_task("7");

STOP_COUNT_TIME("Execution FFT3D");
```

Figuras 2.7 y 2.8: TDG y código enfocado a tareador de la v3 de 3dfft_tar.c

La nueva modificación en los bucles que pide v3 es idéntica a la de v2 pero en otra parte del código así que consideramos innecesario adjuntarla.

2.1.5 - Versión 4

Esta versión pedía cambiar una tarea por otras más pequeñas dentro de la función init_complex_grid y simular la ejecución con 1, 2, 4, 8, 16 y 32 procesadores para saber la escalabilidad.

Primero observaremos el Grafo de Dependencias que nos da tareador para explicar por qué el código es más paralelizable.

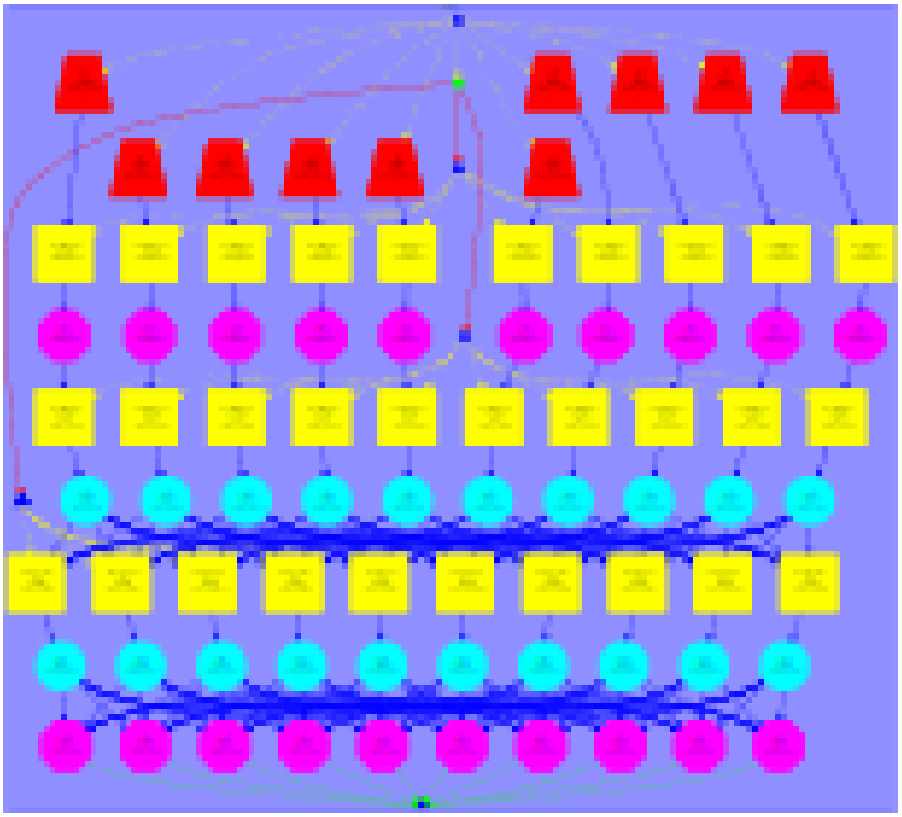


Figura 2.9: TDG de la v4 de 3dfft_tar.c

La división de tareas hecha en esta versión ha provocado que en el primer nivel del Grafo de dependencias en lugar de haber una única tarea con mucho tiempo de ejecución tenga muchas tareas independientes con tiempos de ejecución menores pudiendo ser procesadas y paralelizadas por separado, disminuyendo el tiempo general de la ejecución.

Los datos de todas las versiones se encuentran en la sección 2.2.

En la figura 2.10 se puede ver la parte del código que se modificó para esta versión.

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

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

Figura 2.10: Código enfocado a tareador de la v4 de 3dfft_tar.c

A continuación vamos a ver la escalabilidad de la versión 4 de 3dfft_tar. Para ello hemos ejecutado el programa con un número diferente de procesadores y calculado el tiempo de ejecución:

nº de procesadores	T_{exe} (ns)
1	639780001
2	320310001
4	165389001
8	91496001
16	64018001
32	64018001
64	64018001
128	64018001

Como se puede ver en la tabla, el tiempo de ejecución disminuye a medida que el número de threads aumenta. Pasa igual que en un apartado anterior, cada vez la ganancia de tiempo es más pequeña hasta que llega a ser nula, como se puede ver en el siguiente gráfico:

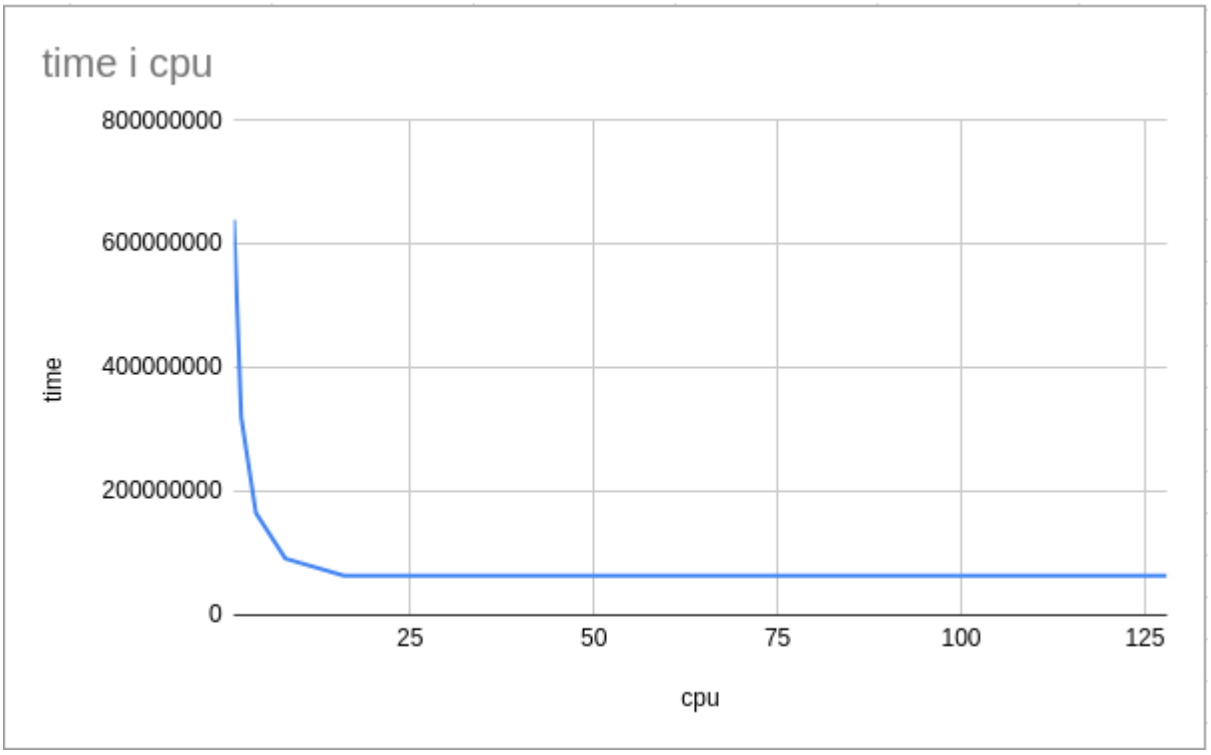


Figura 2.11: Gráfico con el tiempo de ejecución de la v4 de 3dfft_tar a medida que el número de procesadores aumenta.

Gracias al gráfico de la figura 2.11 podemos observar que el número de procesadores máximo hasta que el programa deja de ir más rápido está entre 12 y 20 (suponemos que son 16 ya que es una potencia pura de 2. También se puede ver en la tabla con los valores del tiempo de ejecución: a partir de 16 procesadores este no cambia)

2.1.6 - Versión 5

En esta versión definitiva del código se pedía desgranar aún más las tareas haciendo que fueran dentro del siguiente nivel del bucle y así comprobar si se podía paralelizar aún más.

```
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");
      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");
    }


  }
}
```

Figura 2.12: código enfocado a tareador de la v5 de 3dfft_tar.c

El grafo de dependencias en este caso es a simple vista una locura (y muy chulo):

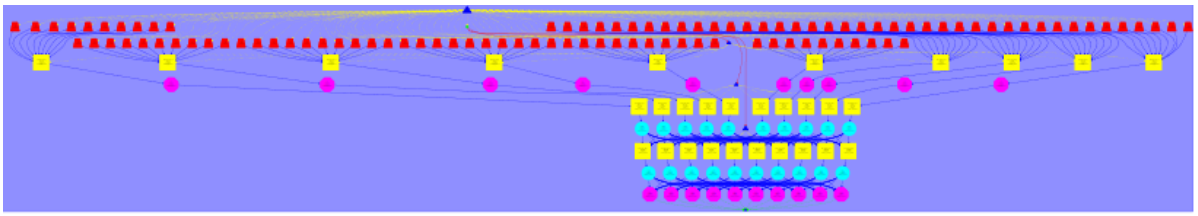


Figura 2.13: TDG de la v5 de 3dfft_tar.c

El primer y segundo piso de este grafo de dependencias es muy largo horizontalmente, esto seguramente significa que para alcanzar el máximo rendimiento se necesitan más procesadores que el máximo especificado en la v4.

Para ver la escalabilidad de la versión 5 se usará el mismo método que para la versión 4. A continuación se adjunta una tabla con los valores obtenidos:

nº de procesadores	**T_{exe} (ns)**
1	639780001
2	320081001
4	165721001
8	94020001
16	60913001
32	57928001
64	56233001
128	55820001

A diferencia de la versión 4, en esta tabla el tiempo de ejecución deja de disminuir notablemente a los 32 procesadores en lugar de a los 16, como ya teorizamos anteriormente.

Para que quede más visual, se puede observar este cambio gráficamente en la figura 2.14. La curvatura del gráfico es mucho más progresiva que la de v4:

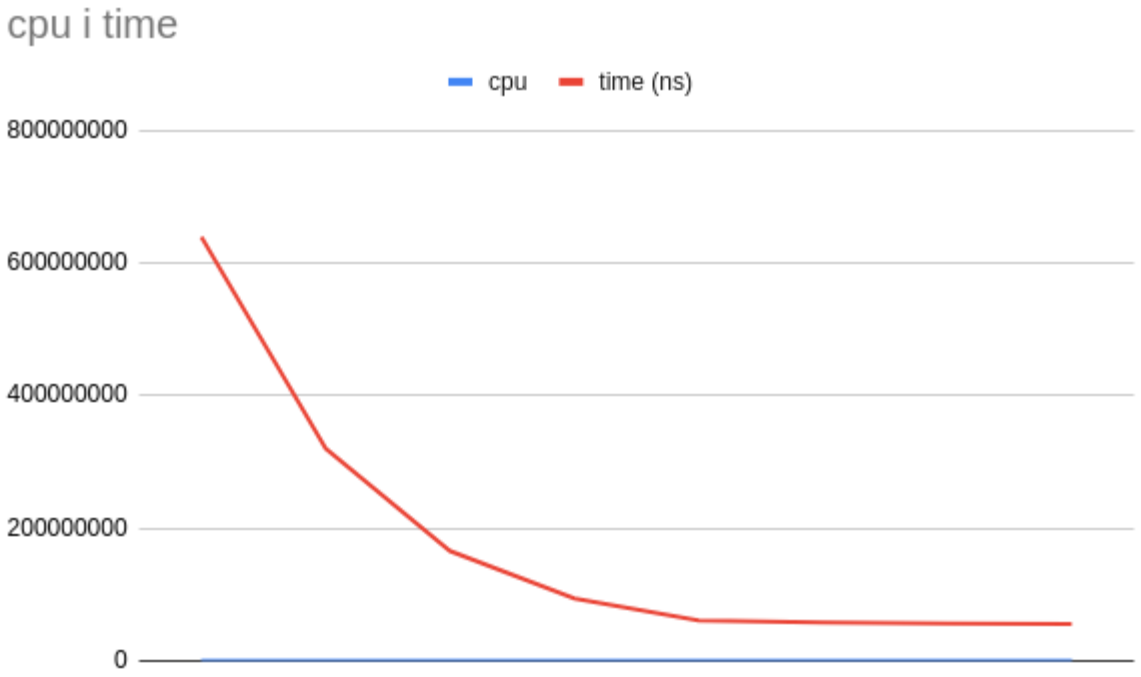


Figura 2.14: Gráfico con el tiempo de ejecución de la v5 de 3dfft_tar a medida que el número de procesadores aumenta.

2.2 - Análisis de los resultados

Tabla con los resultados de tiempos y paralelismo de cada versión:

Versión	t_1 (ms)	t_∞ (ms)	Paralelismo
v0	639.780	639.780	1
v1	639.780	639.707	~1
v2	639.780	361.190	1.77
v3	639.780	150.681	4.25
v4	639.780	64.018	9.99
v5	639.780	55.820	11.46

Como ya se ha comentado anteriormente, el tiempo secuencial del programa es siempre el mismo, lo que cambia es el tiempo al ejecutarse con supuestamente infinitos recursos (de ahí viene el paralelismo bueno/malo). Se puede ver cómo en cada versión del código este tiempo disminuye.

Una propiedad curiosa que se puede ver en la tabla es que el “paralelismo” de una versión es exactamente el Speedup que se obtiene de esa versión comparándola con la original. Por ejemplo, en la versión 3:

Paralelismo -> t_1 / t_∞ -> 639.780 / 150.681 = 4.25

Speedup respecto a v0: $t_{\infty 0} / t_{\infty 3}$ -> 639.780 / 150.681 = 4.25

En la siguiente tabla podemos ver el Speedup obtenido de cada versión respecto a la anterior:

V_i	**Speedup respecto a V_{i-1}**
v1	639.780 / 639.707 = **~1**
v2	639.707 / 361.190 = **1.77**
v3	361.190 / 150.681 = **2.397**
v4	150.681 / 64.018 = **2.354**
v5	64.018 / 55.820 = **1.147**

Como indica la tabla, en todas las versiones se logra hacer el programa más rápido.

Sesión 3: Understanding the execution of OpenMP programs

En esta última sesión trabajaremos con el entorno de Paraver, usado para obtener información sobre la ejecución de aplicaciones paralelas en OpenMP y visualizarlo.

Necesitaremos obtener varios datos que nos proporciona Paraver, y otros los cuales calcularemos a partir de ellos:

- T1: El tiempo de ejecución usando un solo procesador(secuencial).
- T8: El tiempo de ejecución usando 8 procesadores simultáneamente.
- S8: El speed-up que se consigue teniendo 8 procesadores.
- Φ: Porcentaje de parte paralela.
- S∞: El speed-up que podremos conseguir si tenemos infinitos procesadores.

(En esta práctica , Paraver simulará threads en lugar de procesadores)

3.1 Versión inicial

En esta primera parte, empezaremos encontrando T1, que se puede observar directamente en la figura 3.1. Para T_{par} utilizaremos *OMP implicit tasks duration* (Figura 3.2 , sumando todo los tiempos). Obtenemos:

- Tpar = 1617394770 ns
- T1 = 2491270845 ns

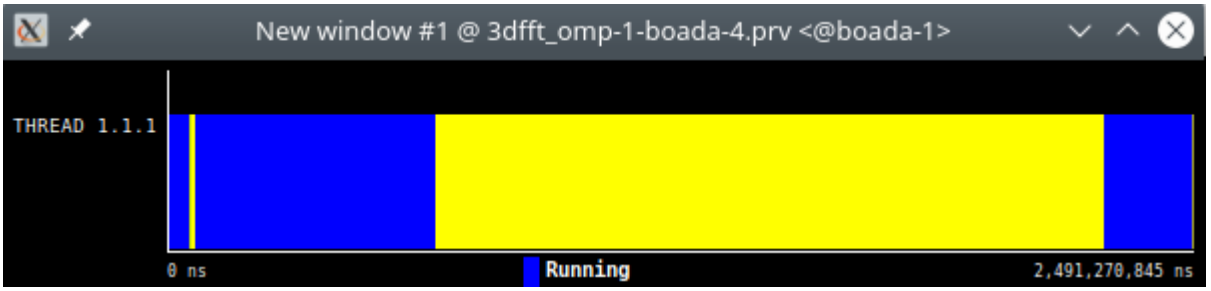
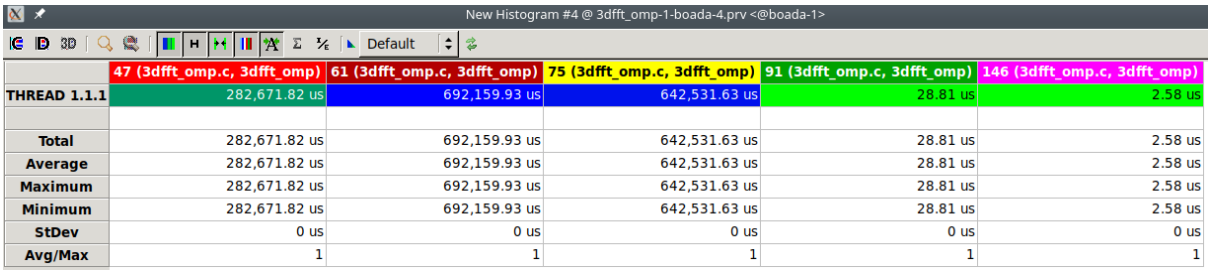


Figura 3.1 : Traza de ejecución con un solo thread



	47 (3dfft_omp.c, 3dfft_omp)	61 (3dfft_omp.c, 3dfft_omp)	75 (3dfft_omp.c, 3dfft_omp)	91 (3dfft_omp.c, 3dfft_omp)	146 (3dfft_omp.c, 3dfft_omp)
THREAD 1.1.1	282,671.82 us	692,159.93 us	642,531.63 us	28.81 us	2.58 us
Total	282,671.82 us	692,159.93 us	642,531.63 us	28.81 us	2.58 us
Average	282,671.82 us	692,159.93 us	642,531.63 us	28.81 us	2.58 us
Maximum	282,671.82 us	692,159.93 us	642,531.63 us	28.81 us	2.58 us
Minimum	282,671.82 us	692,159.93 us	642,531.63 us	28.81 us	2.58 us
StDev	0 us	0 us	0 us	0 us	0 us
Avg/Max	1	1	1	1	1

Figura 3.2 : Tiempo de ejecución de la parte paralela

Aplicamos las siguientes fórmulas para conseguir Tseq y Φ:

Tseq = T1 - Tpar = 2491270845 - 1617394770 = 873876075 ns
Φ = Tpar / (Tpar+Tseq) = 1617394770 / (1617394770+873876075) = 0.6492247815
Φ = 64.92247815%

Después hemos cambiado la traza de ejecución por 8 threads , para conseguir T8(Figura 3.3).

T8 = 1560509813 ns

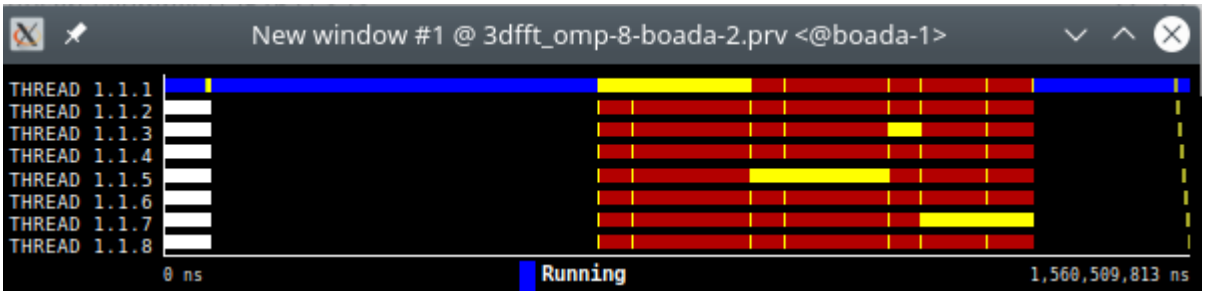


Figura 3.3: Traza de ejecución con 8 threads

Para conseguir el speed-up S8:

S8 = T1/T8 = 2491270845 / 1560509813 = 1.596446766

Antes de calcular S∞, tenemos que volver a ejecutar el programa con 1 thread y 8 threads, pero sin usar Paraver. Esto lo hacemos ya que Paraver nos introduce algún tiempo extra para su inicio. Ejecutamos el script submit-omp.sh que no producirá ningún tiempo extra. Obtendremos diferentes partes paralelizadas y su tiempo de ejecución según la figura 3.4 y 3.5.

```
par2311@boada-1:~/lab1/3dfft$ cat 3dfft_omp-1-boada-2.txt
3D FFT Plan Generation:0.000462s
Init Complex Grid FFT3D:0.581500s
Execution FFT3D:1.626750s
```

Figura 3.4: Sin tiempo extra T1

```
par2311@boada-1:~/lab1/3dfft$ cat 3dfft_omp-8-boada-2.txt
3D FFT Plan Generation:0.000388s
Init Complex Grid FFT3D:0.590093s
Execution FFT3D:0.686283s
```

Figura 3.5: Sin tiempo extra T8

T1 = 0.000462 + 0.581500 + 1.626750 = 2.208712s = 2 208 712 000 ns
T8 = 0.000388 + 0.590093 + 0.686283 = 1.276764s = 1 276 764 000 ns

S8 = T1/T8 = 2 208 712 000 / 1 276 764 000 = 1.729929729

Podemos observar un overhead significativo cuando usamos Paraver. Esto nos indica que si no lo utilizamos el rendimiento aumenta un porcentaje significativo. Al final para encontrar S∞ utilizaremos la siguiente ecuación:

S∞ = 1/(1-Φ) = 1 / (1-0.6492247815) = 2.850828529

Esto quiere decir que con un 65% de código paralelizado sólo podríamos conseguir ejecutar el programa 2.85 veces más rápido teniendo infinitos procesadores. Las

siguientes figuras nos muestran el tiempo de ejecución y el speed-up según el número de threads (strong-scalability).

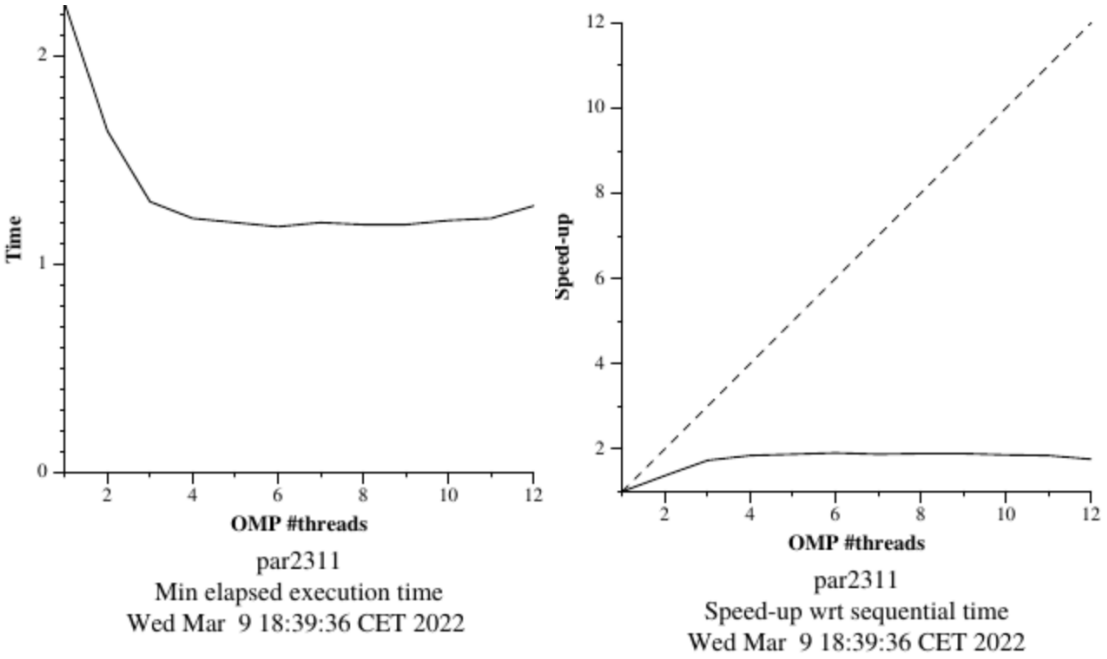


Figura 3.6: Evolución de tiempo y speed-up según el incremento de threads

3.2 Improving Φ

En este apartado tenemos que encontrar la parte del código que hace nuestro código menos paralelizable. Esta parte era la función *init_complex_grid.* Para aumentar el paralelismo descomentamos los “pragma” de la función excepto el taskloop que está fuera del bucle.

```
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
      }
}
```

Figura 3.7: Código versión 2

Ahora tenemos que aplicar las mismas fórmulas en el apartado 3.1 para encontrar los datos que necesitamos. T1 no tiene el mismo resultado que el apartado anterior, pensamos que puede ser culpa de Paraver.

T1 = 2657901257 ns
Tpar = 2380951360 ns
Tseq = T1 - Tpar = 276949897 ns
Φ = Tpar/(Tseq+ Tpar) = 0.8958012845 = 89,58012845%
T8 = 1035682605 ns
S8 = T1/T8 = 2.566327989
S∞=1/(1-Φ) = 9.597047288

Los resultados que hemos obtenido tienen sentido ya que han mejorado el tiempo de ejecución porque hemos aumentado el paralelismo. Los valores que hemos obtenido para los cálculos están en las siguientes figuras:

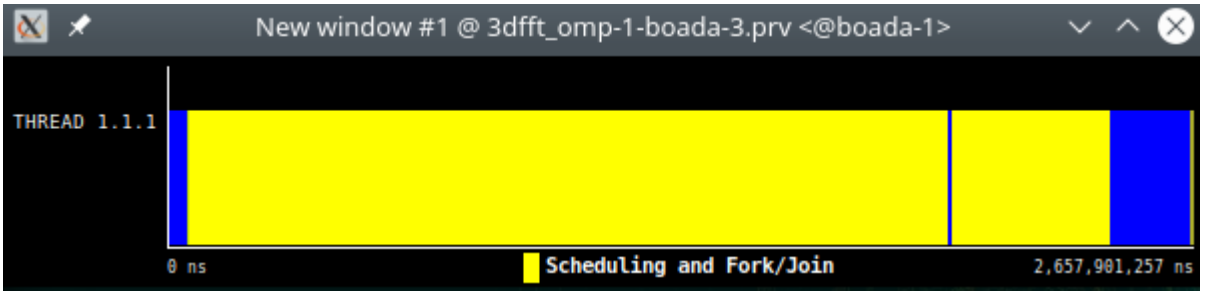


Figura 3.8: Tiempo de ejecución : T1_v2

New Histogram #1 @ 3dfft_omp-1-boada-3.prv <@boada-1>

Default

	np.c, 3dfft_omp)	49 (3dfft_omp.c, 3dfft_omp)	63 (3dfft_omp.c, 3dfft_omp)	77 (3dfft_omp.c, 3dfft_omp)	93 (3dfft_omp.c, 3dfft_omp)
THREAD 1.1.1	602,318.84 us	298,146.39 us	837,250.36 us	643,215.31 us	18.02 us
Total	602,318.84 us	298,146.39 us	837,250.36 us	643,215.31 us	18.02 us
Average	602,318.84 us	298,146.39 us	837,250.36 us	643,215.31 us	18.02 us
Maximum	602,318.84 us	298,146.39 us	837,250.36 us	643,215.31 us	18.02 us
Minimum	602,318.84 us	298,146.39 us	837,250.36 us	643,215.31 us	18.02 us
StDev	0 us	0 us	0 us	0 us	0 us
Avg/Max	1	1	1	1	1

Figura 3.9: Tiempo de ejecución de la parte paralela

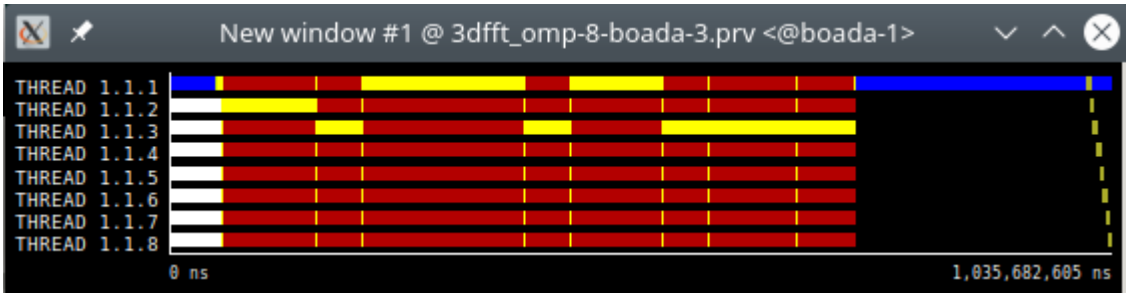


Figura 3.10: Tiempo de ejecución : T8_v2

Finalmente, para este apartado podemos observar en la figura 3.11 en comparación con la 3.6. Ha mejorado significativamente por el aumento de paralelismo.

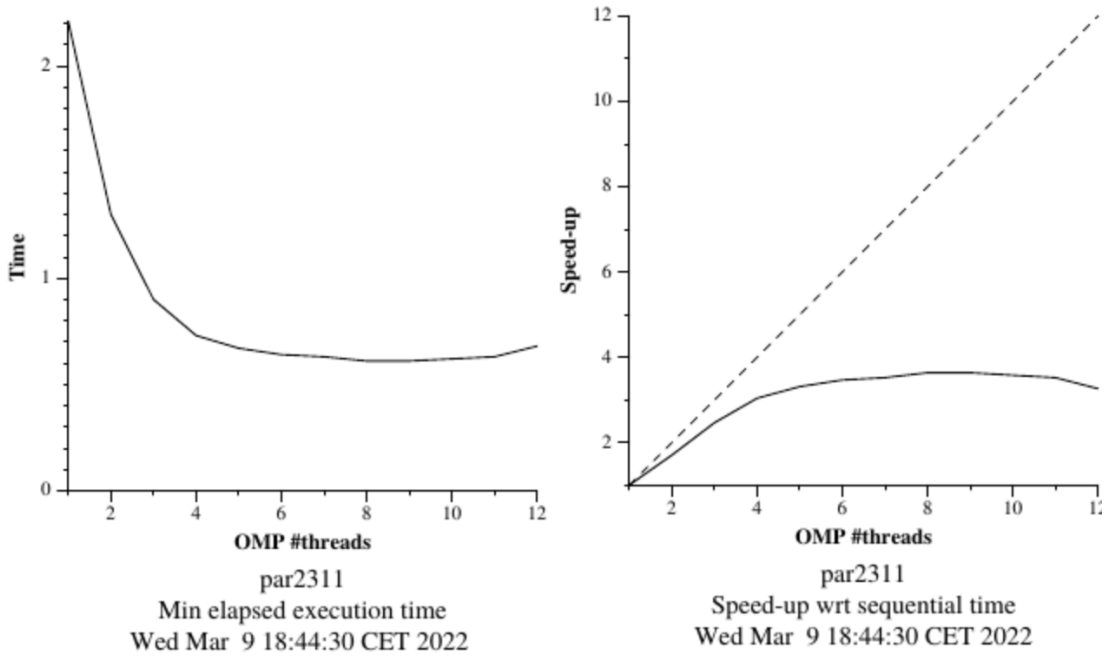


Figura 3.11: Evolución de tiempo y speed-up según el incremento de threads (v_2)

3.3 Reducing parallelisation overheads

Por último, tenemos que aumentar la granularidad de las tareas comentando el taskloop interior del código y descomentar el de fuera:

```
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
      }
}
```

Figura 3.12

Este cambio nos permite reducir los overheads para poder obtener mejores rendimientos.

T1 = 2471976794 ns
Tpar = 2280037540 ns
Tseq = T1 - Tpar = 191939254 ns
Φ = Tpar /(Tpar+Tseq) = 0.9223539418 = 92,23539418 %
T8 = 1177242078 ns
S8 = T1/T8 = 2.099803295
S∞ = 1/(1-Φ) = 12.87895385

Hemos visto que este pequeño cambio nos mejora el S∞ , pero nos reduce el S8, pensamos que puede que haya sido algún problema de Paraver que ha ralentizado la ejecución o la gran parte secuencial que muestra la figura 3.15. En las siguientes figuras mostramos los algunos tablas y trazas de donde hemos obtenido valores:

Figura 3.13: Tiempo de ejecución : T1_v3

New Histogram #1 @ 3dfft_omp-1-boada-2.prv <@boada-1>

Default

	29 (3dfft_omp.c, 3dfft_omp)	47 (3dfft_omp.c, 3dfft_omp)	61 (3dfft_omp.c, 3dfft_omp)	75 (3dfft_omp.c, 3dfft_omp)	91 (3dfft_omp.c, 3dfft_omp)
THREAD 1.1.1	587,131.08 us	283,279.97 us	774,562.14 us	635,033.84 us	28.03 us
Total	587,131.08 us	283,279.97 us	774,562.14 us	635,033.84 us	28.03 us
Average	587,131.08 us	283,279.97 us	774,562.14 us	635,033.84 us	28.03 us
Maximum	587,131.08 us	283,279.97 us	774,562.14 us	635,033.84 us	28.03 us
Minimum	587,131.08 us	283,279.97 us	774,562.14 us	635,033.84 us	28.03 us
StDev	0 us	0 us	0 us	0 us	0 us
Avg/Max	1	1	1	1	1

Figura 3.14:Tiempo de ejecución de la parte paralela

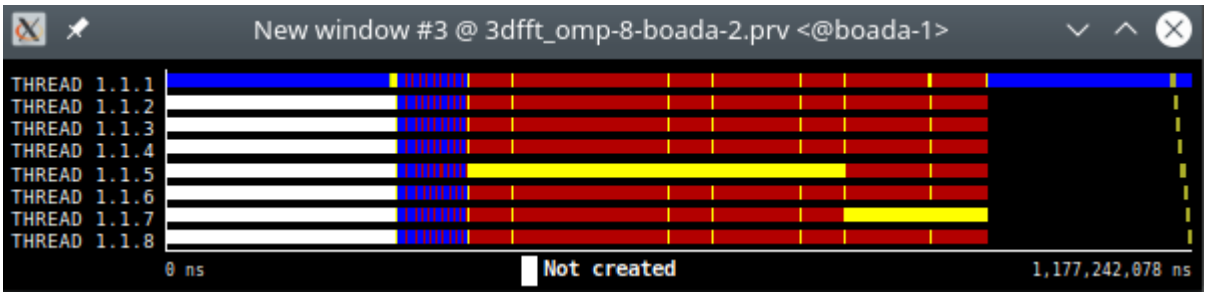


Figura 3.15: Tiempo de ejecución : T8_v3

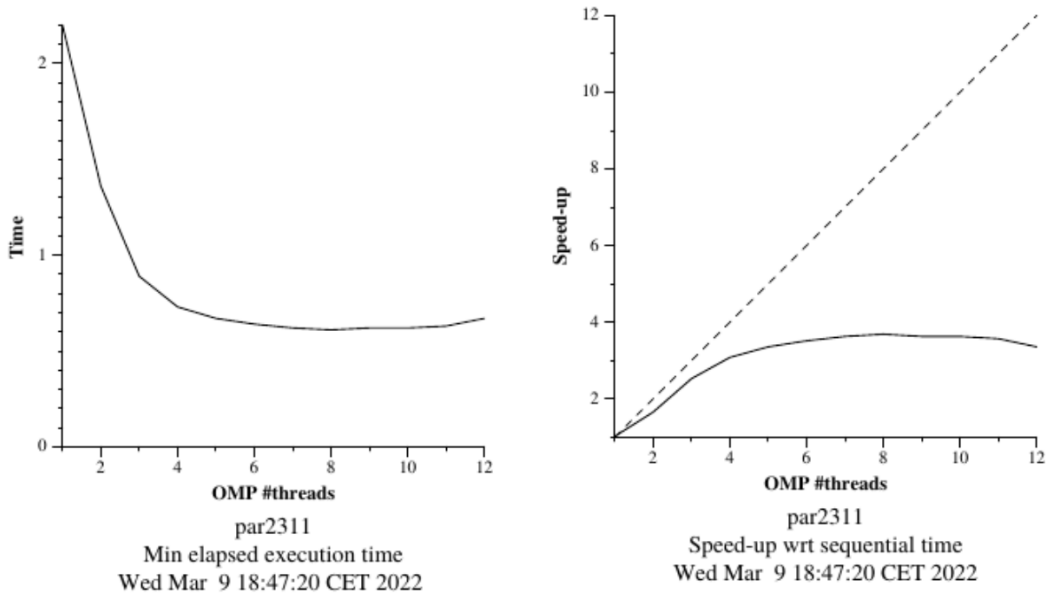


Figura 3.16: Evolución de tiempo y speed-up según el incremento de threads (v_3)

3.4 Conclusión

Versión	Φ	S∞	T1(s)	T8(s)	S8
Inicial	64,92%	2.85	2.21	1.28	1.73
Version_v2	89.58%	9.60	2.66	1.03	2.566
Version_v3	92.24%	12.88	2.47	1.17	2.1

En conclusión: hemos observado cómo ha evolucionado el rendimiento del programa. Primero de todo añadimos paralelismo a la función, con eso el rendimiento mejoró significativamente cuando se usaba más de un procesador (en nuestro caso threads). Después, aumentando la granularidad pudimos observar un pequeño aumento en la fracción paralela, pero también un decremento cuando hay 8 procesadores (threads) respecto la versión 2, que pensamos que podría ser un problema del programa.