PAR Laboratory Assignment
Lab 1: Experimental setup and tools

Natalia Dai
Daniel Ruiz Jiménez

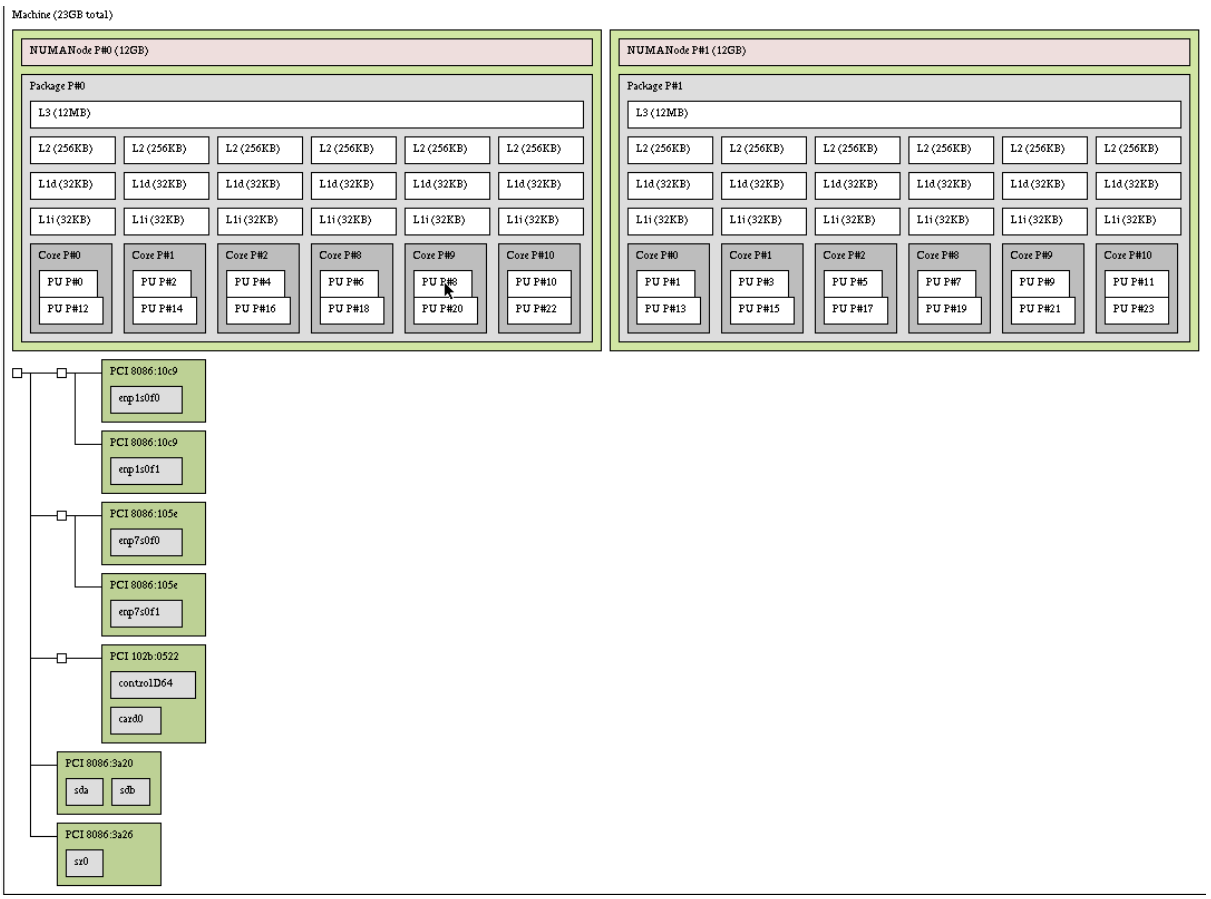
Username: par2306
Date: 9/3/2022
Spring 2021-22

1. Node architecture and memory

Durante el curso de la asignatura usaremos los servidores multiprocesador Boada, un cluster dividido en 8 nodos con diferentes arquitecturas. Los nodos que a nosotros nos interesarán en PAR son los nodos boada-1 al boada-4. El primero es un nodo interactivo, es decir, que comparte recursos con otros programas al ejecutarlo, de modo que sacrifica potencia de CPU a cambio de paralelismo, y los 3 restantes son nodos que cuentan con una cola de ejecución, es decir, que ejecutan programas aislados en orden FIFO. Cuando un nodo está disponible, ejecuta el programa correspondiente.

	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	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)	12288KB
Main memory size (per socket)	12GB
Main memory size (per node)	23GB

Habiendo ejecutado el comando lstopo - - of fig, nos generaba un fichero .fig, y al abrirlo mediante el comando xfig, se nos muestra la arquitectura del nodo boada -1 (se consideran también iguales las arquitecturas de los nodos boada-2 al boada-4).
La parte superior está dividida en dos nodos iguales, NUMANode P#0 y NUMANode P#1, los sockets del boada, cada uno con una memoria de 12GB, 6 cores por socket y con 3 niveles de caché: la de primer nivel dividida en caché de instrucciones y de datos, la de segundo nivel y por último la de tercer nivel:



2. Strong vs weak scalability

La escalabilidad se mide calculando la relación entre los tiempos de ejecución secuencial y paralelo (speed-up). Hay dos tipos de escalabilidad:

• En escalabilidad strong, el número de subprocesos se cambia con un tamaño de problema fijo. En este caso el paralelismo se utiliza para reducir el tiempo de ejecución del programa.

• En escalabilidad weak el tamaño del problema es proporcional al número de hilos. En este caso el paralelismo se utiliza para aumentar el tamaño del problema para el que se ejecuta el programa.

A continuación hemos ejecutado el programa pi_omp.c de las dos siguientes formas:
Una, de manera interactiva (columna izquierda) y la otra con el sistema de colas (columna derecha), ejecutando en cada caso 10^{12} iteraciones con distinto número de hilos de ejecución. El resultado está escrito en la siguiente tabla:

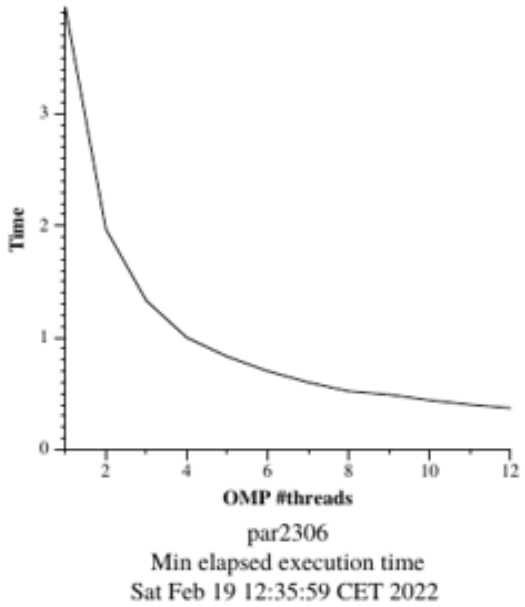
# threads	Interactive				Queued			
	user	system	elapsed	% of CPU	user	system	elapsed	% of CPU
1	3.94	0	3.95	99	3.94	0	3.95	99
2	8	0	4.01	199	3.95	0	1.99	198
4	8.02	0.21	4.12	199	4.01	0	1.01	393
8	8.04	0.06	4.06	199	5.62	0	0.72	782

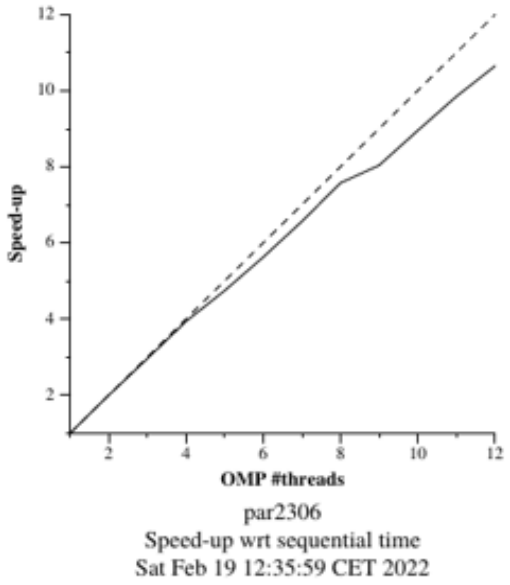
Comentando los resultados, podemos observar:

- Primero, en el modo interactivo, cuando aumentamos el número de threads, el tiempo de ejecución aumenta, para nuestra sorpresa. Esto se puede deber a que el programa esté aplicando de la forma incorrecta el paralelismo. Se puede ver en la columna "elapsed". El % de la CPU usada llega hasta el tope de 199% cuando se usan 2 hilos en adelante. La columna system no muestra una variación significativa, y en la columna user, el tiempo aumenta significativamente de 1 a 2 threads, pero luego no sube tan pronunciado.

- Segundo, en el modo de colas, se aprovecha mucho más el paralelismo, ya que el % de CPU usado es muchísimo más alto, llegando casi hasta el 800% con 8 threads. La columna system se ha mantenido en 0 durante las 4 ejecuciones. La columna de elapsed ha ido disminuyendo a medida que aumentaban los threads y la columna user ha aumentado pero no de forma tan ascendente como en el modo de ejecución anterior.

Pasamos ahora a comentar los ficheros Postscript de las ejecuciones con escalabilidad weak y strong:

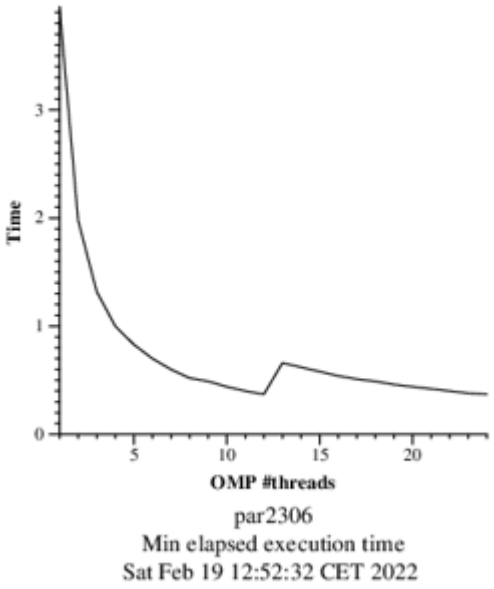


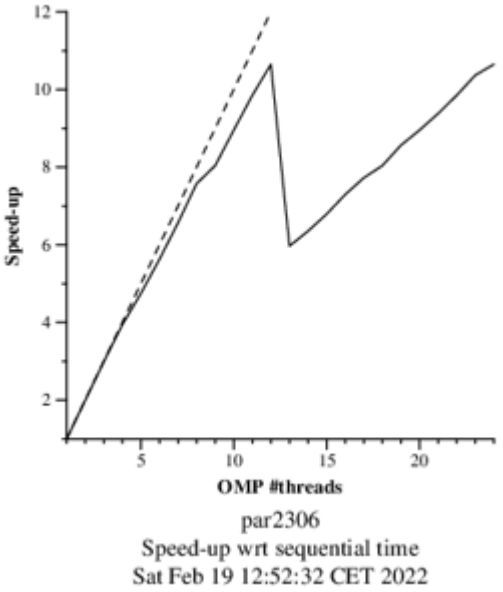


- La imagen de la izquierda muestra, de las ejecuciones en escalabilidad strong, con las variables np_MIN igual a 1 y np_MAX igual a 12 threads. La curva que representa este gráfico nos indica que al aumentar el número de threads, el tiempo de ejecución disminuye.

- La imagen de la derecha representa el speed-up de las ejecuciones respecto al número de threads. Podemos ver que la línea discontinua representa un aumento lineal esperado o deseado del speed-up, a cuya inclinación nuestra línea se aproxima. A partir de los 8 threads, el speed-up se reduce un poco, pero cuando llega a 10, se recupera.

  A continuación vemos dos imágenes más, pero esta vez poniendo como número de threads máximo 24 en lugar de 12:

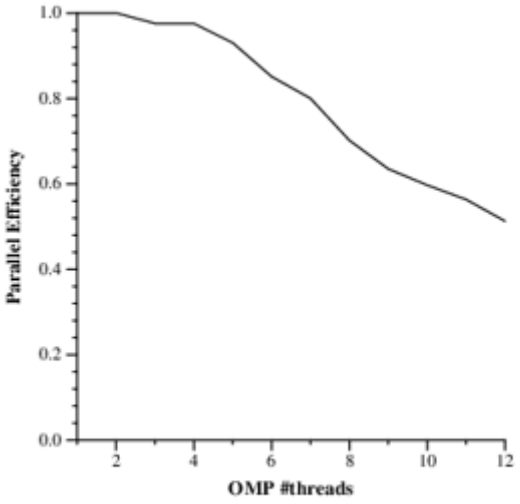




- La imagen de la izquierda muestra de nuevo una gráfica del tiempo de ejecución del programa en función del número de threads. Igual que con 12, el tiempo de ejecución baja cuando los threads suben. Sin embargo, al llegar a 12 podemos observar que sube un poco el tiempo al llegar a un punto mínimo, por alguna razón, pero de ese punto en adelante, el pendiente se vuelve a normalizar, tomando al punto mínimo como asíntota.

- La imagen de la derecha muestra el speedup en función de los threads, esta vez a los 12 threads, el speed-up baja repentinamente hasta un speed-up de 6. Más adelante vuelve a subir la curva pero con menos ángulo.

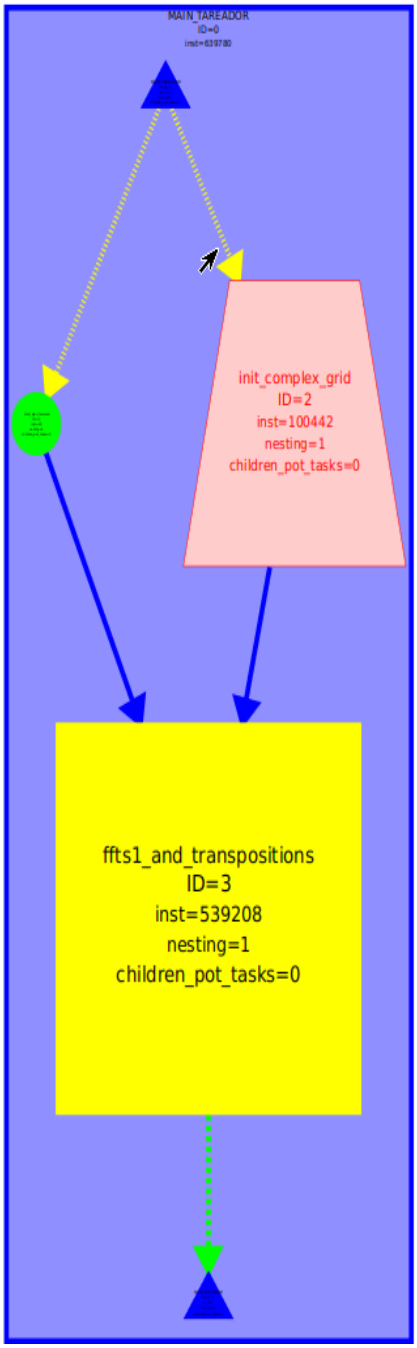
En esta última imagen podemos ver que, con la escalabilidad débil, a medida que el número de threads aumenta, la paralelización tampoco es demasiado eficiente, y el speedup baja cuando los threads suben, debido a los cálculos intermedios (overheads) y la sincronización entre hilos.



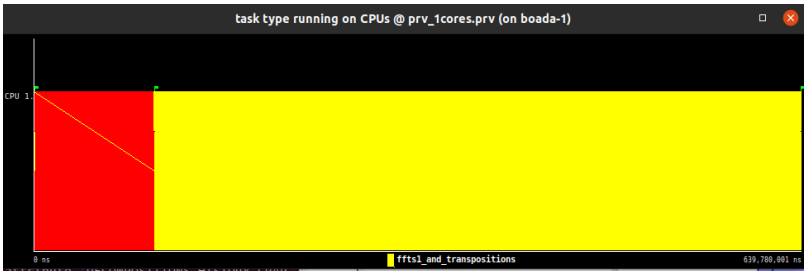
3. Systematically analysing task decompositions with Tareador

En la segunda sesión nos hemos introducido en la herramienta Tareador para simular paralelismo mediante la descomposición del código en tareas. Usando ciertas funciones en las diferentes versiones del programa 3dfft_tar.c, hemos rellenado una tabla con diferentes estrategias de paralelismo. Hemos creado el grafo de dependencias, y hemos calculado el tiempo de ejecución si lo ejecutáramos con un solo procesador y el tiempo si lo ejecutáramos con "infinitos" procesadores (hemos usado valores como el 4 en la versión menos paralelizable, ya nos era suficiente como para observar la escasa diferencia de tiempo, y el 128 en el resto de versiones).

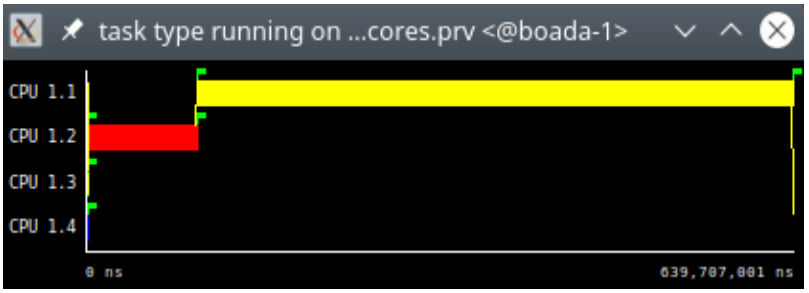
Versión 0 (secuencial):



CORE: 1

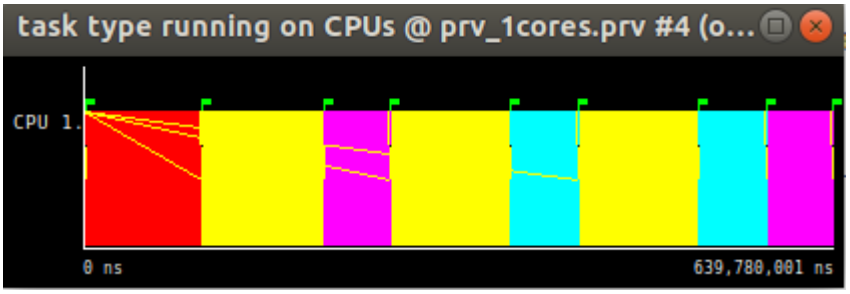


CORES: 4

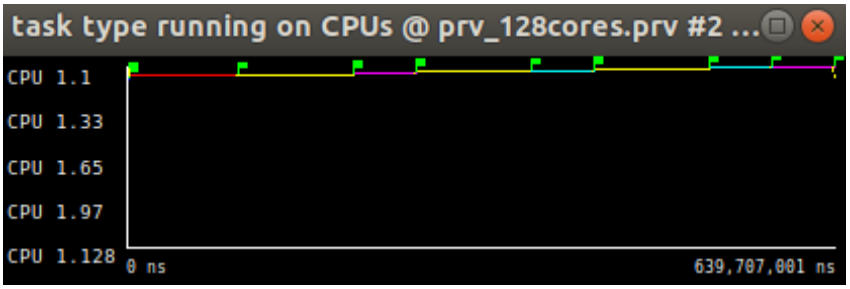


Versión 1:

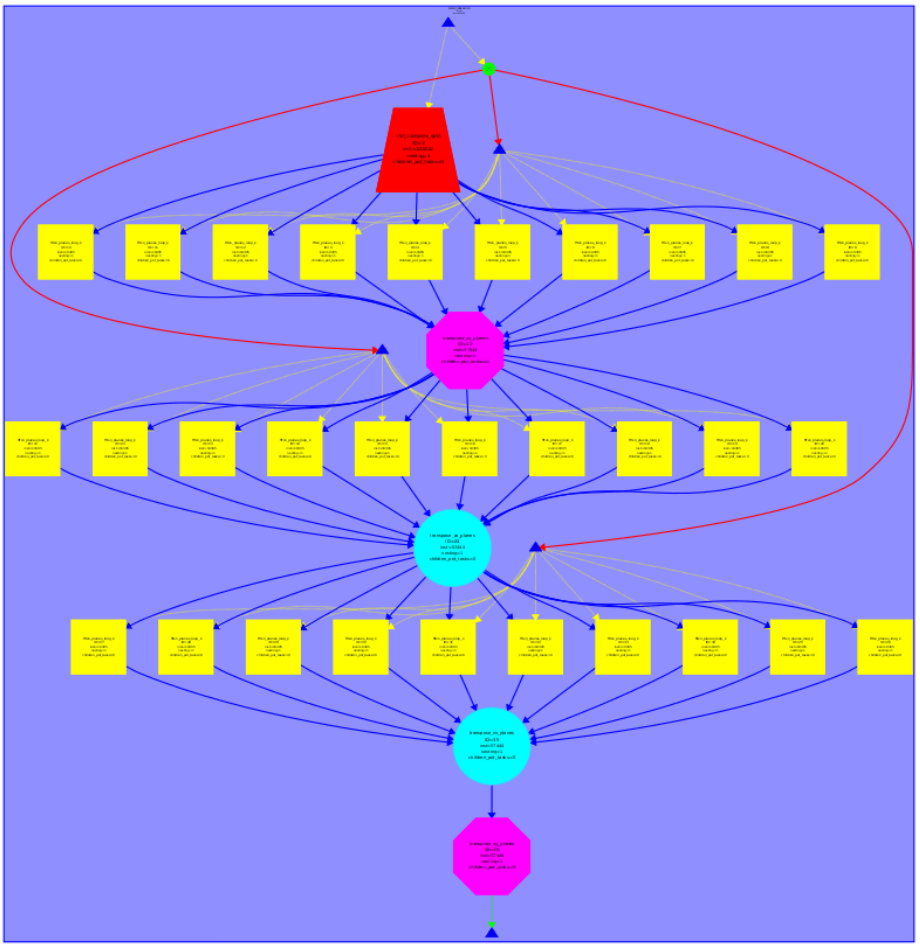
CORE: 1



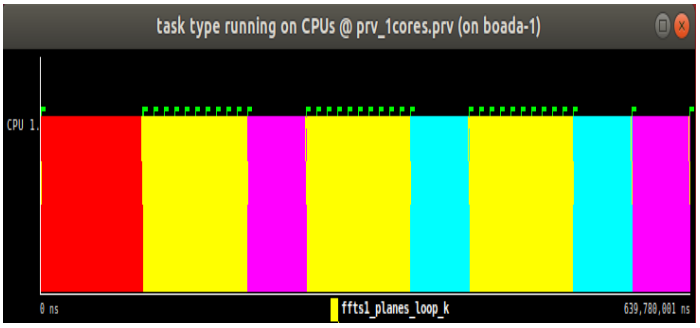
CORES: 128



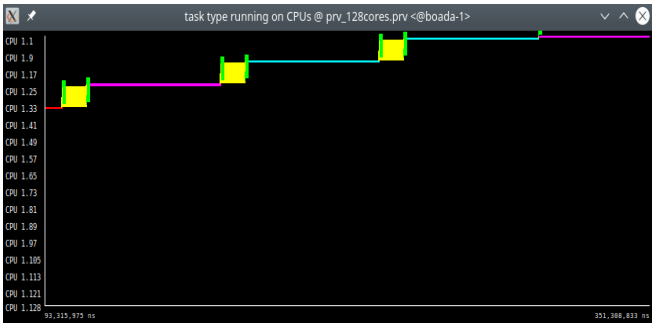
Versión 2:



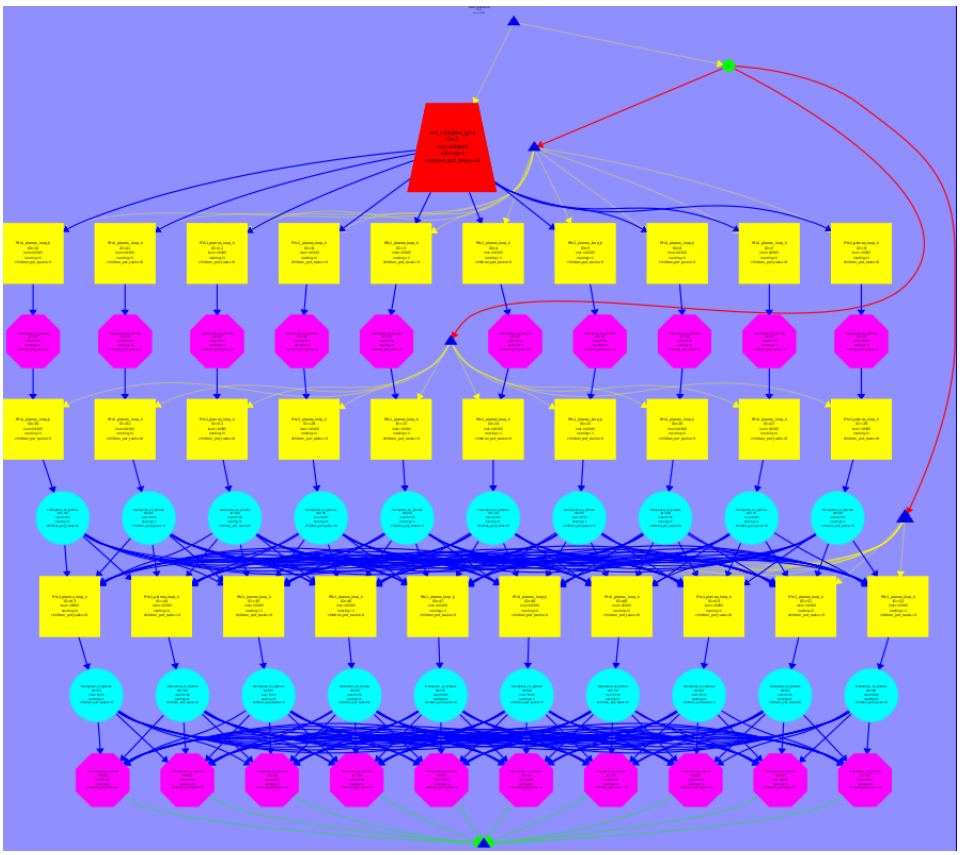
CORE: 1



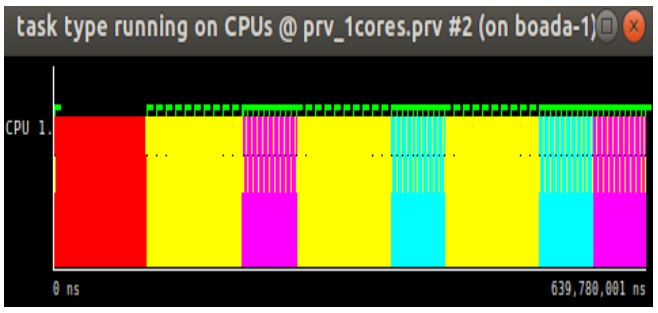
CORES: 128



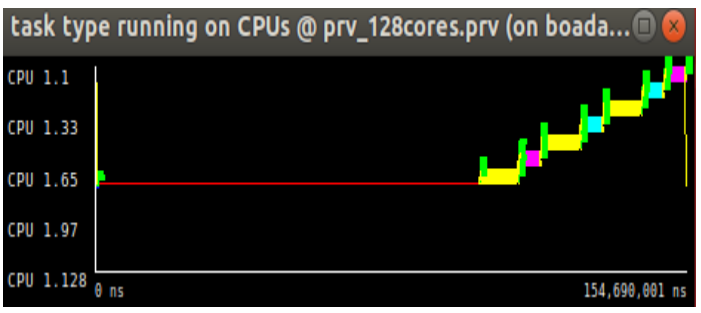
Versión 3:



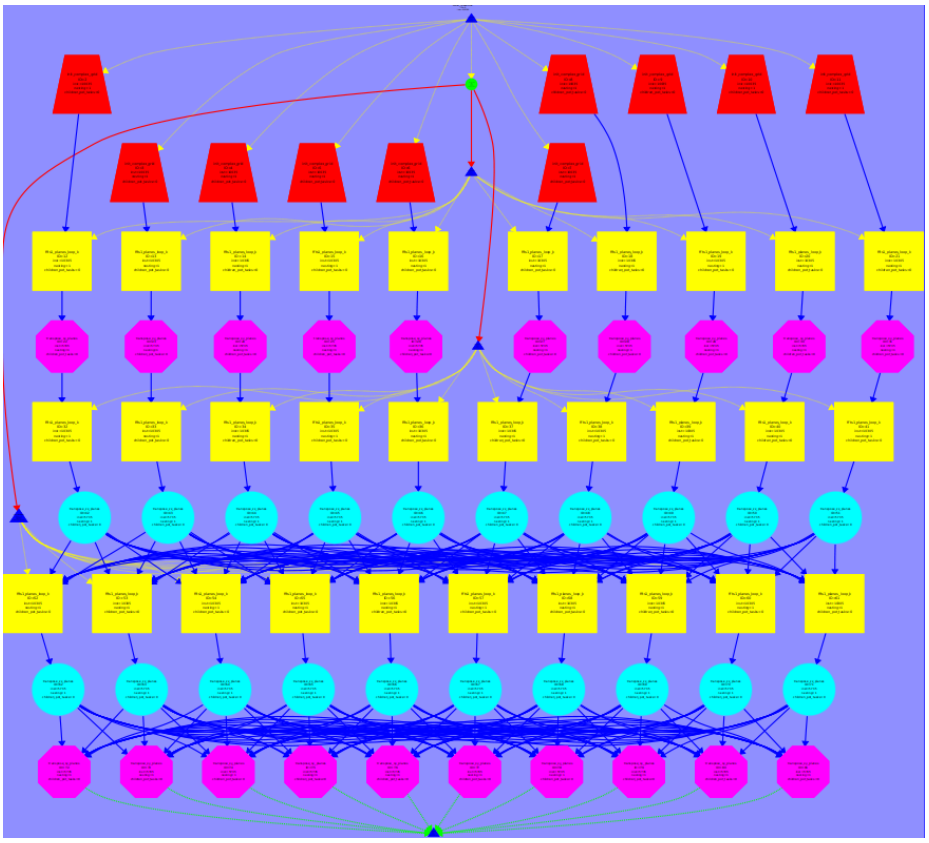
CORE: 1



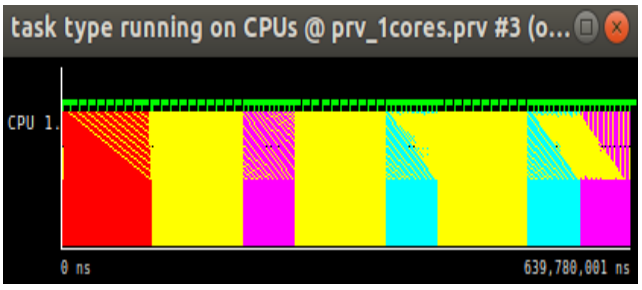
CORES: 128



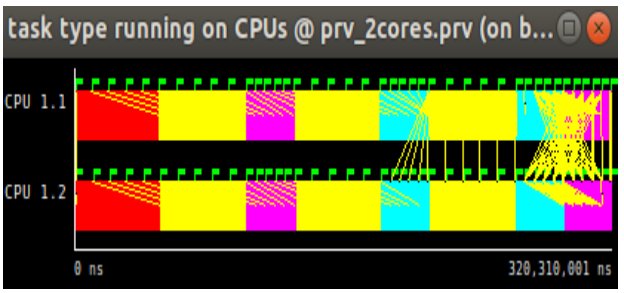
Versión 4:



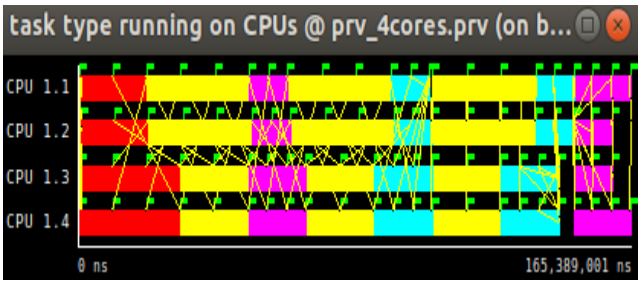
CORE: 1



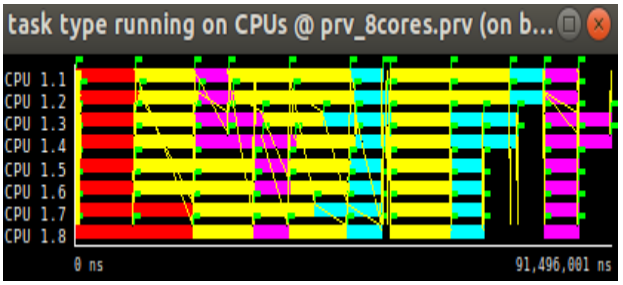
CORES: 2



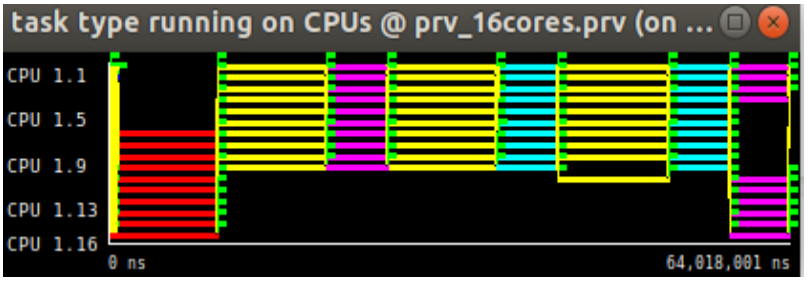
CORES: 4



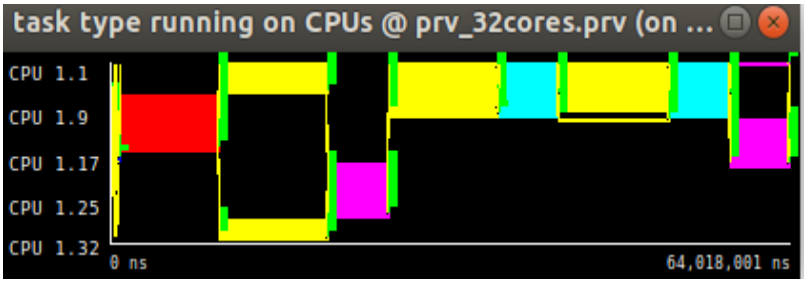
CORES: 8



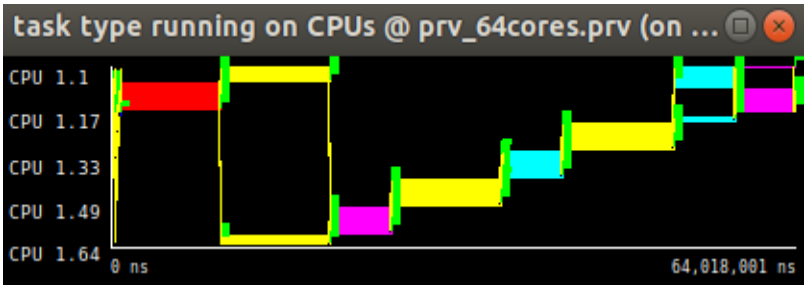
CORES: 16



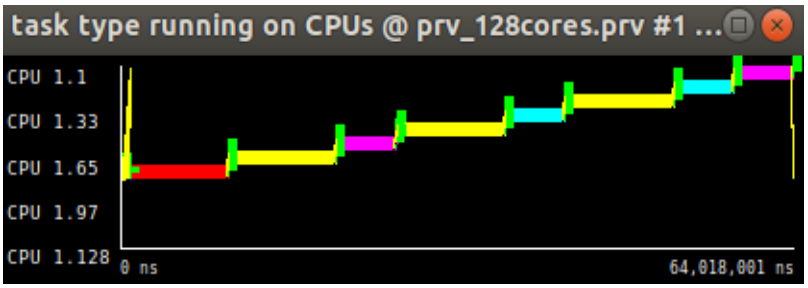
CORES: 32



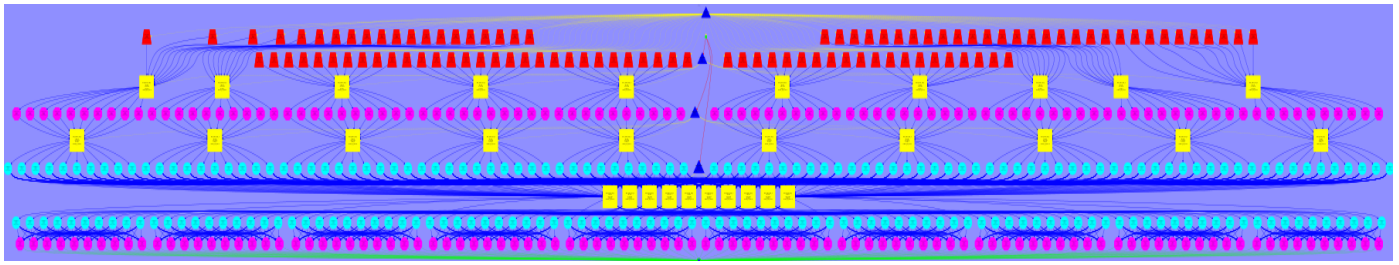
CORES: 64



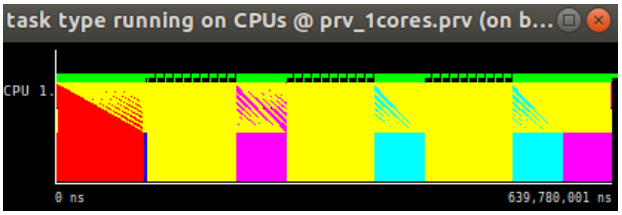
CORES: 128



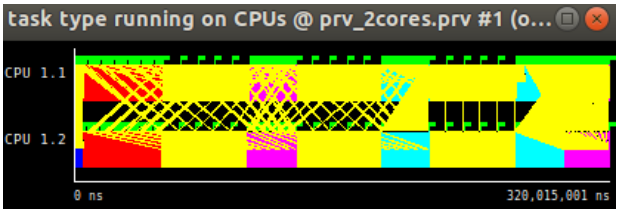
Versión 5:



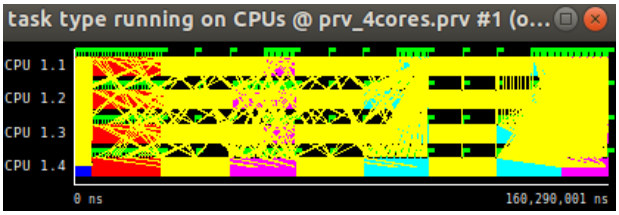
CORE: 1



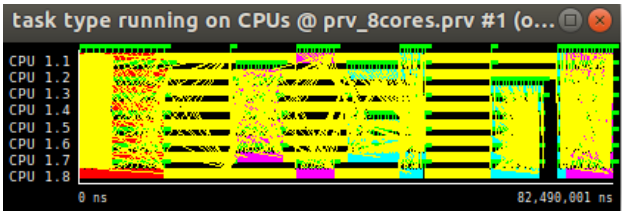
CORES: 2



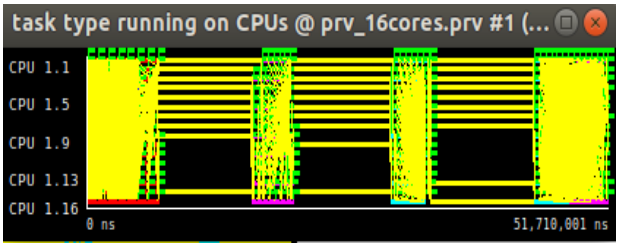
CORES: 4



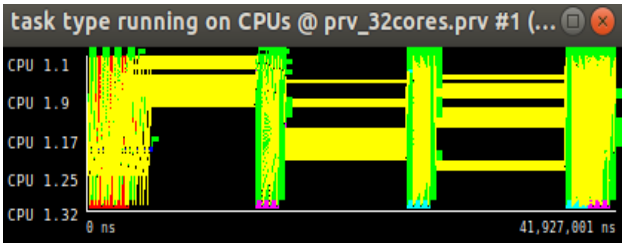
CORES: 8



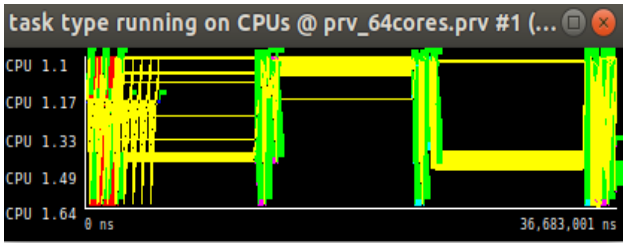
CORES: 16



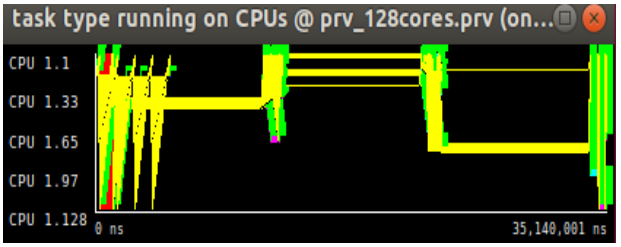
CORES: 32



CORES: 64



CORES:128

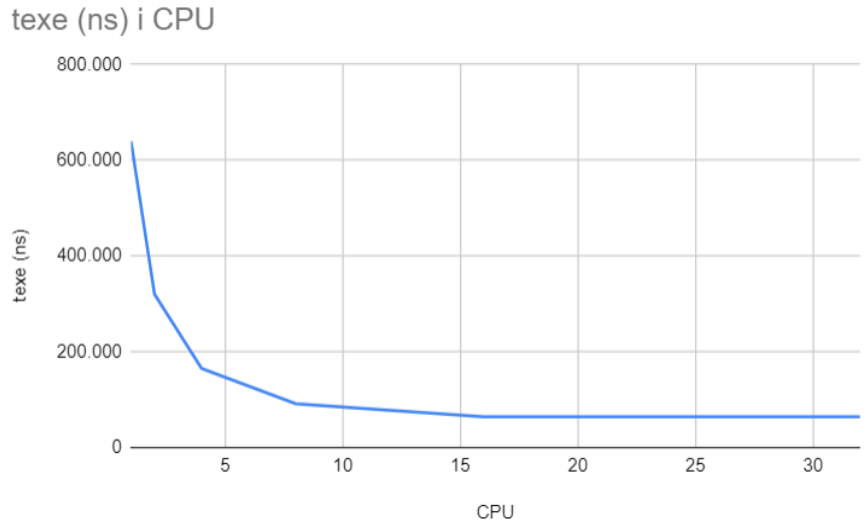


Versión	T1 (s)	T∞	Paralelismo
Secuencial	6,39780 s	6,39760 s	**1**
V1	6,39780 s	6,39760 s	**1**
V2	6,39780 s	3,51388 s	**1.82**
V3	6,39780 s	1,54690 s	**4.14**
V4	6,39780 s	0,64018 s	**9.99**
V5	6,39780 s	0,35140 s	**18.2**

Habiendo rellenado la tabla, podemos observar que el tiempo en todas las versiones con 1 procesador es el mismo, debido a que el programa sólo puede ejecutarse de forma secuencial. En la columna de los "infinitos" procesadores, para cada nueva versión, hemos ido modificando las llamadas a las funciones, para que en cada iteración del bucle externo (K) se cree una tarea nueva.

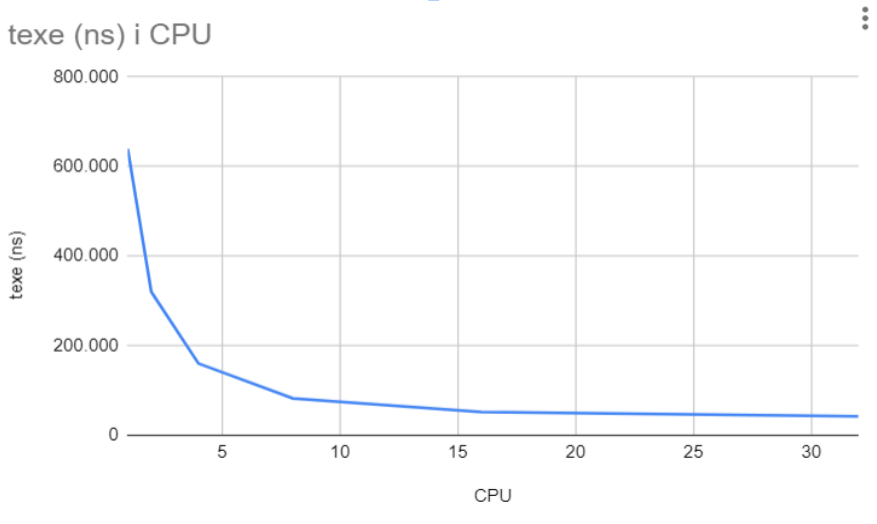
En la **primera** versión no hemos observado mucho cambio respecto a la original, el paralelismo es casi el mismo.
En la **segunda** versión hacemos que la función *fffts1_planes* se ejecute en paralelo, y observamos una reducción del tiempo con infinitos procesadores bastante sustancial, y el paralelismo se reduce.
En la **tercera** versión paralelizamos las funciones *transpose_xy_planes* y *transpose_zx_planes* de la misma forma que en la versión anterior, y la granularidad de las tareas nos permite aumentar todavía más el paralelismo.
En la **cuarta** versión hemos paralelizado la última función que nos quedaba, *init_complex_grid*. El paralelismo, como podemos observar en la tabla, ha mejorado mucho.
Por último, en la **quinta** versión, hemos cambiado la forma en la que paralelizábamos las tareas, empezando por el segundo bucle (J) en lugar del primero (K). A consecuencia de esto, se han creado muchas más tareas que han ayudado a paralelizar aún más el código. El paralelismo se ha casi duplicado respecto a la versión anterior.

Gráfico escalabilidad v4:



Este gráfico muestra cómo a medida que aumentan los procesadores con los que ejecutamos el programa, el tiempo de ejecución (en ns) disminuye. En esta versión, al llegar a 16 CPU, el tiempo pasa a ser constante. Por lo tanto $T\infty$ pasa a ser T_{16}.
Los speedups respecto a la versión con 1 procesador son:
2 CPU = 1,96x
4 CPU = 3,81x
8 CPU = 6,89x
16 CPU = 9,85x
32 CPU = 9,85x

Gráfico escalabilidad v5:



En este gráfico vemos una forma de la gráfica parecida a la anterior, pero con valores mucho inferiores. Sin embargo, no podemos ver todavía T∞, ya que el la línea se va acercando más y más al 0. Observamos que el cambio de versión mejora en gran medida los tiempos.
Los speedups respecto a la versión con 1 procesador son:
2 CPU = 1,97x
4 CPU = 3,94x
8 CPU = 7,64x
16 CPU = 12,19x
32 CPU = 15,04x

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

   for (k = 0; k < N; k++) {
     for (j = 0; j < N; j++) {
         tareador_start_task("init_complex_grid");
         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.
             in_fftw[k][j][i][1] = 0;
#if TEST
         out_fftw[k][j][i][0]= in_fftw[k][j][i][0];
         out_fftw[k][j][i][1]= in_fftw[k][j][i][1];
#endif
         }
         tareador_end_task("init_complex_grid");
     }
   }
}

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

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

La versión 5 es capaz de escalar más que la versión 4 debido a la posición de las llamadas de comienzo de tareas. En la quinta están en el bucle J mientras que en la cuarta están en el bucle K, el más externo. Como vemos en los grafos de dependencias, la versión 5 es capaz de escalar a un número mucho mayor de procesadores, por lo tanto es capaz de realizar un número mayor de tareas en paralelo. En cambio, debido al número de dependencias que tiene la versión 4, nunca podrá llegar a aprovechar los procesadores.

4. Understanding the parallel execution of 3DFFT

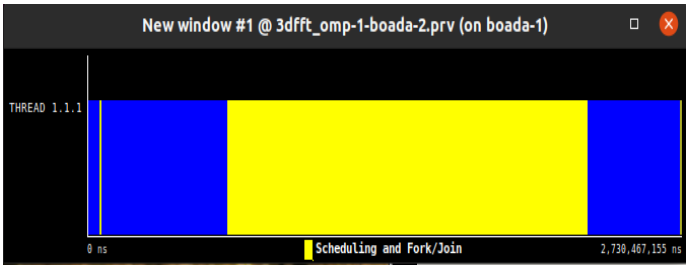
En esta tercera sesión hemos profundizado en el uso de la herramienta Paraver que vimos en el ejercicio anterior, para analizar las tareas de programas secuenciales, pero esta vez con el programa 3DFFT en su versión paralela. Además, hemos hecho una toma de contacto con la librería Extrae, que nos proporciona unos archivos binarios con la información de la ejecución del programa (llamados traces). El programa que analizamos es 3dfft_omp.c, y lo hemos ejecutado de 3 formas diferentes:

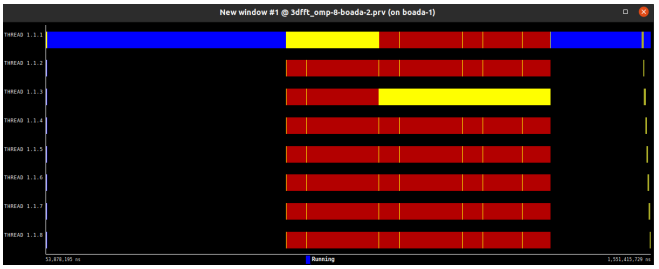
-La primera, con el archivo intacto.

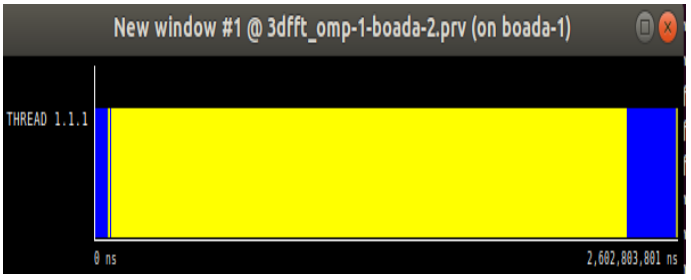
-La segunda, paralelizando la función init_complex_grid con funciones #pragma.

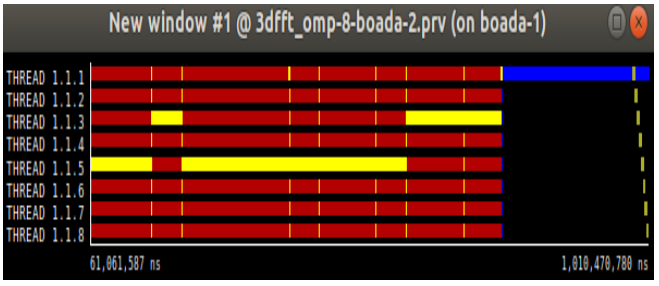
-La tercera, paralelizando el resto del programa y cambiando la posición de la paralelización del bucle para reducir los overheads de creación de tarea.

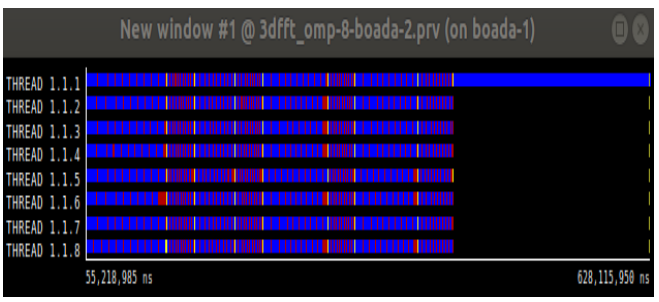
Las siguientes imágenes nos muestran las ejecuciones de las tres versiones, la columna izquierda con 1 único thread y la derecha con 8 threads:











Utilizando Paraver, hemos calculado la fracción paralela del programa, el tiempo de ejecución con 1 thread y el tiempo de ejecución con 8, el speedup ideal y el real, usando las siguientes fórmulas:

$$\boldsymbol{T_1} = Tseq + Tpar$$
$$\varphi = Tpar / T_1$$
$$\boldsymbol{S_8\ real} = T_1 / T_8$$
$$\boldsymbol{S_8 ideal} = 1 / (1 - \varphi + (\varphi / p)$$

Versión	φ	S_8 ideal	T_1 (ns)	T_8 (ns)	S_8 real
Inicial	0.6	2.1	2.730.467.155	1.497.537.534	1.82
Nueva versión	0.86	4.04	2.602.803.801	949.409.193	2.74
Versión final	0.906	4.825	2.575.307.628	572.896.965	4.49

La **primera** versión es la más simple, por lo tanto el tiempo de ejecución es el más grande de todos, y su speed-up el más pequeño.
La **segunda** versión ya cuenta con trazas en la función init_complex_grid, por lo que su fracción paralela ha aumentado, lo que conlleva una reducción del tiempo de ejecución y un aumento de su speedup.
En la **tercera** versión hemos comentado y descomentado pragmas a lo largo de todo el código, pero lo más importante es que ahora se generan trazas fuera del bucle K, por lo que se reducen los overheads de creación de tareas y de sincronización. Esto conlleva a una mejoría significativa del paralelismo en el programa, y eso se demuestra en los tiempos de ejecución y los speedups.

<u>Histogramas de las 3 versiones:</u>

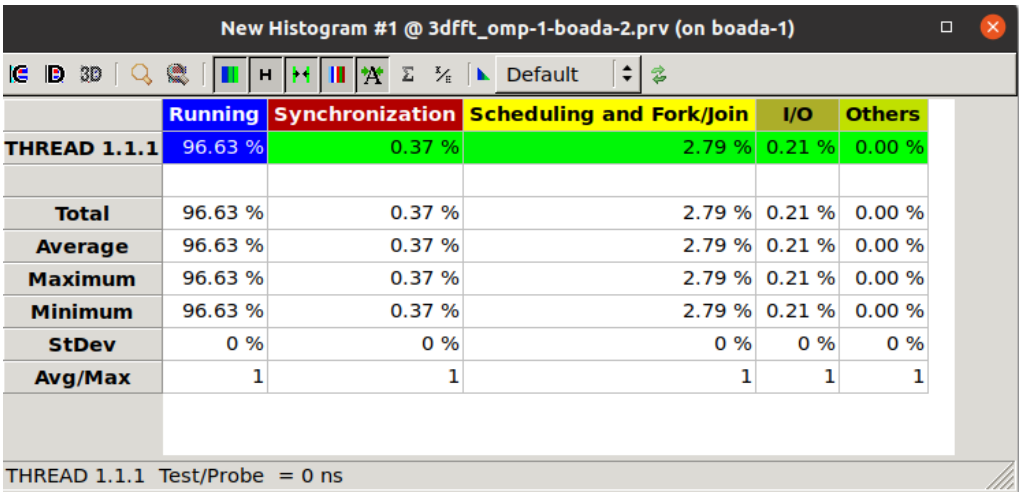
Versión inicial con 1 thread

	Running	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	96.99 %	0.31 %	2.52 %	0.17 %	0.00 %
Total	96.99 %	0.31 %	2.52 %	0.17 %	0.00 %
Average	96.99 %	0.31 %	2.52 %	0.17 %	0.00 %
Maximum	96.99 %	0.31 %	2.52 %	0.17 %	0.00 %
Minimum	96.99 %	0.31 %	2.52 %	0.17 %	0.00 %
StDev	0 %	0 %	0 %	0 %	0 %
Avg/Max	1	1	1	1	1

Versión inicial con 8 threads

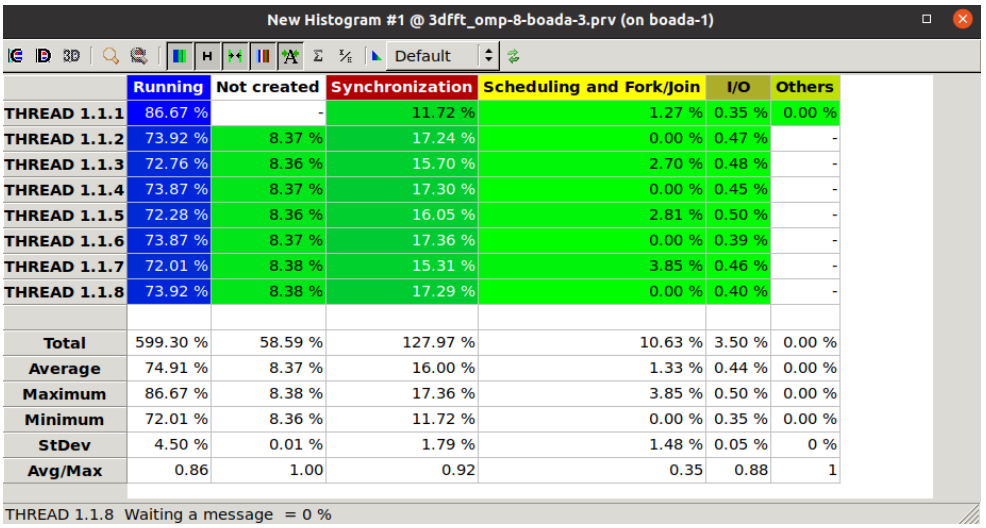
	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	91.65 %	-	7.14 %	0.99 %	0.22 %	0.00 %
THREAD 1.1.2	74.85 %	9.05 %	15.67 %	0.00 %	0.43 %	-
THREAD 1.1.3	67.90 %	9.03 %	13.53 %	8.87 %	0.67 %	-
THREAD 1.1.4	74.95 %	9.06 %	15.60 %	0.00 %	0.39 %	-
THREAD 1.1.5	74.76 %	9.06 %	15.83 %	0.00 %	0.34 %	-
THREAD 1.1.6	75.00 %	9.07 %	15.57 %	0.00 %	0.35 %	-
THREAD 1.1.7	74.74 %	9.07 %	15.84 %	0.00 %	0.35 %	-
THREAD 1.1.8	74.96 %	9.09 %	15.61 %	0.00 %	0.34 %	-
Total	608.81 %	63.43 %	114.80 %	9.87 %	3.08 %	0.00 %
Average	76.10 %	9.06 %	14.35 %	1.23 %	0.39 %	0.00 %
Maximum	91.65 %	9.09 %	15.84 %	8.87 %	0.67 %	0.00 %
Minimum	67.90 %	9.03 %	7.14 %	0.00 %	0.22 %	0.00 %
StDev	6.31 %	0.02 %	2.82 %	2.90 %	0.12 %	0 %
Avg/Max	0.83	1.00	0.91	0.14	0.58	1

Versión 2 con 1 thread



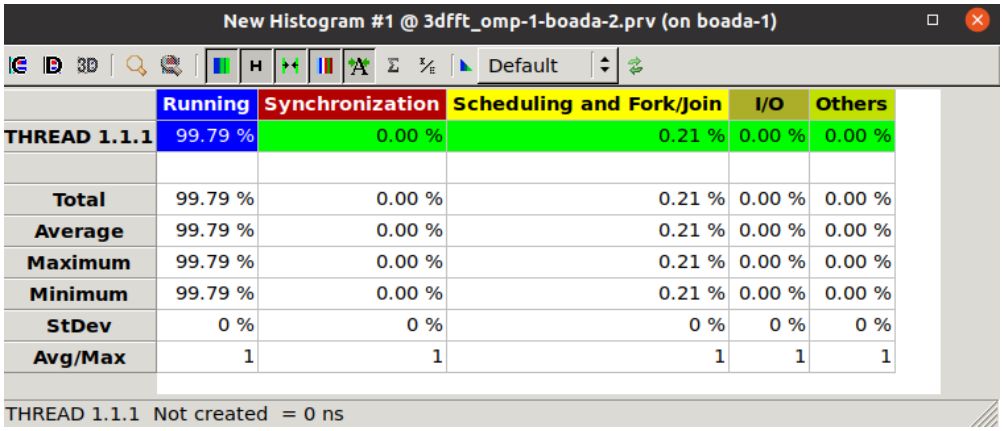
	Running	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	96.63 %	0.37 %	2.79 %	0.21 %	0.00 %
Total	96.63 %	0.37 %	2.79 %	0.21 %	0.00 %
Average	96.63 %	0.37 %	2.79 %	0.21 %	0.00 %
Maximum	96.63 %	0.37 %	2.79 %	0.21 %	0.00 %
Minimum	96.63 %	0.37 %	2.79 %	0.21 %	0.00 %
StDev	0 %	0 %	0 %	0 %	0 %
Avg/Max	1	1	1	1	1

Versión 2 con 8 threads



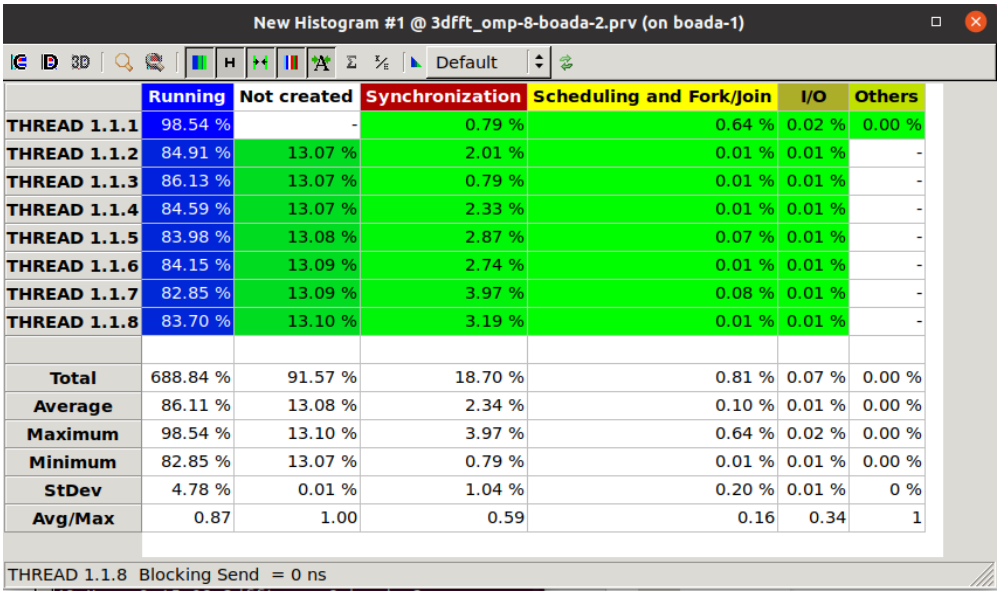
	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	86.67 %	-	11.72 %	1.27 %	0.35 %	0.00 %
THREAD 1.1.2	73.92 %	8.37 %	17.24 %	0.00 %	0.47 %	-
THREAD 1.1.3	72.76 %	8.36 %	15.70 %	2.70 %	0.48 %	-
THREAD 1.1.4	73.87 %	8.37 %	17.30 %	0.00 %	0.45 %	-
THREAD 1.1.5	72.28 %	8.36 %	16.05 %	2.81 %	0.50 %	-
THREAD 1.1.6	73.87 %	8.37 %	17.36 %	0.00 %	0.39 %	-
THREAD 1.1.7	72.01 %	8.38 %	15.31 %	3.85 %	0.46 %	-
THREAD 1.1.8	73.92 %	8.38 %	17.29 %	0.00 %	0.40 %	-
Total	599.30 %	58.59 %	127.97 %	10.63 %	3.50 %	0.00 %
Average	74.91 %	8.37 %	16.00 %	1.33 %	0.44 %	0.00 %
Maximum	86.67 %	8.38 %	17.36 %	3.85 %	0.50 %	0.00 %
Minimum	72.01 %	8.36 %	11.72 %	0.00 %	0.35 %	0.00 %
StDev	4.50 %	0.01 %	1.79 %	1.48 %	0.05 %	0 %
Avg/Max	0.86	1.00	0.92	0.35	0.88	1

Versión 3 con 1 thread



	Running	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	99.79 %	0.00 %	0.21 %	0.00 %	0.00 %
Total	99.79 %	0.00 %	0.21 %	0.00 %	0.00 %
Average	99.79 %	0.00 %	0.21 %	0.00 %	0.00 %
Maximum	99.79 %	0.00 %	0.21 %	0.00 %	0.00 %
Minimum	99.79 %	0.00 %	0.21 %	0.00 %	0.00 %
StDev	0 %	0 %	0 %	0 %	0 %
Avg/Max	1	1	1	1	1

Versión 3 con 8 threads:

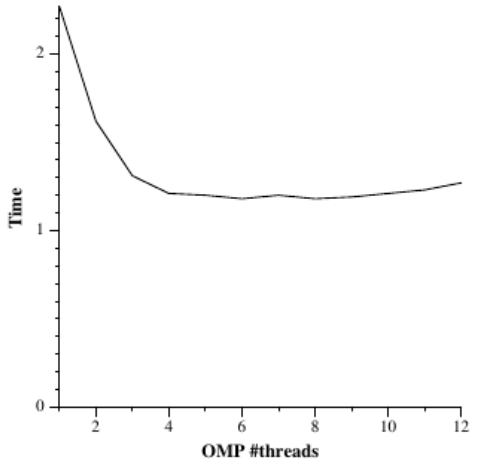


	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	98.54 %	-	0.79 %	0.64 %	0.02 %	0.00 %
THREAD 1.1.2	84.91 %	13.07 %	2.01 %	0.01 %	0.01 %	-
THREAD 1.1.3	86.13 %	13.07 %	0.79 %	0.01 %	0.01 %	-
THREAD 1.1.4	84.59 %	13.07 %	2.33 %	0.01 %	0.01 %	-
THREAD 1.1.5	83.98 %	13.08 %	2.87 %	0.07 %	0.01 %	-
THREAD 1.1.6	84.15 %	13.09 %	2.74 %	0.01 %	0.01 %	-
THREAD 1.1.7	82.85 %	13.09 %	3.97 %	0.08 %	0.01 %	-
THREAD 1.1.8	83.70 %	13.10 %	3.19 %	0.01 %	0.01 %	-
Total	688.84 %	91.57 %	18.70 %	0.81 %	0.07 %	0.00 %
Average	86.11 %	13.08 %	2.34 %	0.10 %	0.01 %	0.00 %
Maximum	98.54 %	13.10 %	3.97 %	0.64 %	0.02 %	0.00 %
Minimum	82.85 %	13.07 %	0.79 %	0.01 %	0.01 %	0.00 %
StDev	4.78 %	0.01 %	1.04 %	0.20 %	0.01 %	0 %
Avg/Max	0.87	1.00	0.59	0.16	0.34	1

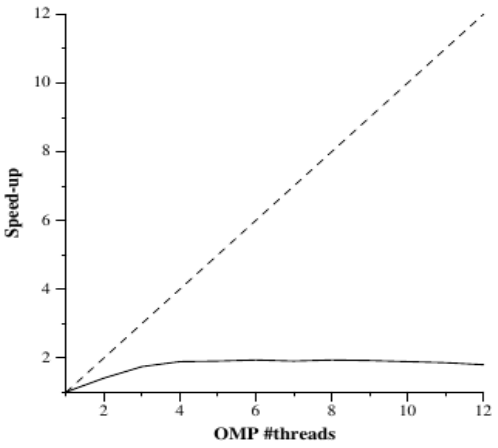
Pasemos ahora a ver los plots de escalabilidad y speedup de las ejecuciones de las tres versiones:

En la **primera** versión vemos un speedup que rápidamente se aleja del caso ideal, puesto que no se paraleliza en gran medida. Su tiempo de ejecución se reduce conforme los threads aumentan pero no es una gran mejora, incluso vuelve a subir un poco.
La **segunda** versión vemos mejoría en el tiempo de ejecución pero decepcionantemente no observamos un cambio muy notable en su speedup, ya que sólo hemos paralelizado parte del código.

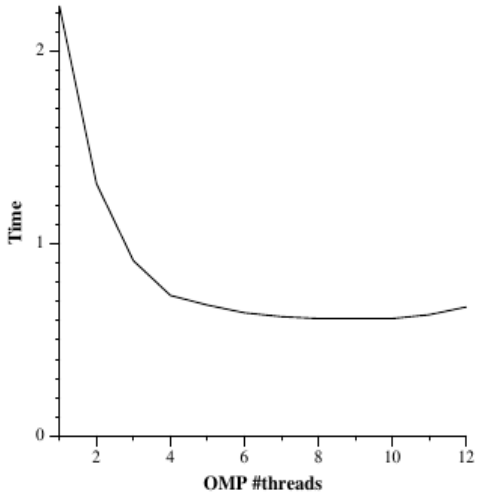
Por otro lado, en la **última** versión vemos una gran mejora tanto en tiempo de ejecución como en rendimiento, cuyo speedup está muy parejo con el caso ideal.



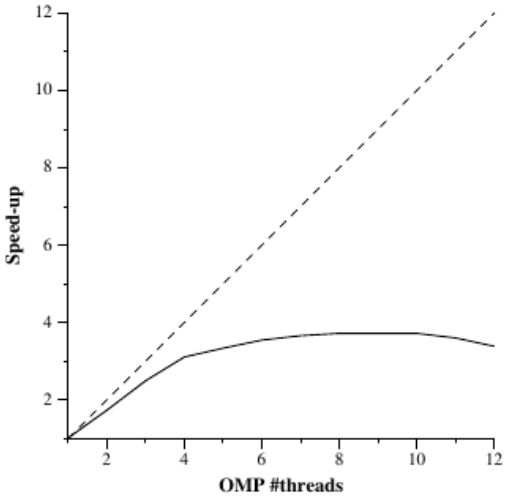
par2306
Min elapsed execution time
Sat Mar 5 17:10:55 CET 2022



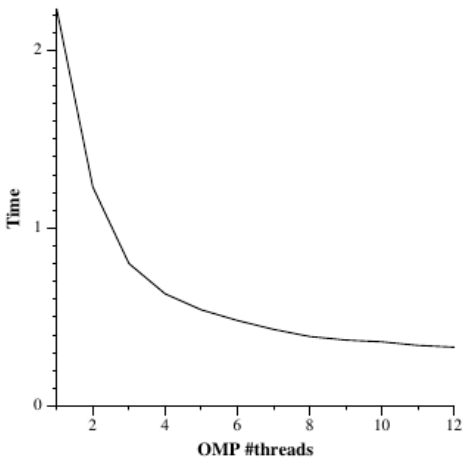
par2306
Speed-up wrt sequential time
Sat Mar 5 17:10:55 CET 2022



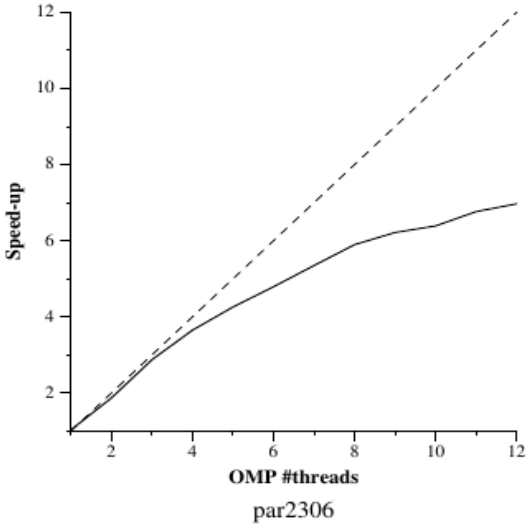
par2306
Min elapsed execution time
Sat Mar 5 17:35:02 CET 2022



par2306
Speed-up wrt sequential time
Sat Mar 5 17:35:02 CET 2022



par2306
Min elapsed execution time
Sat Mar 5 18:01:20 CET 2022



par2306
Speed-up wrt sequential time
Sat Mar 5 18:01:20 CET 2022