Lab1: Experimental Setup and Tools
Q2 2021-22

Integrantes:
Pol Pérez Castillo - par2116
Cristian Sánchez Estapé - par2118

Fecha de entrega: 08-03-2022

Índice

Node architecture and memory

El servidor boada.ac.upc.edu está formado por 9 nodos (desde boada-1, que es el login node, hasta el boada-9). En el marco de esta asignatura solo usaremos desde el nodo 1 hasta el 4.

Las especificaciones de cada nodo y de la memoria principal y cache vienen definidas en la siguiente tabla:

	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	2495 MHz
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)	12288kB
Main memory size (per socket)	12GB
Main memory size (per node)	23GB

Figura 1: tabla de información técnica de Boada

En el diagrama de la arquitectura se puede ver que un nodo está dividido en 2 sockets y como se divide la caché de cada socket y los diferentes PCI (Peripheral Component Interconnect) integrados en el nodo.

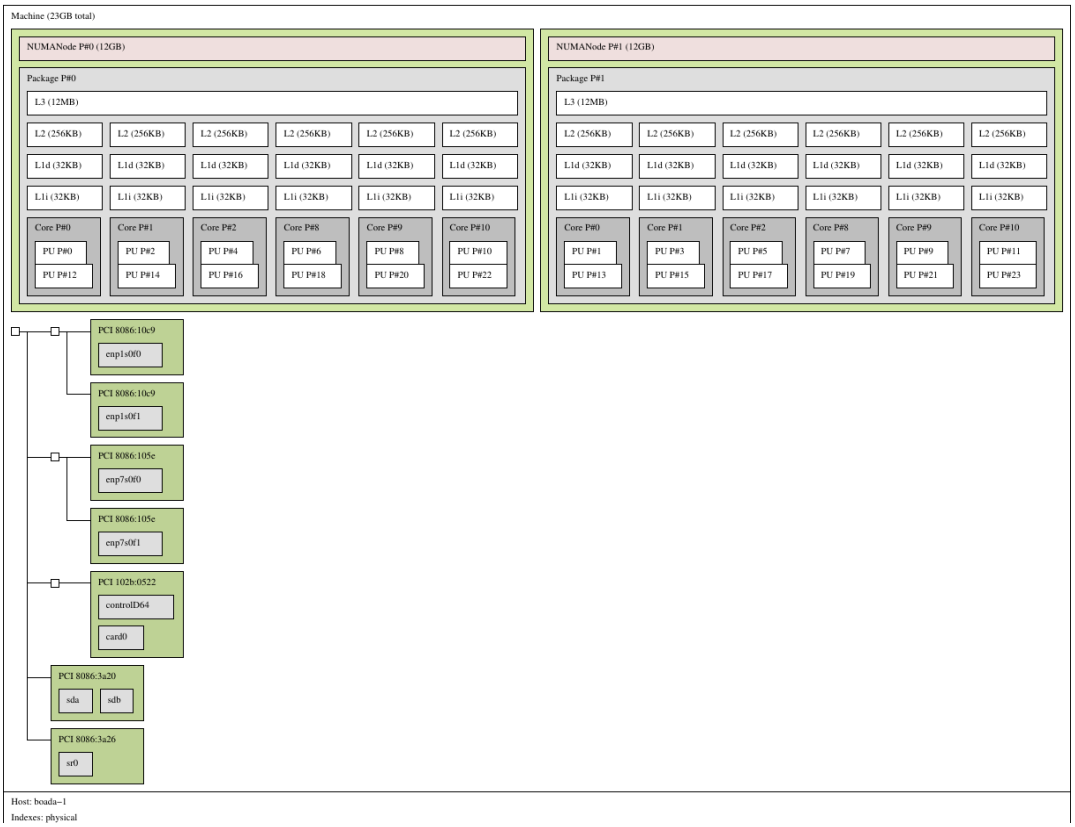

Figura 2: representación de la arquitectura de Boada

Strong vs. weak scalability

# threads	Interactive				Queued			
	user	system	elapsed	% of CPU	user	system	elapsed	% of CPU
1	3,94	0,00	0:03,95	99	3,94	0,00	0:03,97	99
2	8,00	0,00	0:04,01	199	3,95	0,00	0:01,99	197
4	8,00	0,00	0:04,01	199	3,98	0,00	0:01,01	393
8	8,01	0,00	0:04,03	199	4,17	0,01	0:00,54	769

Figura 3: tabla comparativa de los modos de ejecución de Boada

Por un lado, en el caso interactivo tenemos un tiempo *elapsed* y de usuario mayor, juntamente con un uso de CPU generalmente inferior en comparación al *queued*. Esto se debe a que la ejecución no se realiza con todas las CPUs posibles, con lo cual se requiere de un mayor tiempo de ejecución y, por lo tanto, menor uso de CPUs.

Por otro lado, en el caso *queued* tenemos un tiempo *elapsed* y de usuario menor, juntamente con un uso de CPU generalmente superior. Esto se debe a que, para este tipo de ejecución, se reservan varias CPUs, lo cual aumenta el porcentaje de uso de CPU y disminuye (o al menos controla) el tiempo de ejecución.

1- Ejecución de sumbit-strong-omp.sh con np_NMAX=12 threads

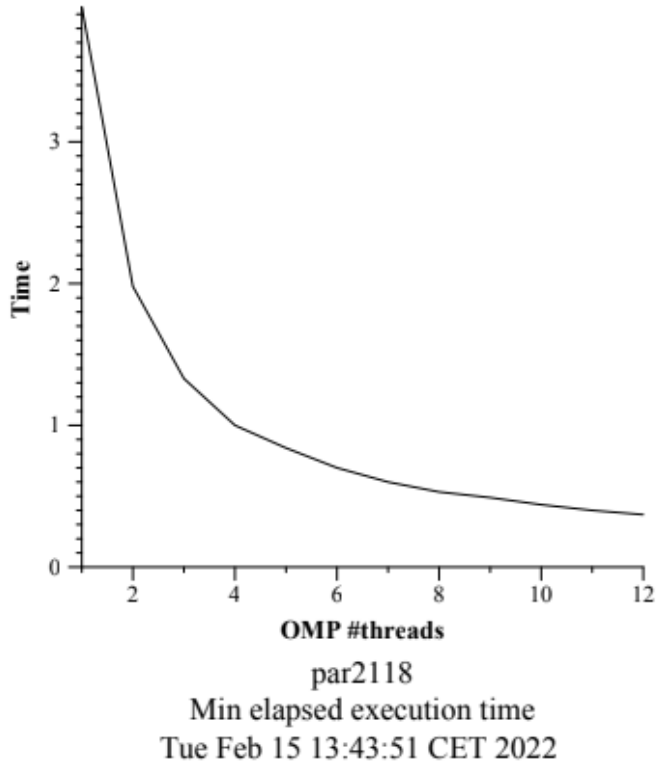

Figura 4: tiempo de ejecución del fichero pi_omp según los threads (12 máximo)

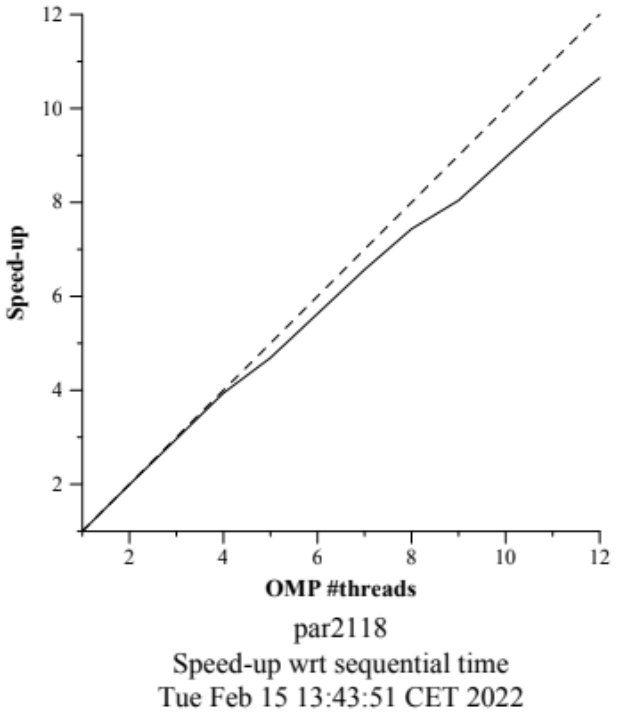

Figura 5: speed-up del fichero pi_omp según los threads (12 máximo)

Tal y como la definición de strong scalability nos indica, se puede ver cómo, al aumentar el número de threads, se reduce el tiempo de ejecución del programa y el speed-up es prácticamente lineal al mismo. Sin embargo, podemos asumir que, cuantos más threads tengamos, más estancamiento de la relación lineal habrá y, por lo tanto, llegará un punto en que el speed-up obtenido generalmente no mejorará más.

2- Ejecución de sumbit-strong-omp.sh con np_NMAX=24 threads

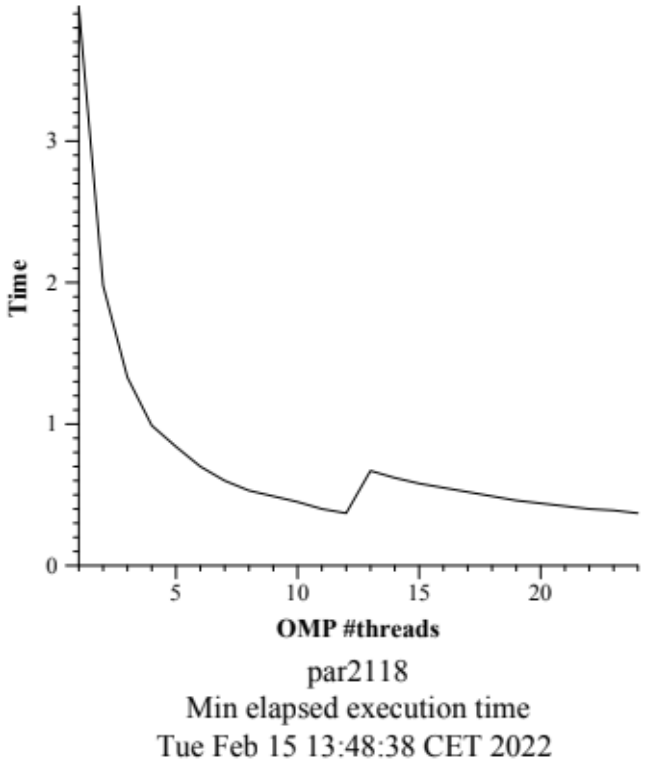

Figura 6: tiempo de ejecución del fichero pi_omp según los threads (24 máximo)

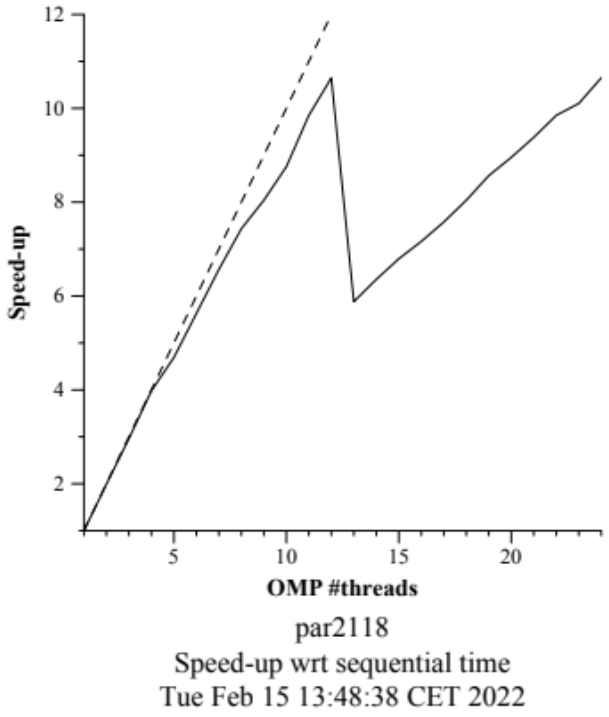

Figura 7: speed-up del fichero pi_omp según los threads (24 máximo)

Tal y como se puede observar, el incremento de 12 a 24 threads, si bien implica un cierto incremento en el rendimiento y en el speed-up, al llegar hasta un cierto nº de threads (en este caso 13), la optimización se vuelve ineficiente y se incurre en un decremento del rendimiento de la ejecución del programa. Esta ineficiencia podría ser provocada por el hecho que, al aumentar de 12 a 13 threads, dado que cada nodo tiene 12 CPUs (6 por socket), se debe hacer uso de *otro* nodo, lo cual se traduce en la caída aquí presentada.

3- Ejecución de submit-weak-omp.sh

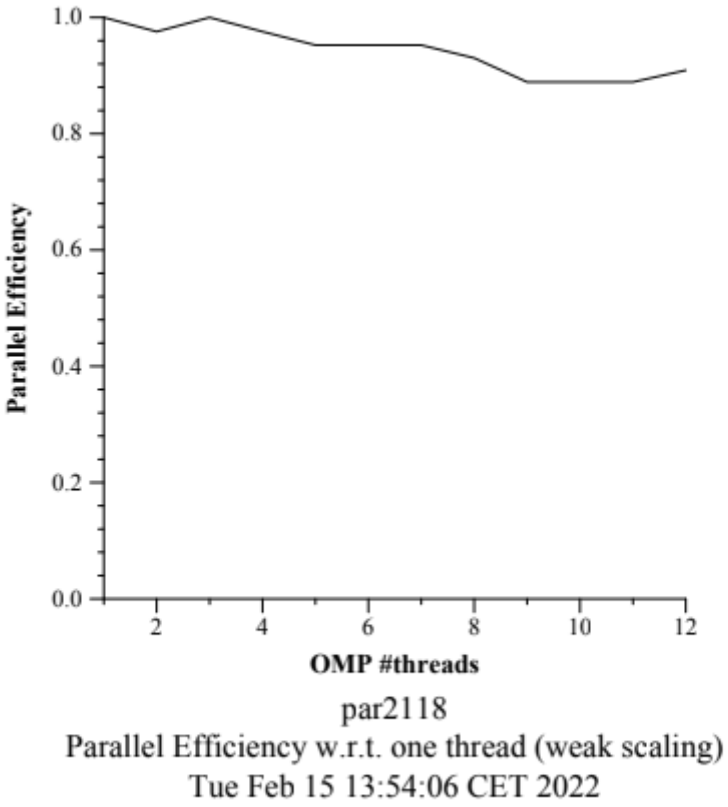

Figura 8: eficiencia del paralelismo según los threads

En este caso, la eficiencia del programa en la ejecución en paralelo disminuye cuantos más threads sean utilizados, ya que el tamaño del problema también aumenta a la par que estos. El aumento de threads, por lo tanto, no es suficientemente significativo como para aumentar la eficiencia de la ejecución del programa.

Analysis of task decompositions for 3DFFT

Version	T_1	T_∞	**Parallelism**
seq	639670	639650	1,00003 ≈ 2
v1	639670	639650	1,00003 ≈ 2
v2	639670	361080	1,7715 ≈ 2
v3	639670	154244	4,1471 ≈ 5
v4	639670	63908	10,0092 ≈ 11
v5	639670	50050	12,7806 ≈ 13

Figura 9: tabla comparativa de los tiempos de ejecución según la versión

Lo que se puede constatar en base a los datos recogidos y aquí presentados es la relación existente entre paralelismo y granularidad: cuanto más fina es la granularidad, o lo que es lo mismo, cuanto más seccionamos los cálculos presentes en un programa, mayor es el paralelismo y, por ende, más capaces somos de reducir el tiempo de ejecución del programa. En relación a lo comentado, también cabe destacar que, si bien T_1 se mantiene constante (tal y como debería), T_∞ disminuye relevantemente a medida que aumentamos la granularidad con que se ejecuta la aplicación (especialmente entre las versiones v3 y v4). Además, también es constatable el hecho que el overhead, para cada versión aquí presentada, nunca es suficientemente elevado como para empezar a notar un decrecimiento del rendimiento con que se ejecuta la aplicación, pero se puede suponer que un incremento aún mayor de la granularidad implicaría una caída global en el rendimiento de la ejecución del programa.

- Grafos de dependencia de tareas

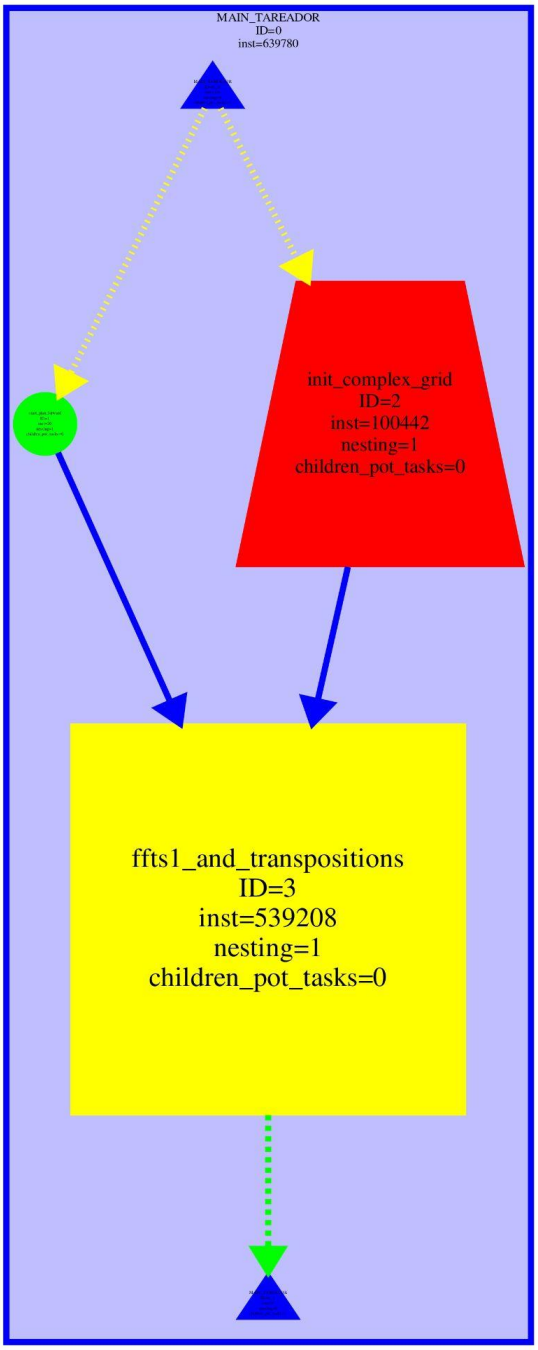

Figura 10: grafo de dependencia de la versión seq

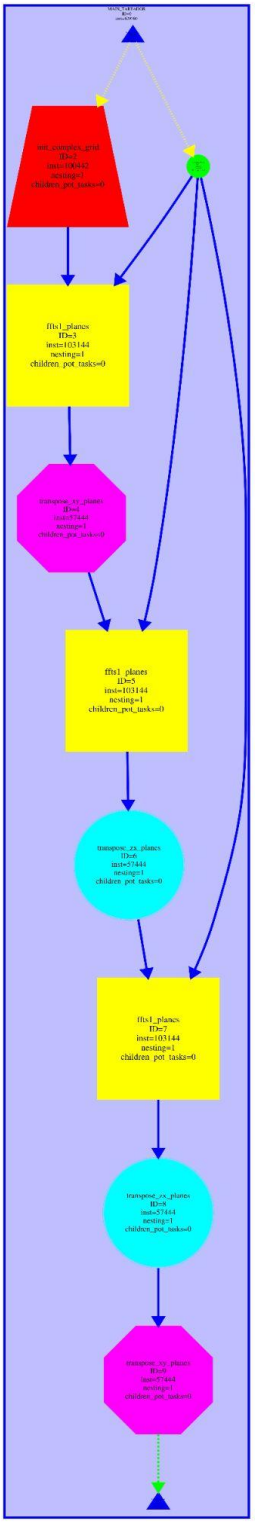

Figura 11: grafo de dependencia de la versión v1

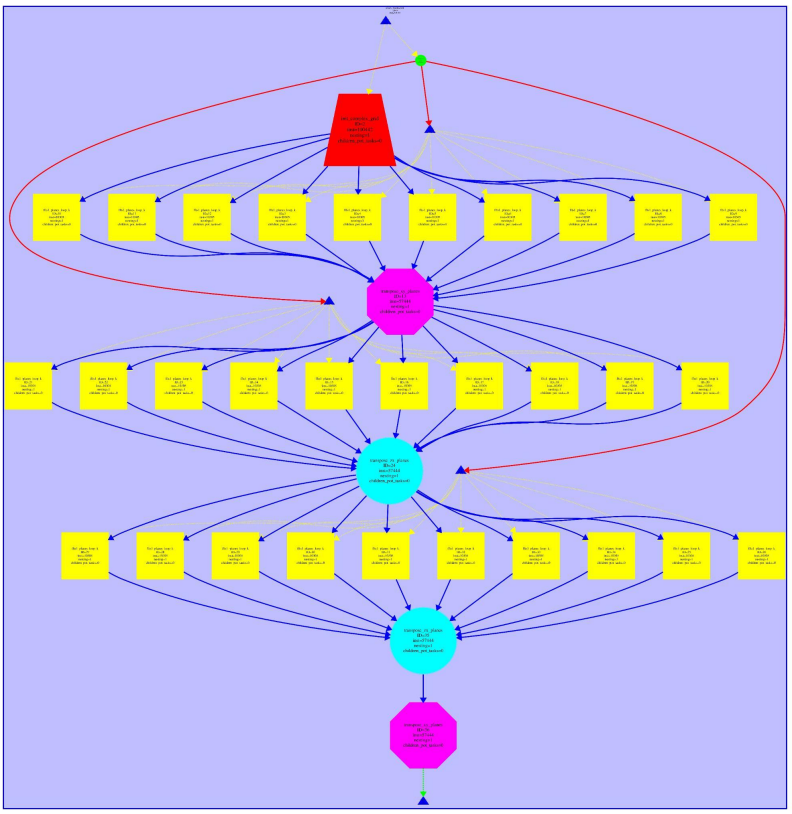

Figura 12: grafo de dependencia de la versión v2

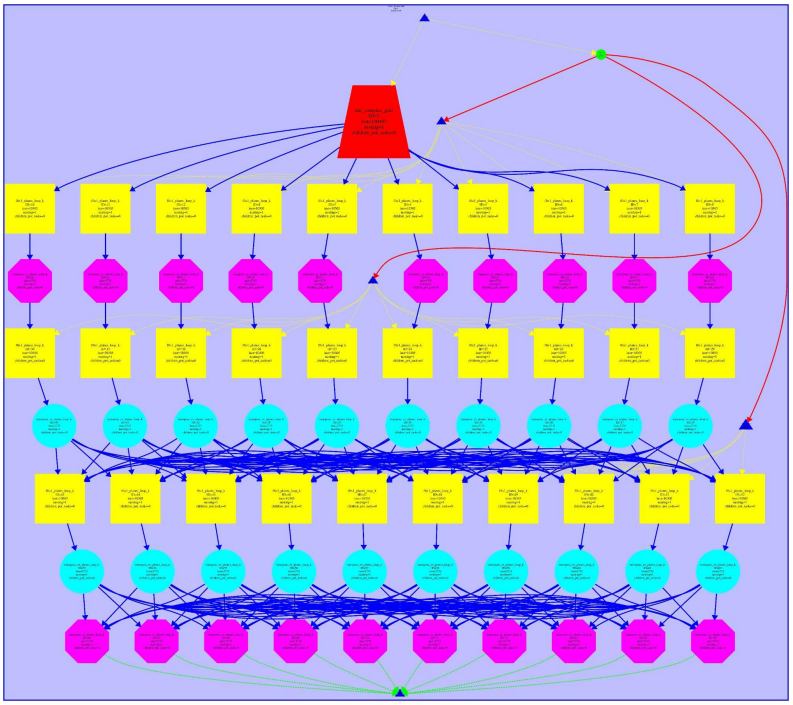

Figura 13: grafo de dependencia de la versión v3

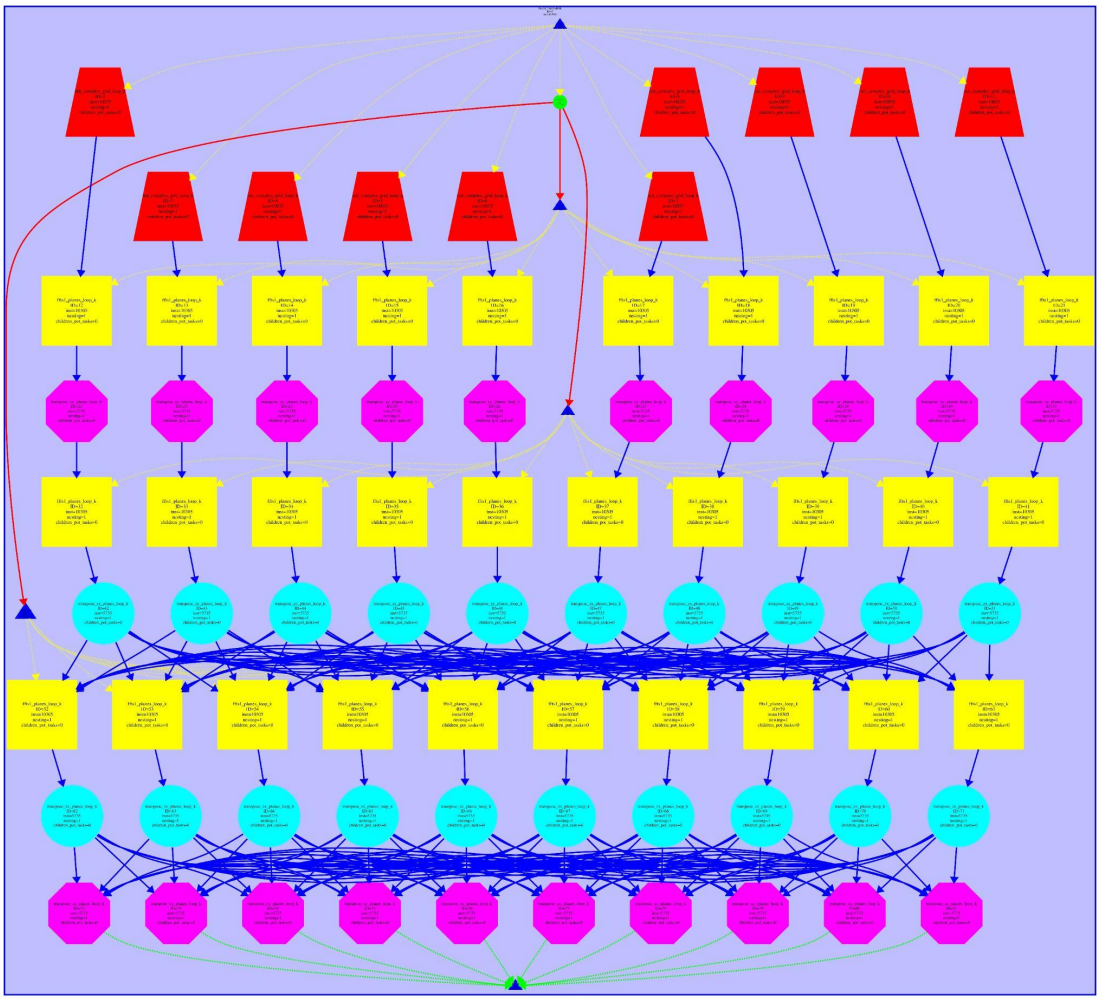

Figura 14: grafo de dependencia de la versión v4

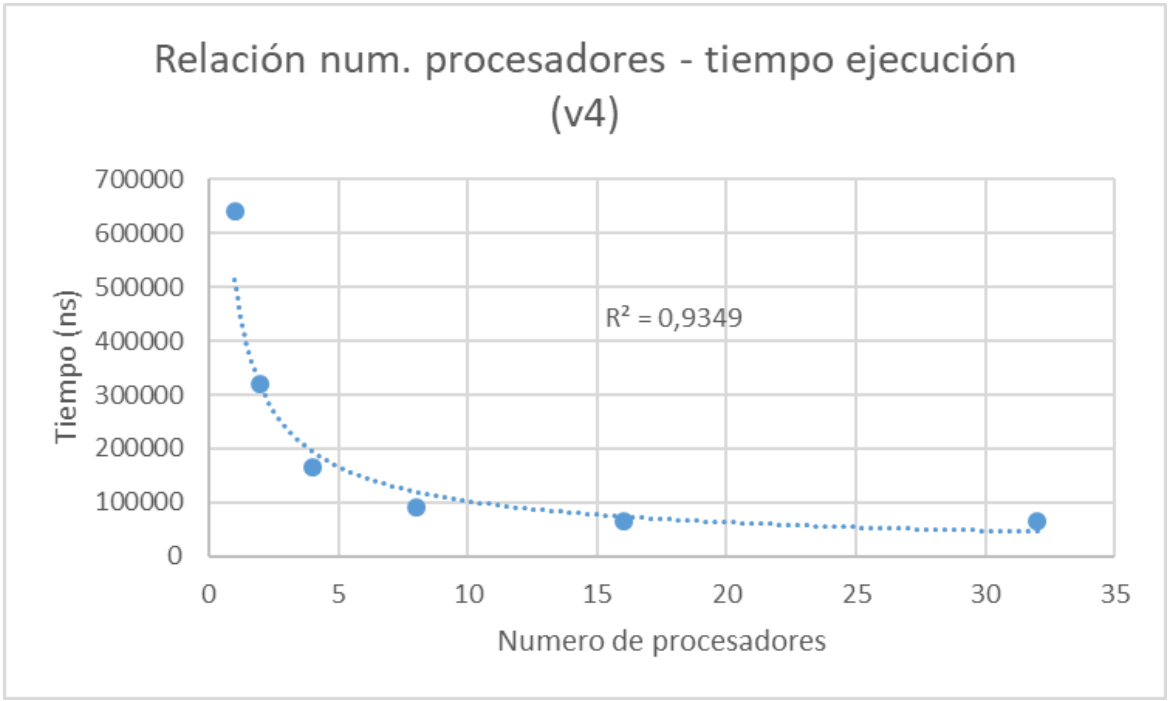

Figura 15: gráfico representando la relación entre tiempo de ejecución y número de procesadores para la versión v4

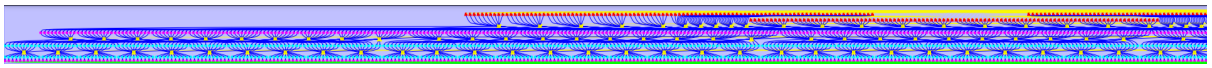

Figura 16: grafo de dependencia de la versión v5[1]

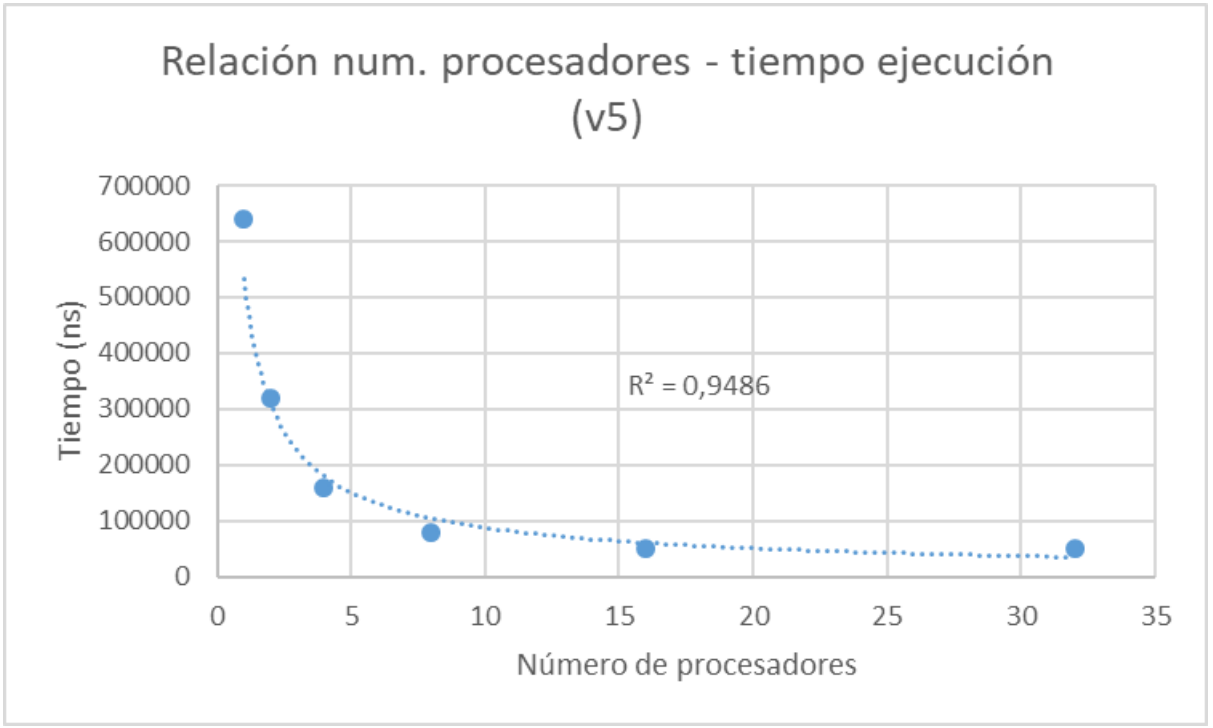

Figura 17: gráfico representando la relación entre tiempo de ejecución y número de procesadores para la versión v5

En las versiones v4 y v5 se aumentan considerablemente el nº de tareas en que dividimos el programa y, por esta razón, al aumentar la granularidad, el paralelismo que se obtiene es mucho mayor que en versiones anteriores, con lo cual el potencial de escalabilidad es muy significativo. Aun así, es apreciable que, eventualmente, resulta imposible disminuir más el tiempo de ejecución.

[1] El tamaño del grafo de dependencias es tal que aquí nos hemos limitado a presentar una pequeña parte, con tal de que se pueda apreciar mínimamente lo que en este aparece.

Versión 4:

```
void init_complex_grid(fftwf_complex in_fftw[][N][N]) {
  int k,j,i;
  for (k = 0; k < N; k++) {
      tareador_start_task("init_complex_grid_loop_k");
    for (j = 0; j < N; j++) {
      for (i = 0; i < N; i++) {
        in_fftw[k][j][i][0] = (float)
(sin(M_PI*((float)i)/64.0)+sin(M_PI*((float)i)/32.0)+sin(M_PI*((float)i/
16.0)));
        in_fftw[k][j][i][1] = 0;
#if TEST
        out_fftw[k][j][i][0]= in_fftw[k][j][i][0];
        out_fftw[k][j][i][1]= in_fftw[k][j][i][1];
#endif
      }
    }
      tareador_end_task("init_complex_grid_loop_k");
  }
}
```

*Figura 18: código de la función **init_complex_grid** de la versión v4*

Versión 5:

```
void init_complex_grid(fftwf_complex in_fftw[][N][N]) {
  int k,j,i;
  for (k = 0; k < N; k++) {
      //tareador_start_task("init_complex_grid_loop_k");
    for (j = 0; j < N; j++) {
      for (i = 0; i < N; i++) {
      tareador_start_task("init_complex_grid_loop_i");
        in_fftw[k][j][i][0] = (float)
(sin(M_PI*((float)i)/64.0)+sin(M_PI*((float)i)/32.0)+sin(M_PI*((float)i/
16.0)));
        in_fftw[k][j][i][1] = 0;
#if TEST
        out_fftw[k][j][i][0]= in_fftw[k][j][i][0];
        out_fftw[k][j][i][1]= in_fftw[k][j][i][1];
#endif
      tareador_end_task("init_complex_grid_loop_i");
      }
    }
      //tareador_end_task("init_complex_grid_loop_k");
  }
}
```

*Figura 19: código de la función **init_complex_grid** de la versión v5*

La diferencia entre estas dos versiones se basa en la granularidad de éstas. En la versión 4, cada tarea es definida en el bucle más externo de la función (se crean N tareas por función); en la versión 5, es definida en el bucle más interno (se crean N^3 tareas por función), esto significa que generalmente se han creado $N^3 - N$ tareas más. Con todo esto, el estancamiento en el paralelismo es debido a los límites de la paralelización: entre v3 y v4 hay un aumento del paralelismo de 6 unidades dado que los *overheads* de creación de tareas no suponen, aún, un elemento que pueda resultar contraproducente; sin embargo, entre v4 y v5 solo aumenta 2 unidades, con lo cual se empieza a apreciar la presencia de dichos *overheads*.

Understanding the parallel execution of 3DFFT

Version	***Φ***	***ideal*** S_8	T_1 ***(ms)***	T_8 ***(ms)***	***real*** S_8 ***(ms)***
initial version in 3dfft_omp.c	0.65	2.87	2672	1799	1.49
new version with improved Φ	0.87	7.44	2607	1034	2.52
final version with reduced parallelisation overheads	0.88	8.51	2597	689	3.77

Figura 20: tabla del tiempo secuencial, paralelizable y speed-up según la versión del fichero 3dfft_omp

La primera fila nos muestra los datos de la versión inicial de *3dfft_omp*. En esta primera versión, la función *init_complex_grid* no se paraleliza (Φ es más pequeña); esto provoca que el speed-up ideal y el speed-up real sean inferiores comparados a las otras versiones.

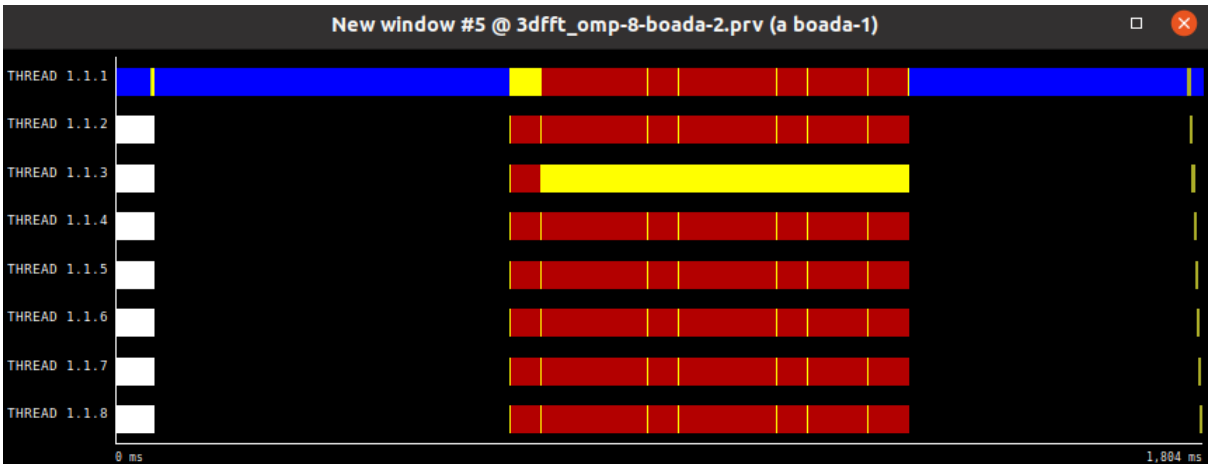

Figura 21: timeline del fichero 3dfft_omp (versión inicial)

New Histogram #13 @ 3dfft_omp-8-boada-2.prv (a boada-1)

Default

	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	247.94 ms	262.78 ms	152.20 ms	0.03 ms	0.00 ms
THREAD 1.1.2	247.94 ms	262.78 ms	152.10 ms	0.23 ms	0.00 ms
THREAD 1.1.3	247.94 ms	262.78 ms	152.11 ms	0.25 ms	0.00 ms
THREAD 1.1.4	247.94 ms	262.78 ms	152.10 ms	0.22 ms	0.00 ms
THREAD 1.1.5	247.94 ms	262.78 ms	152.13 ms	0.24 ms	0.00 ms
THREAD 1.1.6	247.95 ms	262.78 ms	152.01 ms	0.21 ms	0.00 ms
THREAD 1.1.7	247.94 ms	262.78 ms	152.07 ms	0.22 ms	0.00 ms
THREAD 1.1.8	247.94 ms	262.79 ms	152.06 ms	0.21 ms	0.00 ms
Total	1,983.53 ms	2,102.26 ms	1,216.78 ms	1.61 ms	0.02 ms
Average	247.94 ms	262.78 ms	152.10 ms	0.20 ms	0.00 ms
Maximum	247.95 ms	262.79 ms	152.20 ms	0.25 ms	0.00 ms
Minimum	247.94 ms	262.78 ms	152.01 ms	0.03 ms	0.00 ms
StDev	0.00 ms	0.00 ms	0.05 ms	0.07 ms	0.00 ms
Avg/Max	1.00	1.00	1.00	0.80	0.97

Figura 22: tabla con los tiempos de ejecución de las tareas según los threads (versión inicial)

Cabe añadir que la manera en que paralelizamos el programa es relevante, dado que, según esta, podremos mejorar más o menos el tiempo de ejecución total o, en caso contrario, incurrir en problemas de ineficiencia. El hecho de seccionar las tareas según los threads implica la presencia de overheads, tiempos de sincronización y otros elementos que terminan afectando a la mejora del rendimiento del programa. En el caso presente, lo que se puede observar es que los tiempos de ejecución por threads, si bien en total representan una cifra elevada (tanto que incluso llegan a superar el tiempo de ejecución total del programa), al verse ejecutadas en paralelo, consiguen reducir el tiempo de ejecución total. Aún así, se puede constatar una diferencia significativa entre el speed-up ideal y el real, diferencia que viene dada precisamente por las afectaciones comentadas.

En la segunda versión hemos mejorado el código descomentando los pragmas para permitir la ejecución en paralelo de la función *init_complex_grid*. Con esto se ha conseguido una notable mejora de Φ, consiguiendo así un speedup ideal mucho mayor. En contraparte, la gran cantidad de overheads generados provoca que el speedup real no se haya visto mejorado.

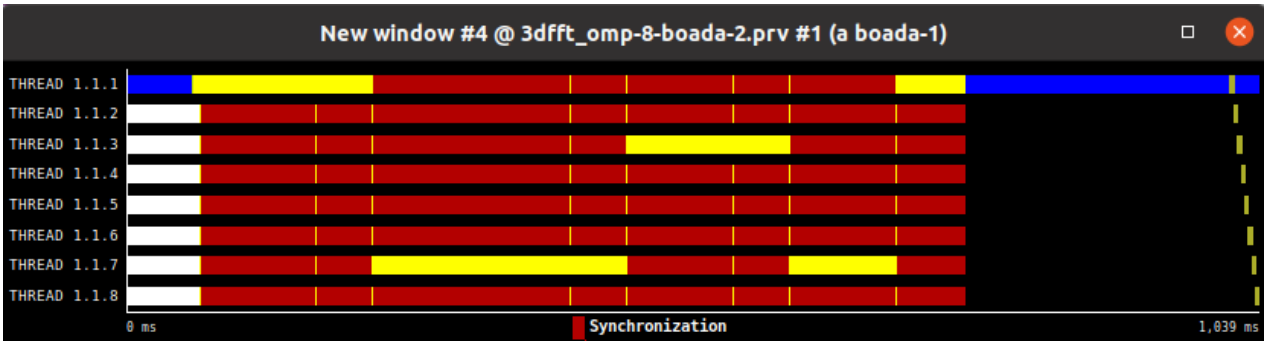

Figura 22: timeline del fichero 3dfft_omp (primera versión mejorada)

En la versión final, para solucionar el problema de los overheads, hemos paralelizado todas las funciones fuera del bucle y así evitar el tiempo malgastado en crear todas estas tareas y poder obtener una mejora del speedup real. También, gracias a estos cambios, ha disminuido considerablemente el tiempo de ejecución con 8 threads.

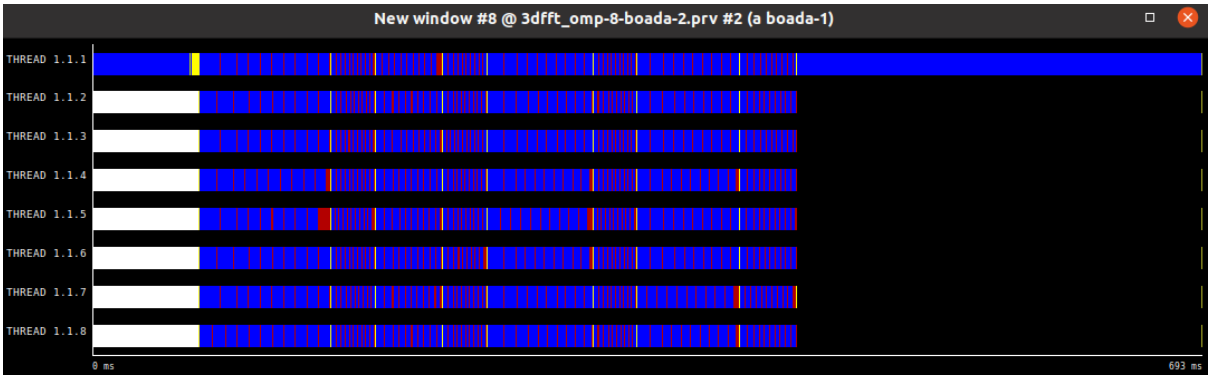

Figura 23: timeline del fichero 3dfft_omp (segunda versión mejorada)

Además, como se puede observar en las figuras 24, 25 y 26, existe una fuerte escalabilidad, mayor en cada una de estas versiones. Esto significa que, a cuantos más threads, menor es el tiempo de ejecución del programa. Tal y como hemos comentado anteriormente, este tiempo de ejecución se reduce en las dos últimas versiones ya que la paralelización llevada a cabo es más eficiente que en nuestra primera versión. Cabe destacar que el aumento de threads tiene un límite de mejora, tras el cual se incurre en una caída del rendimiento total del tiempo de ejecución.

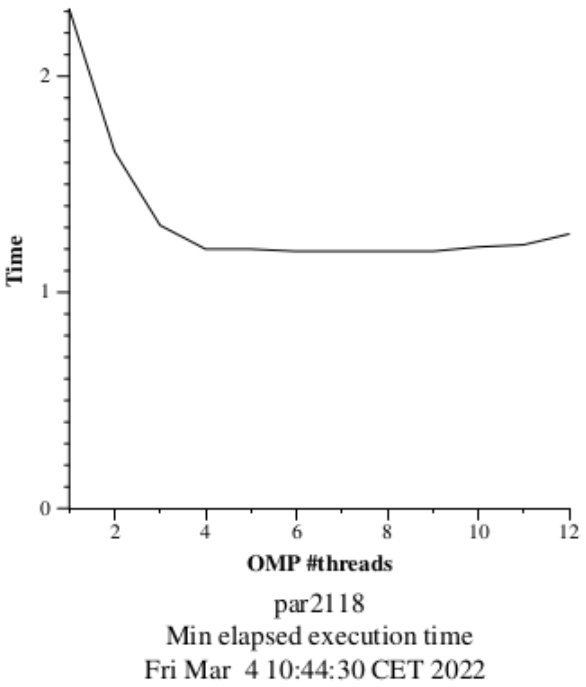

Figura 24: gráfica del tiempo de ejecución de 3dfft_omp (versión inicial)

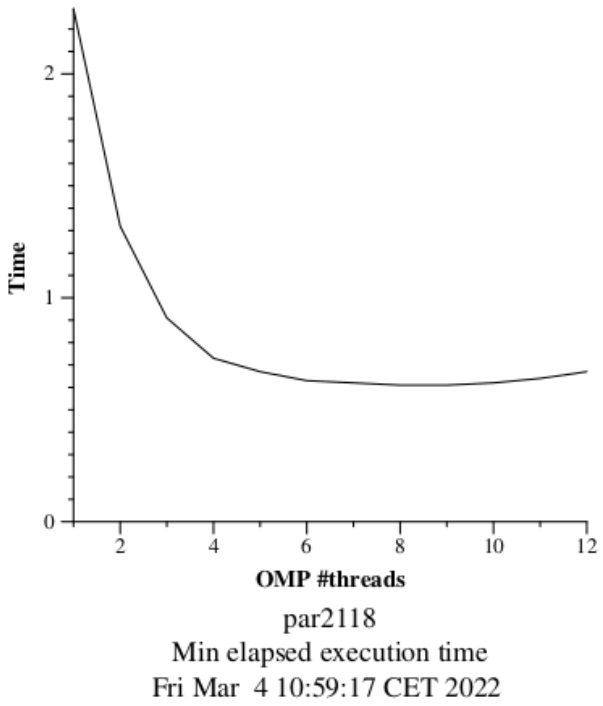

Figura 25: gráfica del tiempo de ejecución de 3dfft_omp (primera versión mejorada)

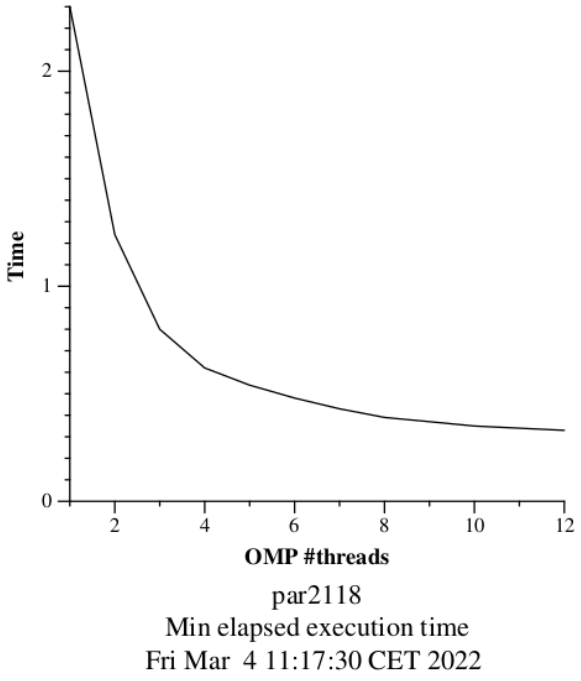

Figura 26: gráfica del tiempo de ejecución de 3dfft_omp (segunda versión mejorada)