Parallelism

Lab 1: Experimental setup and tools

Marian Danci y David Valero

Grupo 42, par4207

Fall 2018-19

Sesión 1: Experimental setup

1.1Node architecture and memory

Primero de todo, hemos investigado la arquitectura de los diferentes nodos en el Boada. Para ello hemos ejecutado lscpu y lstopo, de este último hemos obtenido las siguientes imágenes que nos dan una idea de la estructura que tienen cada uno de los grupos.

Ilustración 1 Arquitectura de los Boada 1-4 obtenido con lstopo

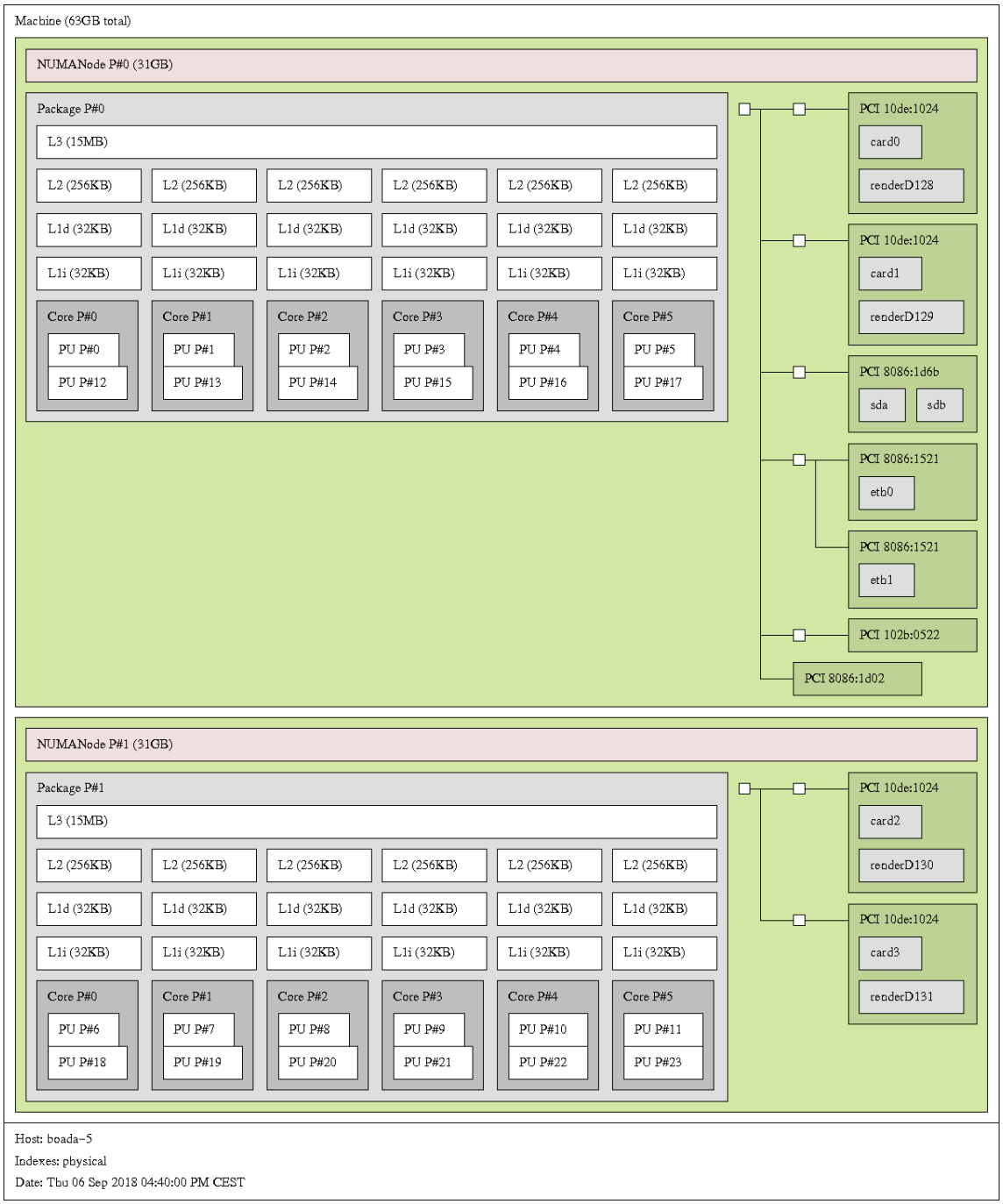

Ilustración 2 Arquitectura del Boada 5 obtenido con lstopo

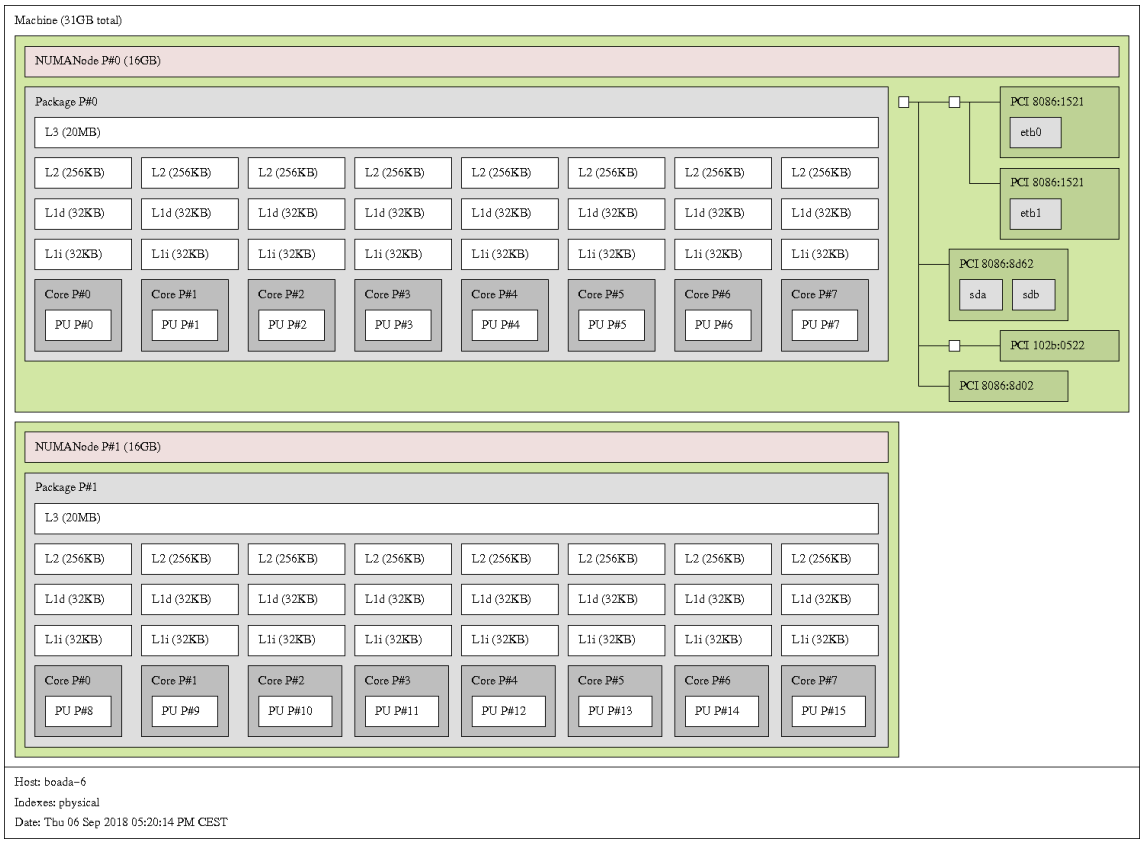

Ilustración 3 Arquitectura de los Boada 6-8 obtenido con lstopo

Finalmente, gracias a las tablas anteriores hemos obtenido los siguientes datos:

	Boada 1-4	*Boada 5*	*Boada 6-8*
Number of sockets per node	2	2	2
Number of cores per socket	6	6	8
Number of threads per core	2	2	1
Maximum core frequency	2395 MHz	2.6 GHz	1.7 GHz
L1-I cache size (per-core)	32kB	32kB	32kB
L1-D cache size (per-core)	32kB	32kB	62kB
L2 cache size (per-core)	256kB	256kB	256kB
Last-level cache size (per-socket)	12MB	15MB	20MB
Main memory size (per socket)	23GB	636B	316B
Main memory size (per node)	12GB	316B	166B

1.2 Serial compilation and execution

Hemos visto que para calcular el número Pi tenemos dos maneras de ejecutar el mismo programa:

-Mediante run-seq.sh: Este se ejecuta de manera interactiva (comparte recursos con el resto de programas, por lo que el tiempo puede que no sea representativo) y en el Boada 1.

-Mediante submit-seq.sh: Este se ejecuta en la cola (se ejecuta de manera aislada de los otros programas, cosa que permite obtener un tiempo más realista) y en el Boada 2-8.

1.3 Compilation and execution of OpenMP programs

En este apartado hemos aprendido como se crean versiones en paralelo OpenMP.

```
static long num_steps = 100000;
void main () {
    double x, pi, step, sum = 0.0;

    step = 1.0/(double) num_steps;
    // omp_set_num_threads(8);
    #pragma omp parallel private(x) reduction(+: sum) // num_threads(8)
    {
        long int myid = omp_get_thread_num();
        long int howmany = omp_get_num_threads();
        for (long int i=myid; i<num_steps; i+=howmany) {
            x = (i+0.5)*step;
            sum += 4.0/(1.0+x*x);
        }
    }
    pi = step * sum;
}
```

Ilustración 4 Versión paralela OpenMP parar el código pi_omp.c

Para hacer paralelo el programa que calcula el número Pi se ha hecho de la siguiente manera:

A partir de la línea donde aparece *#pragma*, que crea una variable privada x donde pondrá el valor obtenido en esa iteración y una variable sum que al final se acaban sumando todos los sum de cada uno de los hilos de ejecución.

Para calcularlo, en cada hilo se pregunta su id (myid) y el número de hilos totales (howmany).

1.3.1

Hemos ejecutado el programa del punto anterior en diferentes circunstancias, ejecutándolo en modo interactivo y aislado con 1 hilo de ejecución y 8 hilos cada uno. El número de hilos se especifica en el script con la variable *OMP_NUM_THREADS=$3*.

Se han obtenido los siguientes resultados:

En modo interactivo:

-En 1 hilo ha tardado 3,95s en ejecutarlo, gastando 3,94s de tiempo de usuario y 0,01s de tiempo de sistema, consumiendo 99% de la CPU.

-En 8 hilos ha tardado 0,75s en ejecutarlo, gastando 5,84 de tiempo de usuario y 0,09s de tiempo de sistema, consumiendo 790% de la CPU.

En modo aislado:

-En 1 hilo ha tardado 3,96s en ejecutarlo, gastando 3,94s de tiempo de usuario y 0,00s de tiempo de sistema, consumiendo 99% de la CPU.

-En 8 hilos ha tardado 0,58 en ejecutarlo, gastando 4,48 de tiempo de usuario y 0,01s de tiempo de sistema, consumiendo 772% de la CPU.

Con estos resultados podemos comprobar que en el modo interactivo ha tardado más, ya que mientras ejecutaba nuestro programa también podía estar ejecutando otros, por ello el tiempo de sistema no es 0s, notándose más conforme con más hilos lo ejecutemos.

Por todo ello, el modo aislado es el más fiable, ya que el tiempo es más realista, tanto el tiempo empleado con el programa como el tiempo de usuario o sistema. A más a más, conforme aumenta el grado de paralelismo del programa, consume menos CPU que en modo interactivo, porque los cambios de ambiente entre programas gastan CPU y tiempo de sistema.

1.4 Strong vs. weak scalability

En este apartado hemos comparado la escalabilidad fuerte con la débil en los diferentes nodos del Boada.

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

En escalabilidad débil, el tamaño del problema es proporcional al número de subprocesos. En este caso, el paralelismo se usa para aumentar el tamaño del problema para el que se ejecuta el programa.

Para ello, hemos ejecutado el programa en los tres tipos de nodos de Boada y con los dos tipos de escalabilidad, obteniendo los siguientes gráficos de Speed-Up:

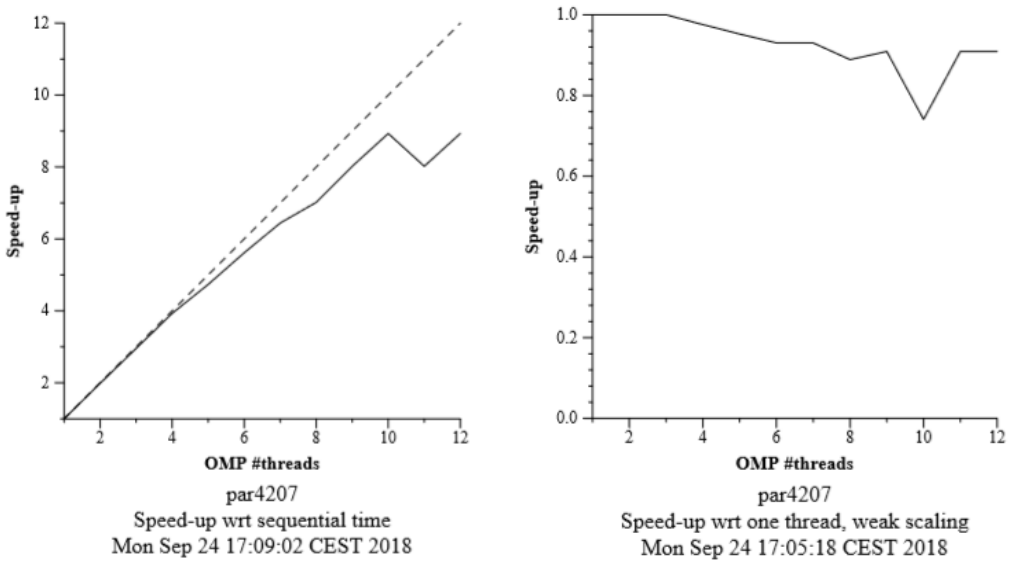

Ilustración 5 Comparación de Speed-up en el Boada 2 con escalabilidad fuerte (izq.), Boada 3 con débil (der.)

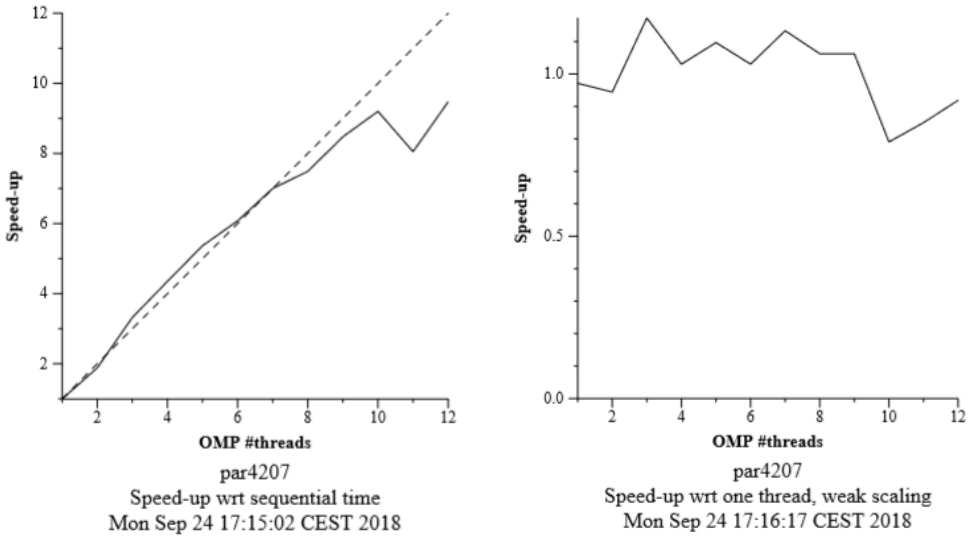

Ilustración 6 Comparación de Speed-up en el Boada 5 con escalibilidad fuerte (izq.) y escalibilidad débil (der.)

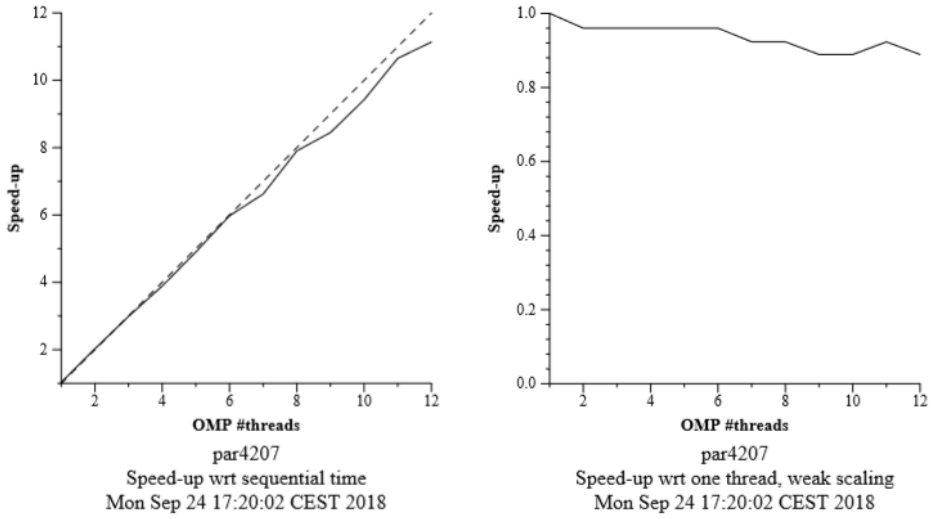

Ilustración 7 Comparación de Speed-up en el Boada 7 con escalabilidad fuerte (izq.), Boada 6 con débil (der.)

En ellos podemos observar que con la escalabilidad fuerte el Speed-Up es mayor conforme más hilos tenga el programa, en cambio, con la escalabilidad débil se mantiene constante.

Hemos detectado que con el Boada 6-8 hay mayor constancia en las dos escalabilidades y también un mayor Speed-Up máximo en la fuerte.

No hemos encontrado grandes diferencias entre los Boada 1-4 y Boada 5, únicamente destacar que el Boada 5 es más irregular en el Speed-Up en la escalabilidad débil y que en ambos casos encontramos una bajada de rendimiento con 11 hilos que no hemos conseguido explicar.

Para descubrir si era algo recurrente o una excepción en la ejecución, hemos vuelto a ejecutar el programa en ambas escalabilidades tanto en el Boada 1-4 como en el Boada 5 y hemos obtenido lo mismo, aunque en diferente número de hilos.

Sesión 2: Systematically analyzing task decompositions with Tareador

Esta sesión trataba de entender cómo funciona la aplicación *Tareador*, que permite similar como se ejecutaría un programa en tantos cores como se desee.

En todos los casos hemos trabajado con el programa 3dfft_tar.c, que trabaja con muchos bucles.

Primeramente hemos ejecutado el código tal cual lo hemos encontrado en la carpeta, obteniendo un tiempo de ejecución de 639ms para 1 core. Destacar que con 1 core no hay paralelismo posible, por lo tanto se ejecuta linealmente, como podemos ver en la ilustración 8. En todas las versiones posteriores hemos obtenido el mismo tiempo con 1 core, ya que todas las mejoras han sido en la paralización del programa.

Con 8 cores ya hemos obtenido un T_∞, al ser un programa que no aprovecha nada del paralelismo de la arquitectura, ha obtenido el mismo tiempo que con 1 core.

Ilustración 8 Gráfico obtenido con 1 core ejecutando la versión 1

En la primera versión hemos añadido una tarea para cada una de las funciones. Está actualización no ha permitido aumentar el grado de paralelismo, ya que cada una de las funciones tenia de esperar a su anterior para continuar ya que dependía de datos que se actualizaban con la función anterior. Se puede ver con detalle en la ilustración 9. Por todo ello, el tiempo infinito ha sido 639ms.

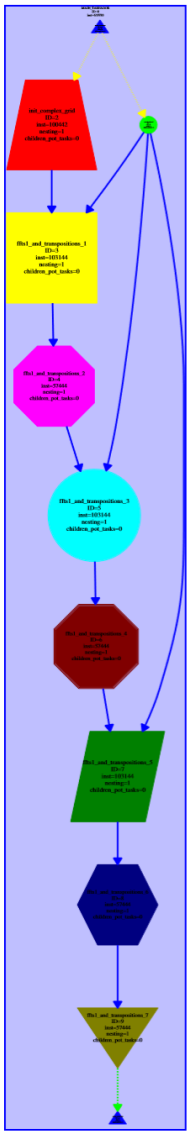

Ilustración 9 Grafo de dependencias de la versión 1

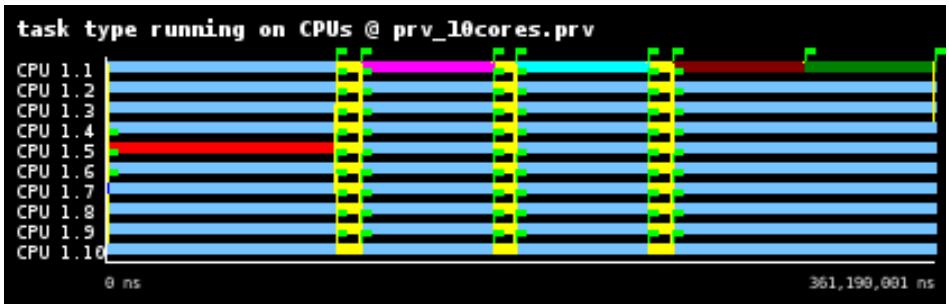

Ilustración 10 Gráfico obtenido con 10 cores ejecutando la versión 2

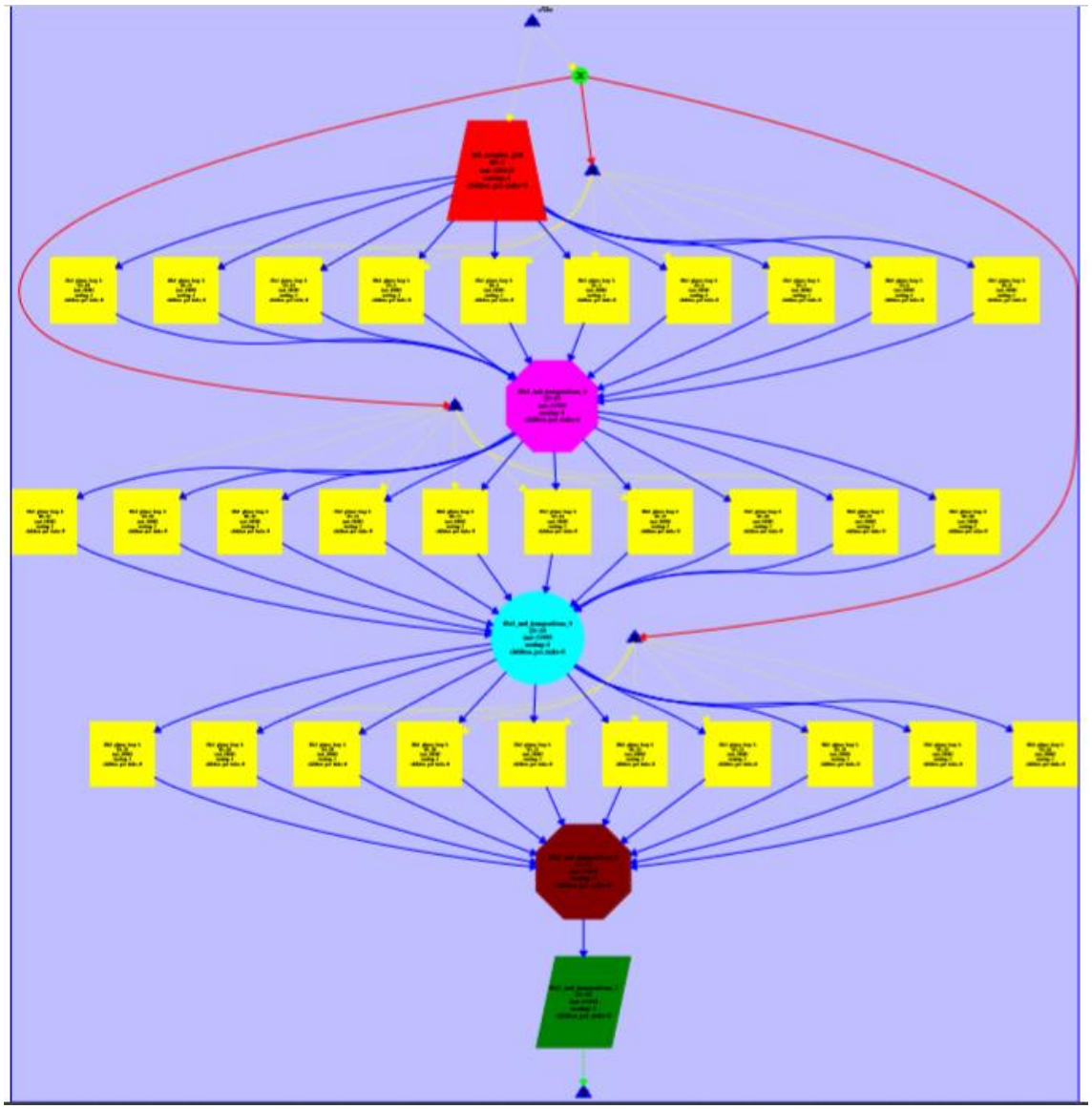

Ilustración 11 Grafo de dependencias de la versión 2

En la segunda versión hemos puesto una tarea en la función *fftsl_planes* en el bucle exterior. Destacar que hemos retirado la tarea de esta función que añadimos en la anterior versión. Con esta actualización, hemos aumentado el grado de paralización, porque la ejecución de cada una de las iteraciones es independiente a las otras, como podemos ver en la ilustración 11. En esta versión hemos obtenido un tiempo infinito de 361 y un grado de paralelismo de 1,77.

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

    for (k=0; k<N; k++) {
        tareador_start_task("fftsl_planes_loop_k");
    for (j=0; j<N; j++) {
      fftwf_execute_dft( pld, (fftwf_complex *)in_fftw[k][j][0], (fftwf_complex *)in_fftw[k][j][0]);
    }
    tareador_end_task("fftsl_planes_loop_k");
    }
}
```

Ilustración 12 Detalle de codigo de la función fftsl_planes en la versión 2

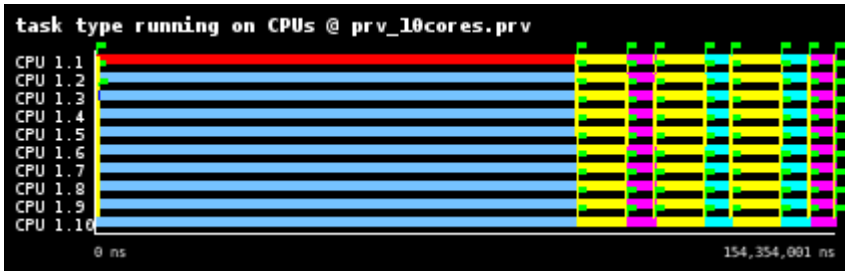

Ilustración 13 Gráfico obtenido con 10 cores ejecutando la versión 3

En la tercera versión hemos añadido una tarea en las funciones *transpose_xy_planes* y *transpose_zx_planes* en el bucle exterior. Destacar que hemos retirado la tarea de cada una de las funciones que añadimos en la versión 1. En este caso, hemos obtenido un tiempo infinito de 154ms, por la misma razón que en la anterior versión. Hemos conseguido incrementar el grado de paralelismo a 4,15.

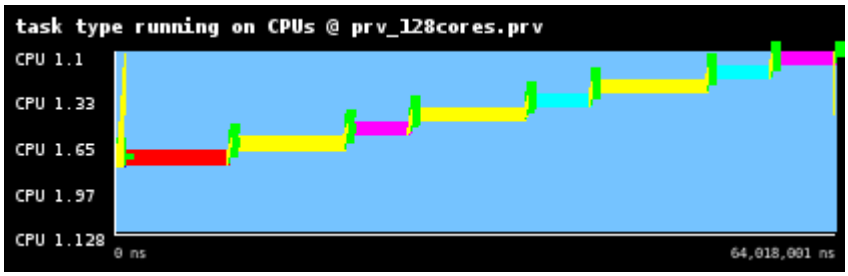

Ilustración 14 Gráfico obtenido con 128 cores ejecutando la versión 4

En la cuarta versión hemos añadido una tarea en la función *init_complex_grid* en el bucle exterior. Destacar que hemos retirado la tarea de la función que añadimos en la versión 1. En este caso, hemos obtenido un tiempo infinito de 64ms, por la misma razón que en la anterior versión. Hemos conseguido incrementar el grado de paralelismo a 9,98.

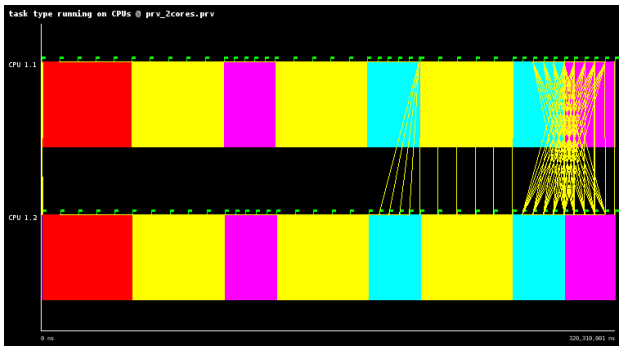

Ilustración 15 Gráfico obtenido con 2 cores ejecutando la versión 4

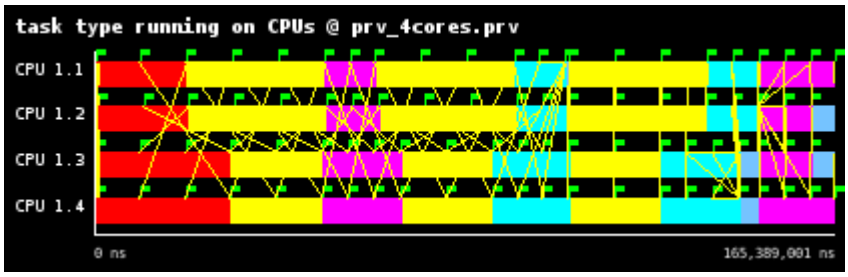

Ilustración 16 Gráfico obtenido con 4 cores ejecutando la versión 4

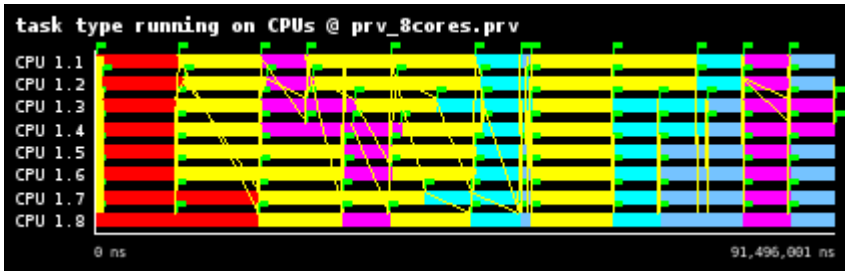

Ilustración 17 Gráfico obtenido con 8 cores ejecutando la versión 4

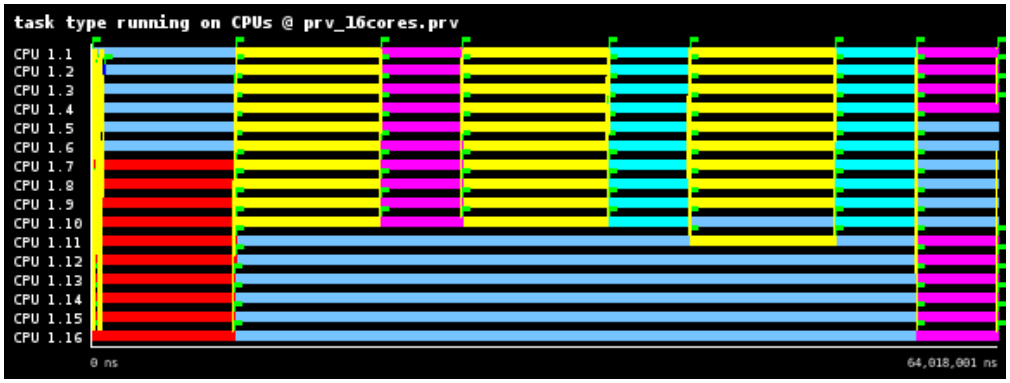

Ilustración 18 Gráfico obtenido con 16 cores ejecutando la versión 4

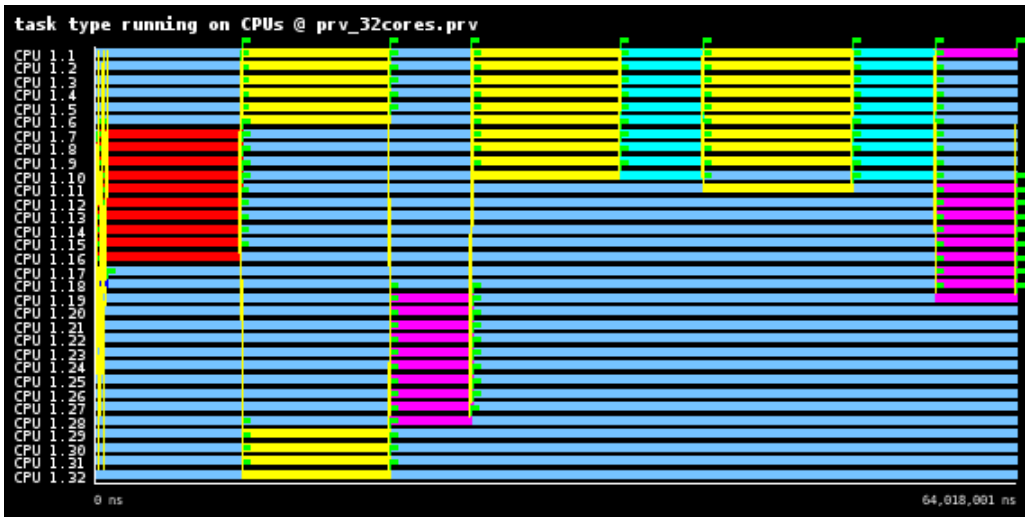

Ilustración 19 Gráfico obtenido con 32 cores ejecutando la versión 4

Ejecutando el programa con 1, 2, 4, 8, 16 y 32 cores, nos damos cuenta que lo que limita la escalabilidad del programa es el tamaño de N, que son las iteraciones de todos los bucles exteriores de cada función.

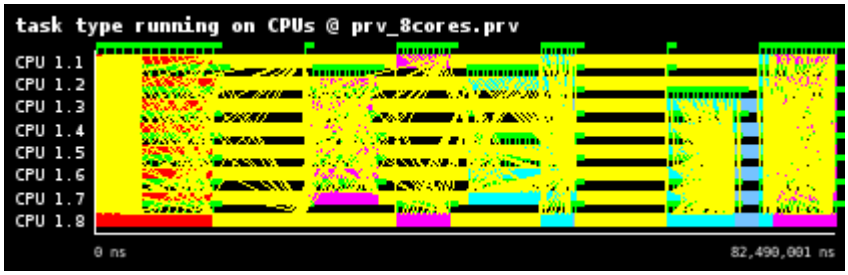

Ilustración 20 Gráfico obtenido con 8 cores ejecutando la versión 4

Por ello, en la versión 5 movemos todas las tareas a un bucle más interior, permitiendo reducir el tiempo hasta 35ms, obteniendo 18,25 de paralelismo.

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

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

```
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("complex_grid_loop_i");
      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("complex_grid_loop_i");
    }
  }
}
```

Ilustración 21 Diferencias en una de la funciones de la versión 4 (arriba) y la versión 5 (abajo)

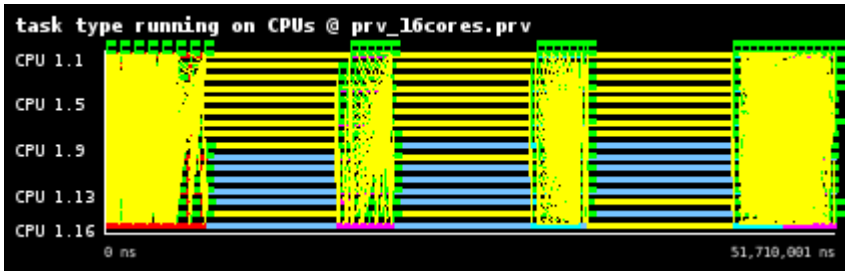

Ilustración 22 Gráfico obtenido con 16 cores ejecutando la versión 4

Podemos ver que en este caso aumentar el nivel de granulación permite reducir el tiempo casi a la mitad, por lo tanto es mejor en este caso.

Pero, si no hay muchos cores disponibles y demasiadas tareas paralelas se puede incrementar drásticamente el tiempo que emplea el sistema en el cambio de ambiente, por lo que puede ser contraproducente.

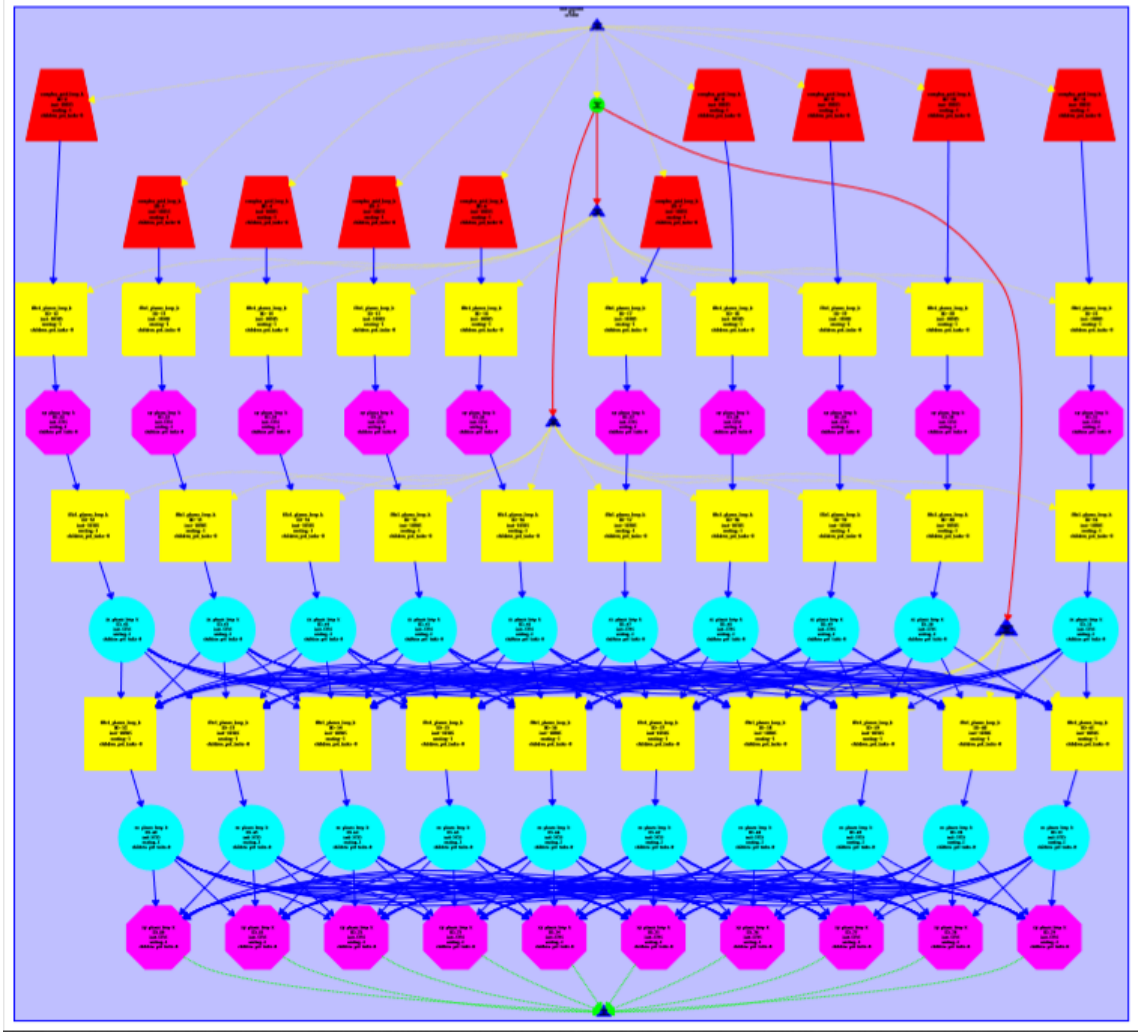

Ilustración 23 Grafo de dependencias de la versión 5

Para resumir, esta ha sido la tabla que hemos obtenido con los datos antes comentados:

Versión	*T1 (ms)*	*T_∞ (ms)*	*Grado de Paralelismo*
secuencial	639	639	1
V1	639	639	1
V2	639	361	1,77
V3	639	154	4,15
V4	639	64	9,98
V5	639	35	18,25

Sesión 3: Understanding the execution of OpenMP programs

En esta última sesión, hemos ejecutado el programa 3DFFT en un modelo real de OpenMP, obteniendo los datos reales con la arquitectura que tenemos disponible.

Para este experimento, todas las versiones las hemos ejecutado en cola en el Boada 2-4.

Primero hemos ejecutado la versión inicial, obteniendo la gráfica de Speed-Up y la de función paralela. Hemos podido comprobar que en esta versión, pasa 0,02s (17% del tiempo de ejecución) esperando a que otras funciones acaben de ejecutar código, ya que tienen dependencia.

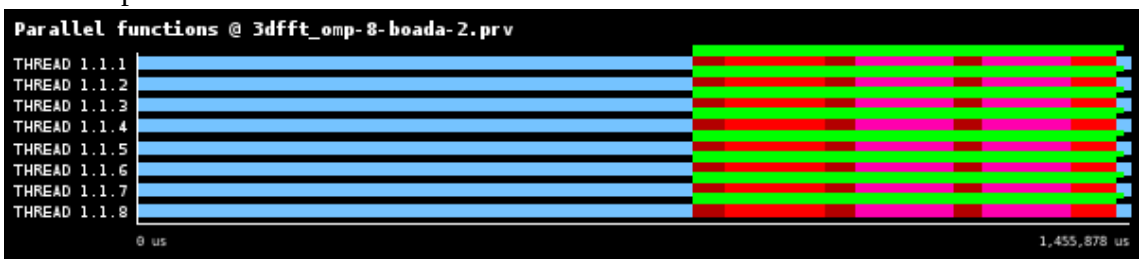

Ilustración 24 Gráfico de función en paralelo de la versión inicial

Por todo ello, hemos aplicado paralelismo al bucle más exterior de la función *init_complex_grid* y ejecutado está primera versión, obteniendo el gráfico de Speed-Up.

Como se puede ver, la mejora es clara, debido a que el cálculo interior del bucle es costoso.

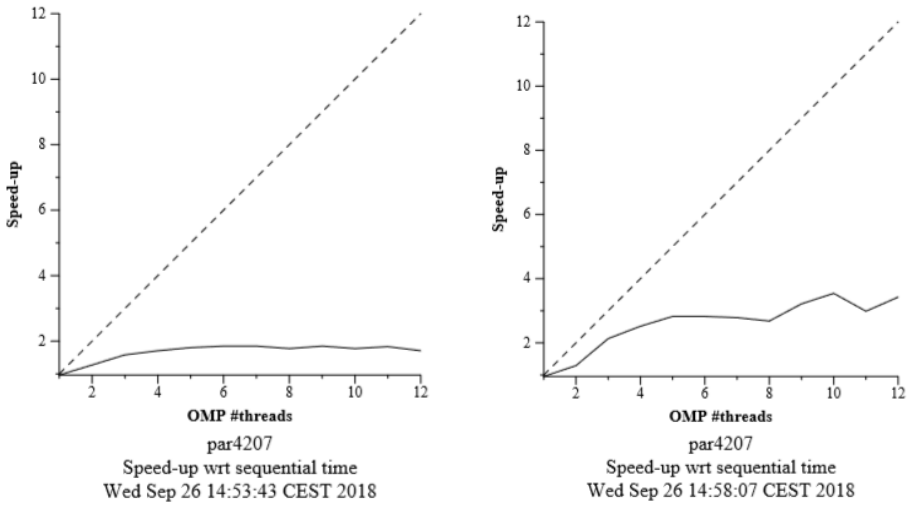

Ilustración 25 Comparación de Speed-Up entre la versión 0 (izq.) y versión 1 (der.)

En la segunda versión intentamos reducir el tiempo que se pierde al crear y sincronizar todos los hilos de ejecución. Los cambios aplicados se realizan sobre las etiquetas de paralelismo (*#pragma omp for(static, 2) private(i)*), así determinamos el número de iteraciones a ejecutar por cada hilo.

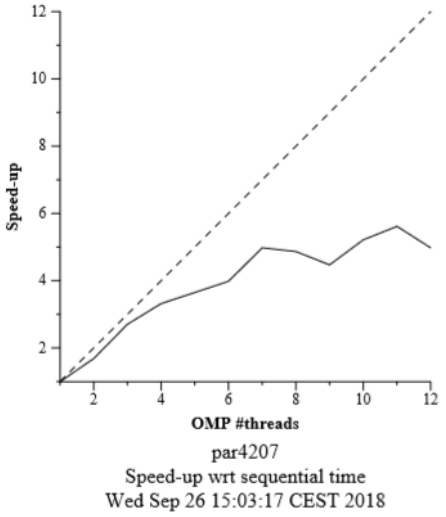

Ilustración 26 Gráfica Speed-Up de la versión 3

Por último, en la tercera versión movimos las etiquetas de paralelismo de los bucles j a los bucles k. A partir de los cambios ya hechos en la versión 2.

Hemos completado la tabla a partir de los tiempos de ejecución y de la configuración que ofrece Paraver en el fichero cfg *OMP_state_profile*:

Version	*φ*	S_∞	*T1*	*T8*	*S8*
Initial	0,12	14,29	1,715	0,66	2,62
Improved φ	0,18	8,67	1,56	0,61	2,57
Improved parallel overheads	0,19	8,1	1,54	0,61	2,52
Improved work-distribution overheads	0,21	7,95	1,62	0,36	4,39