PAR Laboratory Assignment

Lab 1: Experimental setup and tools

Sergio Araguás Villas - par4107
David Pérez Barroso - par4121
27-09-2022
22-23 Q1

Índice

Sesión 1: Experimental Setup

Esta sesión del laboratorio nos presenta las herramientas que utilizaremos a lo largo de las diferentes sesiones. La tecnología principal será el servidor boada, un servidor multiprocesador de la FIB, situado en el departamento de Arquitectura de computadores.

El servidor posee distintos nodos, los cuales utilizaremos para llevar a cabo las distintas tareas. Más concretamente utilizaremos los nodos boada-6 a boada-8 para realizar tareas interactivamente y los nodos boada-11 a boada-14 para ejecutar las tareas con un sistema de colas.

Node architecture and memory

Se nos pidió consultar los datos sobre la arquitectura de los nodos. Para saber que tipo de hardware utiliza el servidor ejecutamos los comandos lscpu y lstopo (utilizando submit-arch.sh para interactuar con el servidor) generando los archivos de texto a consultar lscpu-boada-11, lstopo-boada-11.

```
par4121@boada-6:~/lab1/arch$ more lscpu-boada-11
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Address sizes:                   46 bits physical, 48 bits virtual
Byte Order:                      Little Endian
CPU(s):                          40
On-line CPU(s) list:             0-39
Vendor ID:                       GenuineIntel
Model name:                      Intel(R) Xeon(R) Silver 4210R CPU @ 2.40GHz
CPU family:                      6
Model:                           85
Thread(s) per core:              2
Core(s) per socket:              10
Socket(s):                       2
Stepping:                        7
CPU max MHz:                     3200.0000
CPU min MHz:                     1000.0000
BogoMIPS:                        4800.00
Virtualization:                  VT-x
L1d cache:                       640 KiB (20 instances)
L1i cache:                       640 KiB (20 instances)
L2 cache:                        20 MiB (20 instances)
L3 cache:                        27.5 MiB (2 instances)
NUMA node(s):                    2
NUMA node0 CPU(s):               0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38
NUMA node1 CPU(s):               1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33,35,37,39
```

Fig. 1.1: *Datos proporcionados por el comando lscpu*

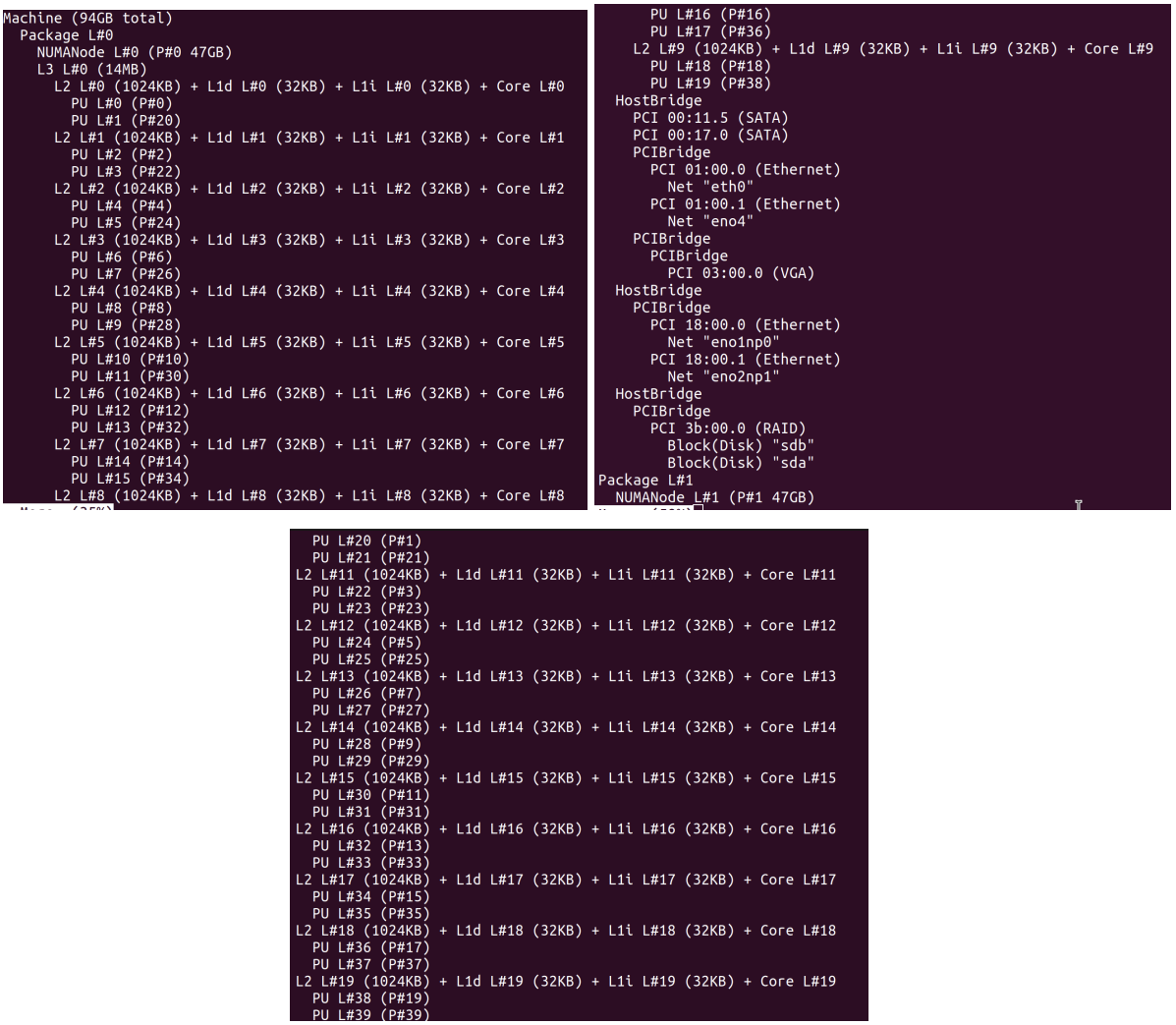


Fig. 1.12: *Datos proporcionados por el comando lstopo*

Se generó también el archivo map-boada-11.fig (*Fig.* 1.3) que nos permitió ver los datos de una manera gráfica ejecutando el comando xfig en el archivo mencionado.

Como se puede observar (*Fig.* 1.3), los cores situados en los sockets, tienen hasta tres niveles de memoria caché, siendo el último nivel L3 una memoria compartida por todos los cores del mismo socket, en cuanto a los niveles L2 y L1, estos, son privados para cada core. Además podemos observar también que el nivel L1 está dividido en una caché para instrucciones y otra para datos, L1i y L1d respectivamente.

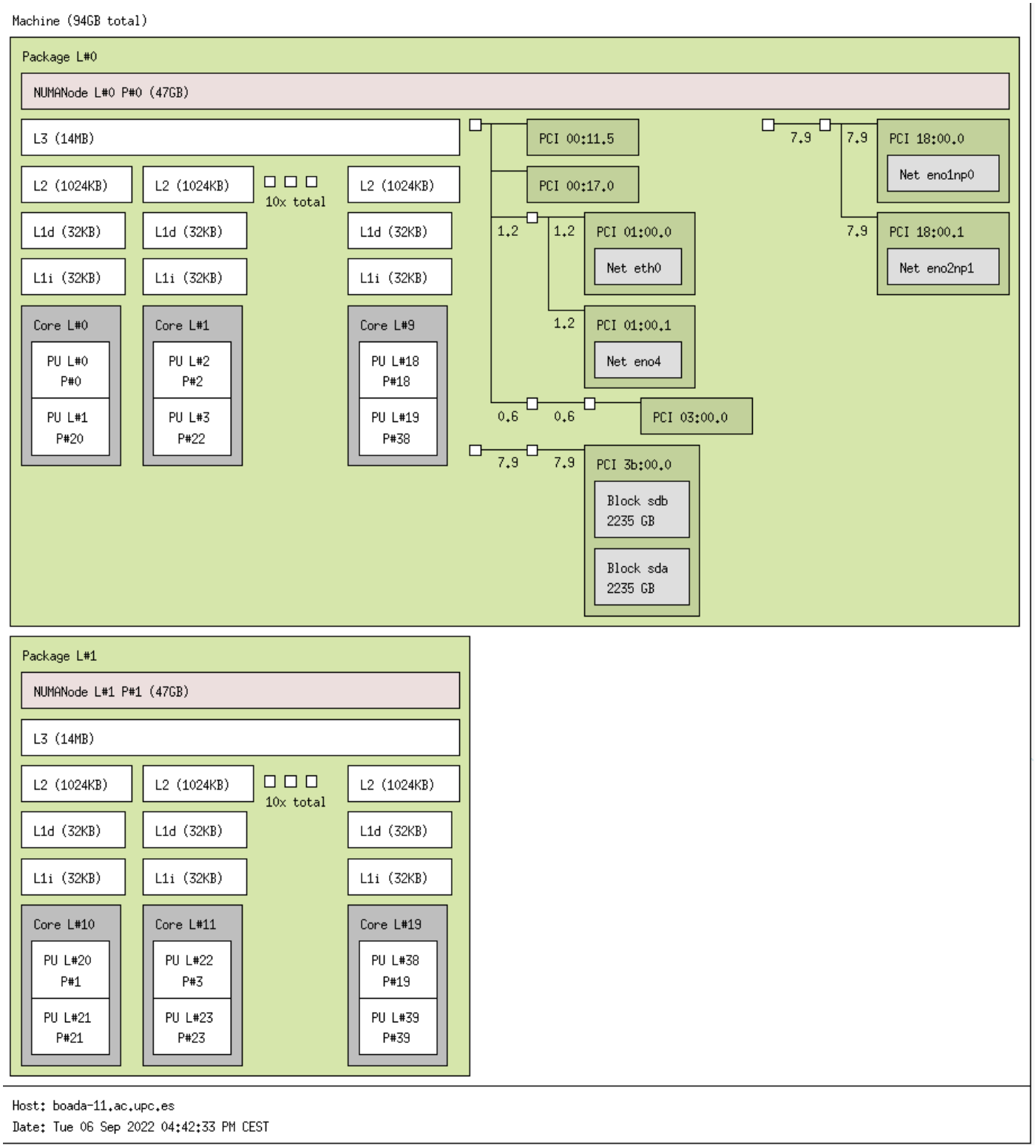


Fig. 1.3: *Estructura de un nodo*

En resumen los datos de un nodo de la máquina son los siguientes (Fig. 1.4):

	Any of the nodes among boada-11 to boada-14
Number of sockets per node	2
Number of cores per socket	10
Number of threads per core	2
Maximum core frequency	3.2 GHz
L1-I cache size (per-core)	32 KB
L1-D cache size (per-core)	32 KB
L2 cache size (per-core)	1024 KB
Last-level cache size (per socket)	14 MB
Main memory size (per socket)	94 GB
Main memory size (per node)	47 GB

Fig. 1.4: *Tabla con diferentes datos de un nodo*

Execution modes: interactive vs queued

En este apartado se nos mostraban las diferencias entre ejecutar programas mediante un sistema de colas (iniciando el programa cuando el hardware esté libre) (entre los nodos boada-11 y boada-14) o interactivamente (iniciando el programa inmediatamente pero compartiendo recursos con otros usuarios, limitando el número de núcleos usados en paralelo a 2) (entre los nodos boada-6 y boada-8). .

Durante las sesiones de laboratorio será obligatorio utilizar el sistema de colas, para obtener resultados fiables (ya que se llevará a cabo la ejecución del programa de forma aislada)
Los comandos para utilizar el sistema de colas serían :

"*sbatch [-p partition] ./submit-xxxx.sh*" - enviar programa
"*squeue*" - consultar estado
"*scancel*" + identificador - cancelar trabajo

Para hacerlo iterativamente simplemente seria ejecutar el programa con "*./run-xxxx.sh*"

Serial compilation and execution

Aquí se nos pedía comprobar las diferencias de lo expuesto en el apartado anterior, ejecutando un programa que calcula el número pi pensado para ejecutarse secuencialmente (pi_seq.sh) con *run-seq.sh* (Interactivamente) y con *submit-seq.sh* (En cola).

Siendo el tiempo de ejecución ejecutándose de manera interactiva de 2.56 segundos y de 0.69 segundos de manera encolada.

Compilation and execution of OpenMP programs

En este apartado utilizaremos el mismo programa que calcula el número pi pero escrito con alguna notación de *OpenMP*, el estándar para la programación en paralelo.

Lo ejecutaremos de dos formas distintas, interactivamente con *run-omp.sh* cambiando el número de *threads* (1, 2, 4, 8, 16 y 20 *threads*) y de manera encolada con *submit-omp.sh.*

Los resultados obtenidos ejecutándolo con 1.000.000.000 iteraciones son los siguientes (Fig. 1.5):

#threads	Interactive				Queued			
	user	system	elapsed	%CPU	user	system	elapsed	%CPU
1	2.36	0.00	0:02.37	99	0.68	0.00	0:00.70	97
2	2.36	0.00	0:01.19	199	0.69	0.00	0:00.36	189
4	2.36	0.00	0:01.19	199	0.70	0.00	0:00.19	357
8	2.38	0.05	0:01.22	199	0.75	0.00	0:00.11	662
16	2.41	0.11	0:01.27	199	0.77	0.00	0:00.06	1152
20	2.48	0.13	0:01.31	199	0.84	0.00	0:00.06	1359

Fig. 1.5: *Tabla con los valores según el número de threads*

Cuando ejecutamos de manera interactiva, los valores de tiempo aun no siendo exactamente iguales, se asemejan, por otro lado el porcentaje de CPU excepto en el uso de un *thread*, se mantiene.

Sin embargo, cuando lo ejecutamos con el sistema de colas, observamos que a medida que usamos más *threads*, el tiempo de ejecución disminuye, excepto cuando se usan 16 y 20 *threads*, que en estos casos el tiempo se mantiene.
Al contrario que el tiempo, el porcentaje de CPU va aumentando a medida que aumentamos los *threads*, creciendo más del 90% de la última ejecución, sin llegar al 100% a causa de los *overheads* que se puedan producir.

Strong vs. weak scalability

Finalmente nos explicaron el concepto de escalabilidad en paralelismo, es decir la facilidad con la que podemos conseguir un mayor poder de cómputo aumentando recursos.

Dentro de la escalabilidad en paralelismo se nos presentaron dos escenarios:

- **Strong scalability** - Capacidad del programa de reducir tiempo de ejecución al añadir potencia de procesamiento
- **Weak scalability** - Capacidad del programa de mantener el tiempo de ejecución aumentando la potencia computacional

Para entender estos escenarios se nos pidió observar estos casos:

Versión paralela de *pi_omp.c* con *strong_scalability* usando hasta 20 threads

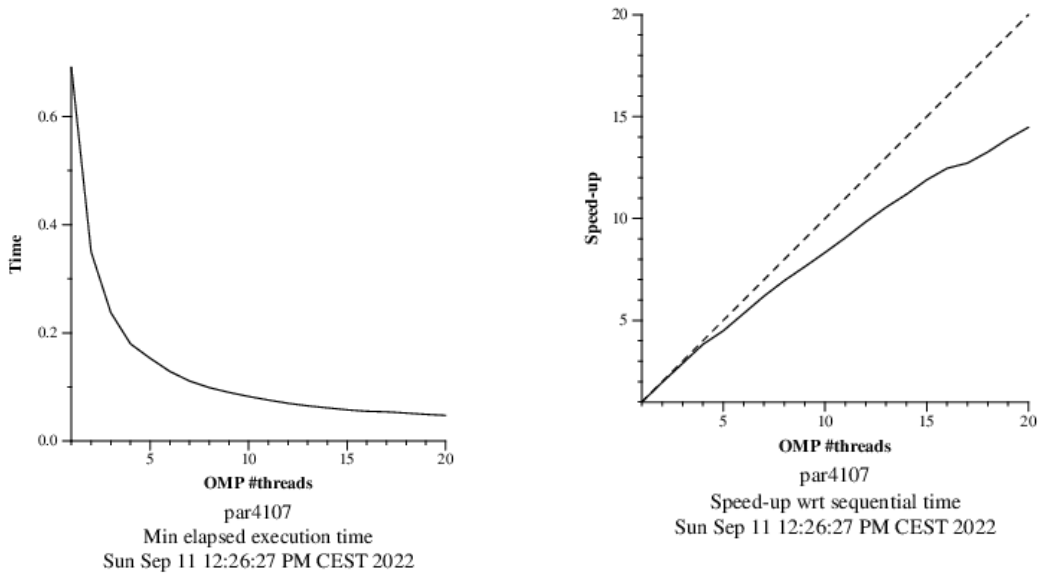


Fig. 1.6: *Gráficos strong scalability con un máximo de 20 threads*

En estos gráficos (Fig. 1.6) se puede ver como el tiempo disminuye exponencialmente y el speed-up crece linealmente hasta los 4 threads (caso ideal) y creciendo a menor ritmo hasta los 20 threads.
En este caso se ve como la strong scalability se cumple hasta los 20 threads

Versión paralela de *pi_omp.c* con *strong_scalability* usando hasta 40 threads

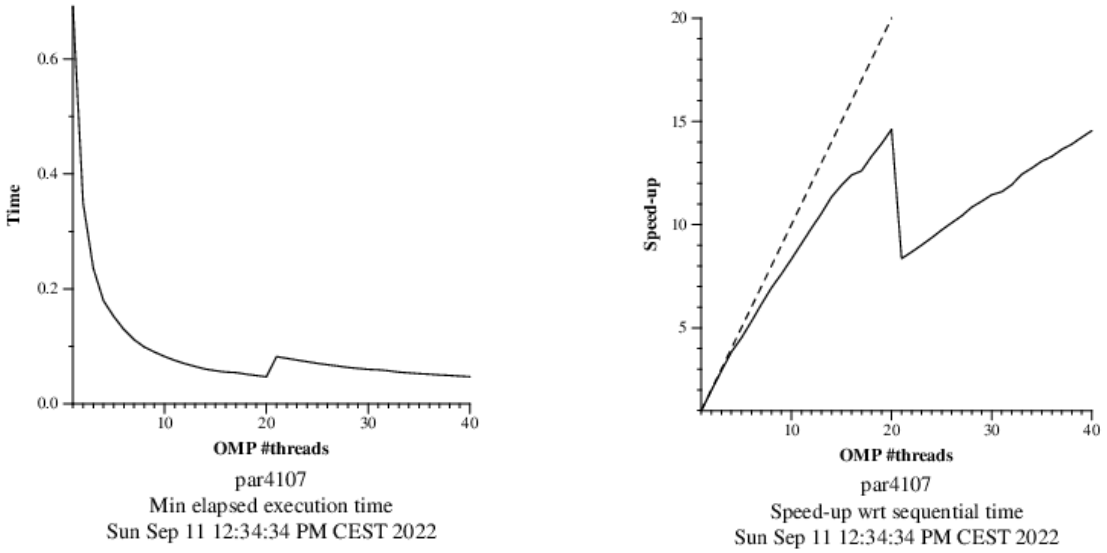


Fig. 1.7: *Gráficos strong scalability con un máximo de 20 threads*

En la Figura 1.7 se ve como el tiempo disminuye exponencialmente hasta los 20 threads (llegando a su punto más bajo en el gráfico) y luego sube ligeramente hasta recuperarse en los 40 threads. Esto se debe a que hasta los 20 threads hay un incremento del speed up (más o menos constante, bajando el crecimiento llegando a los 20 threads) y a partir de los 20 threads el speed-up disminuye drásticamente hasta los 22 threads, donde vuelve a recuperar un crecimiento no tan fuerte como el primero.

En este caso la strong scalability no está tan presente como en el primer caso, por el aumento de tiempo a partir de los 20 threads, aunque se puede ver una tendencia de disminución del tiempo de ejecución a partir de los 22 threads.

Versión paralela de pi_omp.c con weak scalability usando hasta 20 threads

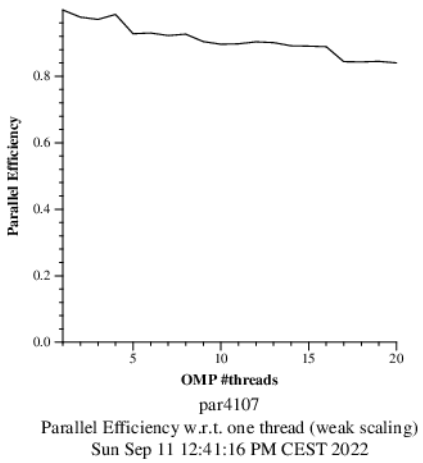


Fig. 1.8: *Gráficos weak scalability con un máximo de 20 threads*

En este caso el programa, cuantos más threads se le proporcionan, más iteraciones añade al tamaño del problema.

Podemos ver que la eficiencia paralela baja hasta los 20 threads, pero que se mantiene por encima del 0.8 haciendo presente su weak scalability.

(Lo ideal sería 1 de eficiencia paralela, ya que significaría una relación sostenible entre nueva potencia ofrecida y trabajo extra)

Sesión 2: Systematically analyzing task decompositions with Tareador

Esta sesión tiene como objetivo familiarizarnos con el entorno de *Tareador*, una herramienta utilizada para analizar el posible paralelismo que obtendríamos a la hora de aplicar diferentes estrategias de descomposición de tareas en el código secuencial que deseemos paralelizar.

Brief Tareador hands-on

Para empezar a familiarizarnos con el entorno, compilamos y ejecutamos con *Tareador* el código que encontramos en *3dfft_tar.c*, obteniendo un gráfico de dependencias entre las tareas del codigo (Fig 2.1),.

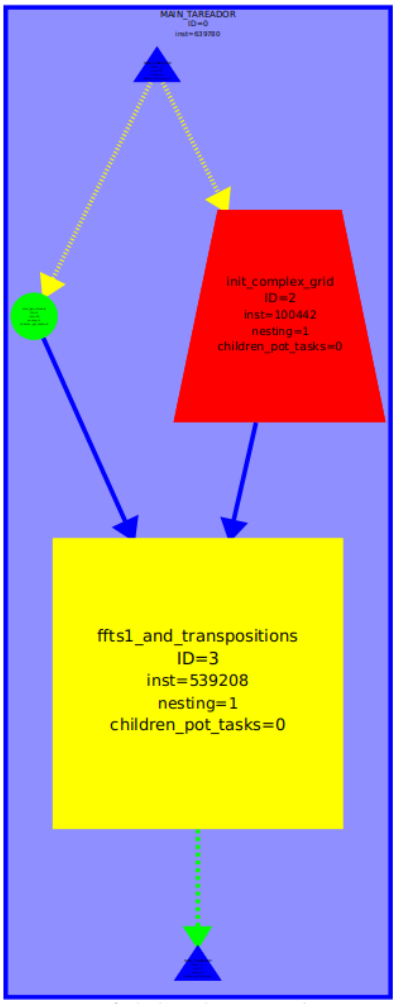


Fig. 2.1: *Grafo de dependencias versión secuencial*

Para poder ejecutarlo, es necesario hacerlo con el script *run-tareador.sh* con el comando *./run-tarreador.sh 3dfft_tar* .

Seguidamente, para poder analizar el paralelismo y la ejecución con diferente número de *cores*, utilizamos la herramienta que contiene *Tareador*, llamada *Paraver*. Ejecutándolo solo con 1 procesador, observamos que el tiempo de ejecución es de 639780001 ns (T1). Éste se ha ejecutado secuencialmente debido a que solo dispone de un procesador.

Aumentando el número de procesadores observamos que el tiempo de ejecución no varía, por lo que T∞ es igual a T1. Lo que significa que no aprovecha nada el paralelismo de la arquitectura.

Exploring new task decompositions for 3DFFT

A continuación se nos pide ejecutar el mismo programa con diferentes versiones, modificando la granularidad de las tareas, y midiendo sus tiempos para poder aprovechar al máximo el paralelismo de la arquitectura.
Los datos y las diferentes versiones se muestran a continuación:

Version	T1	T∞	Parallelism
Seq	639,780,001 ns	639,780,001 ns	1
v1	639,780,001 ns	639,707,001 ns	≈1
v2	639,780,001 ns	392,105,001 ns	1.63
v3	639,780,001 ns	185,269,001 ns	3.45
v4	639,780,001 ns	64,018,001 ns	9.99
v5	639,780,001 ns	35,151,001 ns	18.20

Tabla T1,T∞ y paralelismo por versiones

Versión 1

Para esta versión, modificaremos el código de tal manera que crearemos una tarea por cada llamada a una función de *ffts1* o de *transposición*.
Como se observa por la T∞ de esta versión y por el gráfico de dependencias, el paralelismo es mínimo, por no decir nulo, puesto que realiza las tareas de una manera secuencial, ya que cada función dependerá de la modificación de datos de la anterior función.

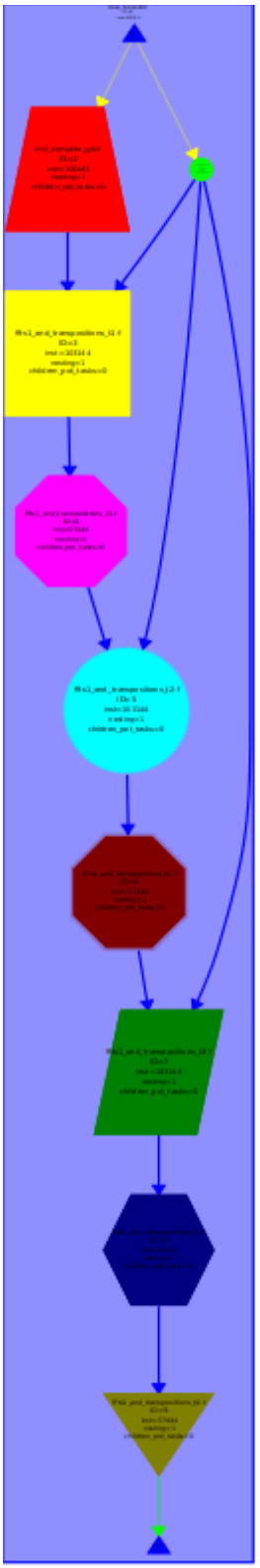


Fig. 2.2: *Grafo de dependencia de V1*

Versión 2

En esta versión, aparte de lo añadido en la anterior, modificaremos el código de la función *ffts1_planes*, añadiendo una tarea en cada iteración del bucle k.
Debido a esta descomposición de tareas más granularmente, conseguiremos un mayor paralelismo.

Y efectivamente esto se observa tanto en el grafo de dependencias de esta versión (Fig. 2.3), como en T∞ obtenida, donde esta ya es menor que en sus versiones anteriores, obteniendo un nivel de paralelismo de 1,63.

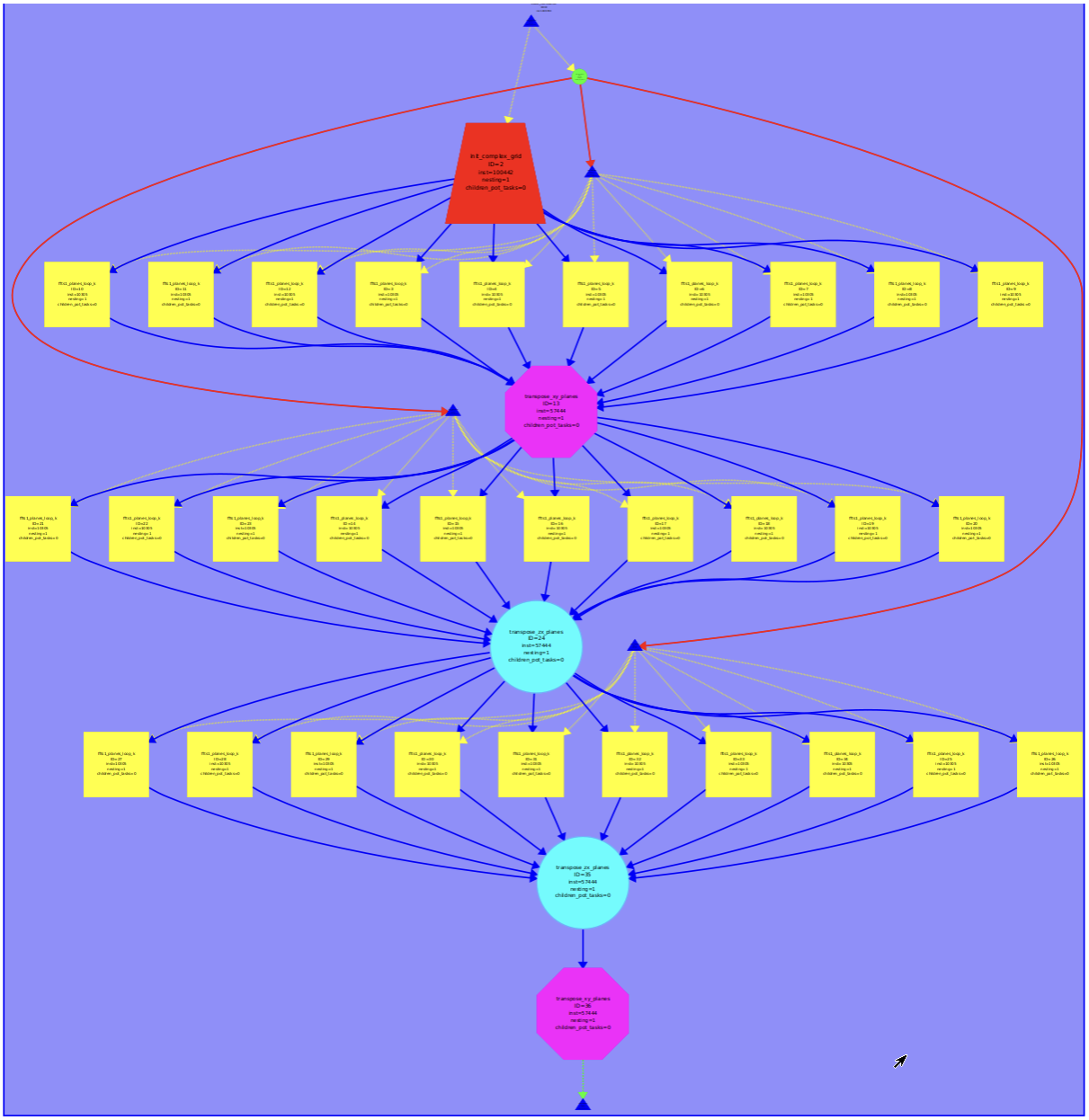


Fig. 2.3: *Gráfico de dependencia de V2*

Versión 3

Partiendo de la versión 2 y modificaremos las funciones *transpose_xy_planes* y *transpose_zx_planes*. Añadiendo tareadores por cada iteración de los bucles k de cada una de las funciones, aumentando su granularidad. Gracias a ello, como se observa en su grafo de dependencias (Fig 2.4) y en el tiempo de T∞ obtenido,

se logra un mayor paralelismo que en cualquiera de sus versiones anteriores. Ahora tenemos un nivel de paralelismo de 3,45.

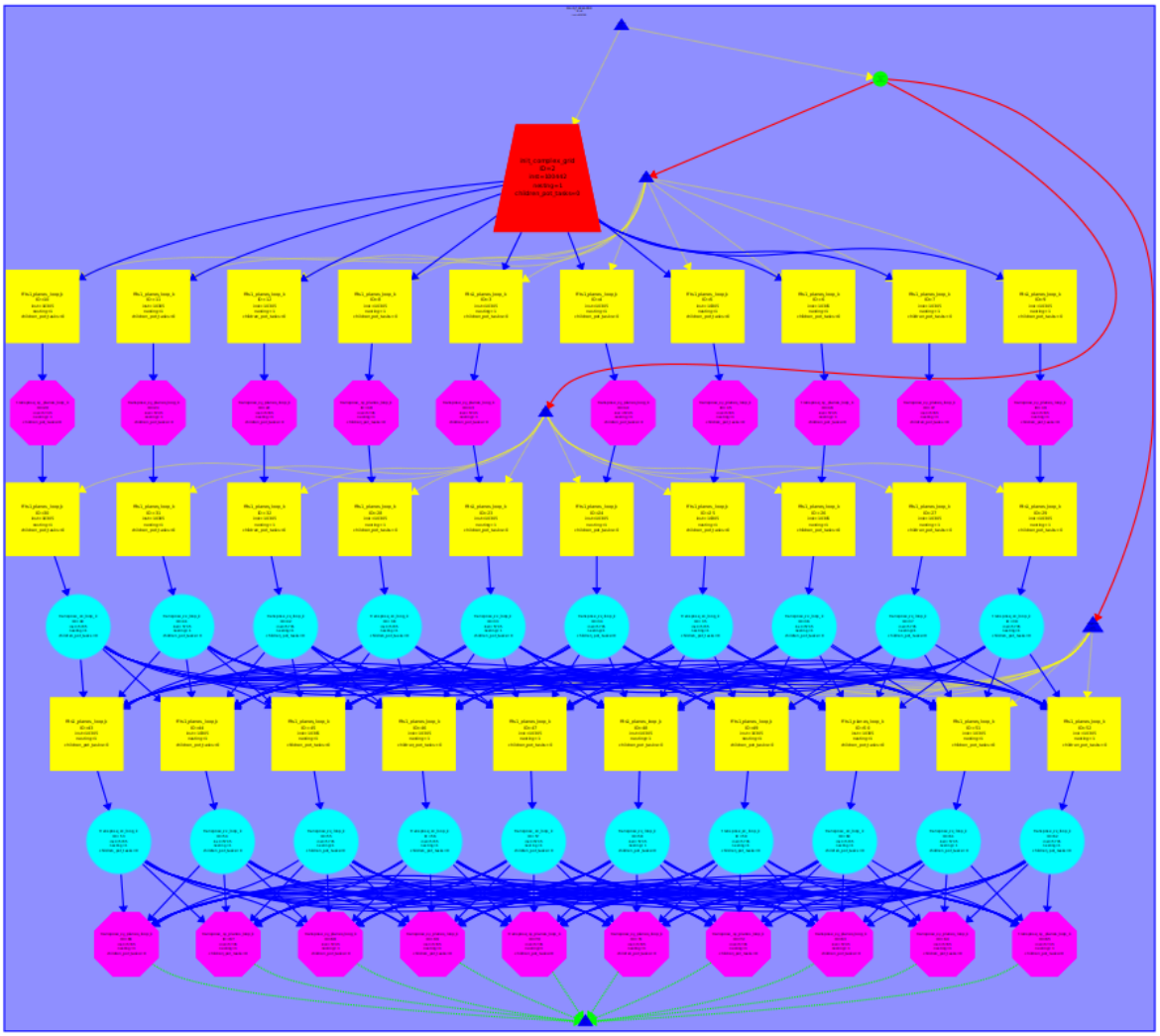


Fig. 2.4: *Gráfico de dependencia de* V3

Versión 4

Partiendo de la versión 3 aumentaremos la granularidad de las tareas de *init_complex_grid*, añadiendo tareadores dentro del bucle exterior (bucle k). Una vez hecho esto simularemos su ejecución paralela con diferentes procesadores.
Como se puede apreciar en el gráfico (Fig. 2.6), el programa obtiene su mejor tiempo de ejecución con 16 núcleos ya que no existe una mejora si se utilizan más.
Vista la diferencia entre v3 y v4, creemos que la escalabilidad de este programa está limitada por la granularidad de las operaciones, ya que si se dividieran todavía más las tareas no dependientes habría más margen para el paralelismo.

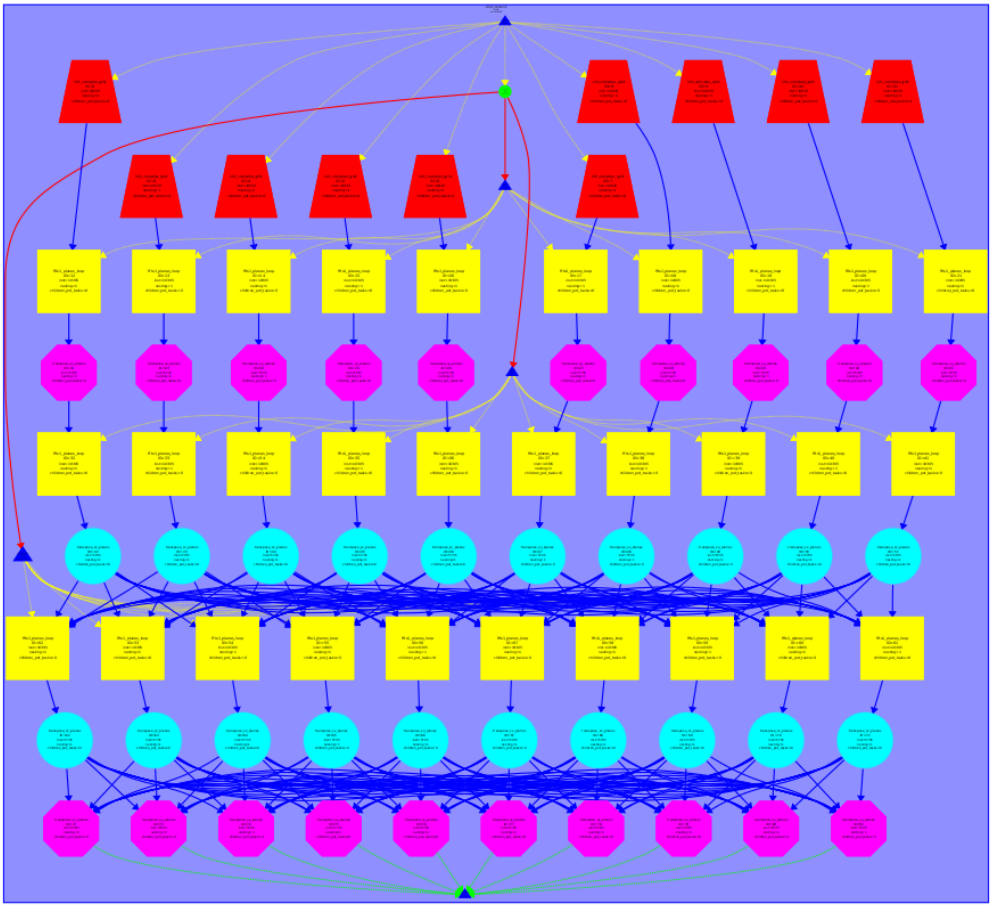


Fig. 2.5: *Grafo de dependencia de V4*

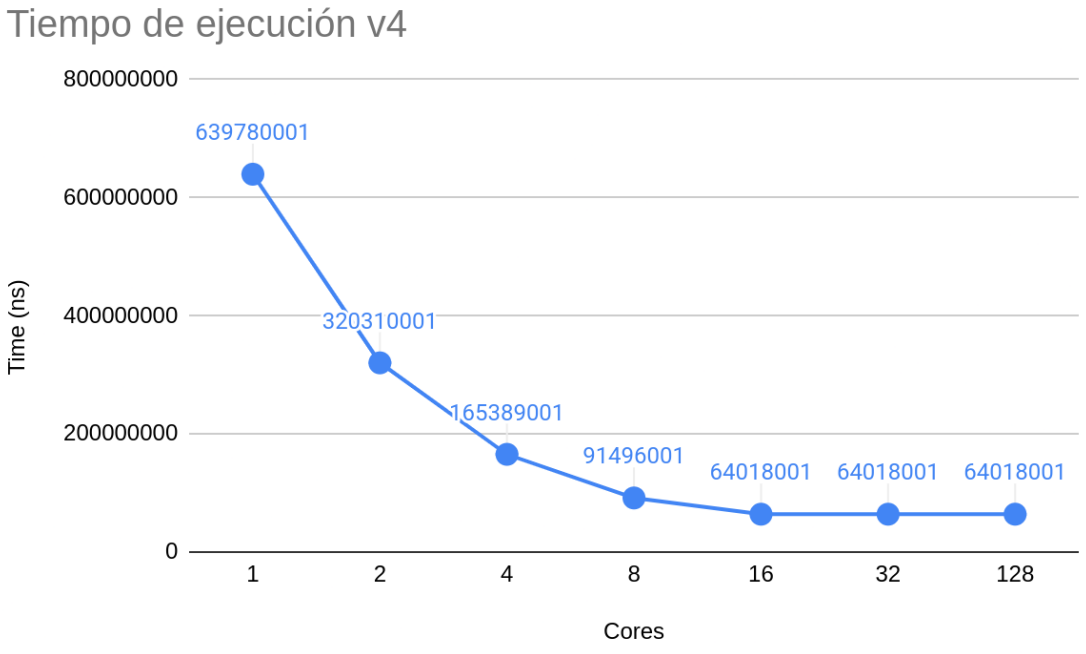


Fig. 2.6: *Gráfico Strong Scalability V4*

Versión 5

En esta última versión, se nos pide modificar la versión v4 para conseguir tareas con todavía más granularidad. Para ello lo que haremos será mover los tareadores del bucle k (Fig. 2.9) al bucle j (Fig. 2.10) de todos los bucles con tareadores.

Lo que obtendremos será una mejora notable en su escalabilidad obteniendo un T∞ de 35151001 con 128 núcleos.

Por lo tanto, podemos decir que en este caso merece la pena aumentar la granularidad ya que pasaremos de un paralelismo de 9,99 en v4 a 18,20 en v5.

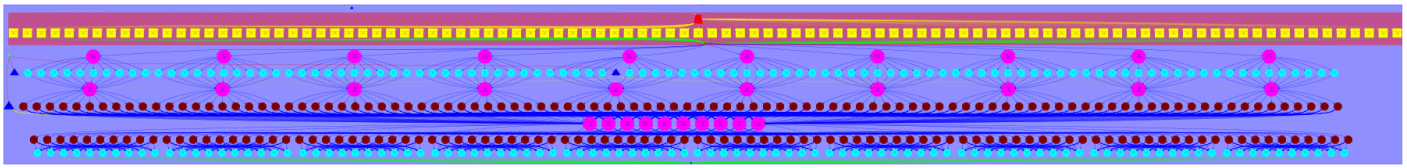


Fig. 2.7: *Grafo de dependencias V5*

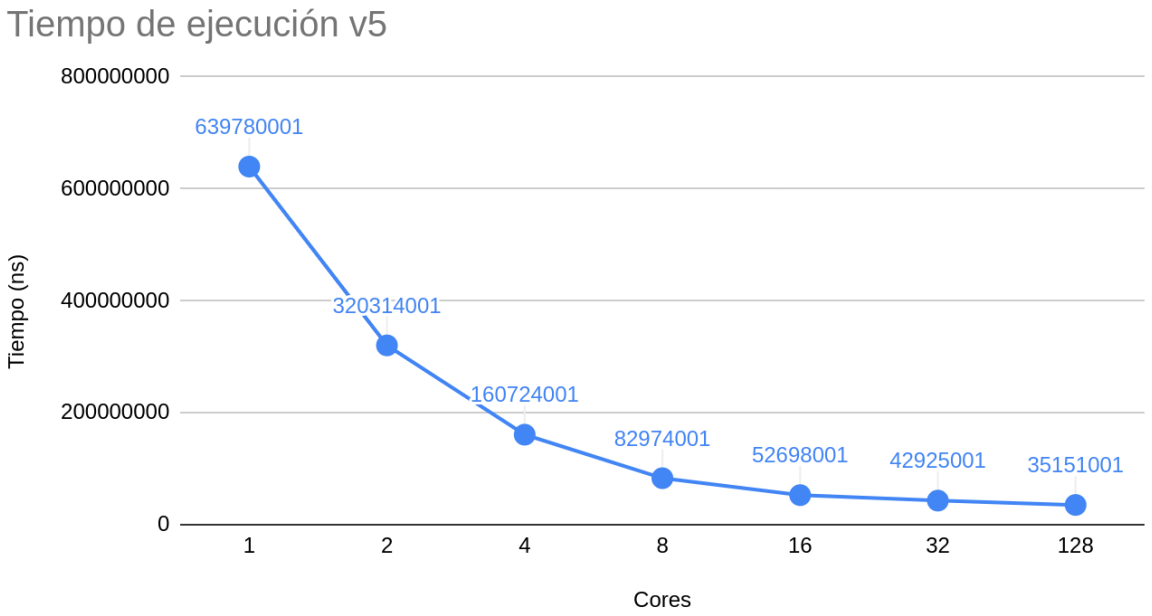


Fig. 2.8: *Gráfico strong scalability V5*

Se puede observar un gráfico similar al de v4, pero si nos fijamos en los valores, podremos ver que gracias al nuevo nivel de granulación de v5, se aprecia un mayor escalado en threads que la versión anterior, por lo que ahora al llegar a 16 threads, el tiempo de ejecución no se mantiene constante.

```
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");
    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");
  }
}

void transpose_xy_planes(fftwf_complex  tmp_fftw[][N][N], fftwf_complex in_fftw[][N][N]) {
    int k,j,i;

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

Fig. 2.9: *Código de v4 con tareadores dentro del bucle 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("init_complex_grid_loop_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_j");
    }
  }
}

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

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

Fig. 2.10: *Código de v5 con tareadores dentro del bucle j*

Sesión 3: Understanding the execution of OpenMP programs

Esta última sesión del Lab1 tiene como objetivo familiarizarnos con *Paraver*, un entorno que nos proporciona información y nos permite ver la ejecución de los programas paralelizados con OpenMP.

Trabajaremos con el fichero *3dfft_omp.c*, el cual iremos modificando su estrategia de paralelización e iremos estudiando la eficiencia, el tiempo, los posibles overheads de entre otros parámetros, de cada versión.

Example 3DFFT: Obtaining parallelisation metrics using modelfactors

Para esta primera versión la cual aún no se ha modificado, se nos pide que ejecutemos el análisis con *modelfactors* (*Fig. 3.1*), obteniendo las tablas de datos y contestar a las preguntas siguientes:

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	1.31	0.78	0.79	1.28	1.46
Speedup	1.00	1.69	1.66	1.02	0.90
Efficiency	1.00	0.42	0.21	0.09	0.06

Table 1: Analysis done on Tue Sep 20 04:41:23 PM CEST 2022, par4121

Overview of the Efficiency metrics in parallel fraction, ϕ=83.97%					
Number of processors	1	4	8	12	16
Global efficiency	98.80%	49.34%	24.01%	8.69%	5.49%
Parallelization strategy efficiency	98.80%	89.07%	86.93%	71.60%	56.58%
Load balancing	100.00%	98.27%	97.76%	97.95%	97.60%
In execution efficiency	98.80%	90.63%	88.92%	73.10%	57.97%
Scalability for computation tasks	100.00%	55.40%	27.62%	12.14%	9.70%
IPC scalability	100.00%	68.51%	51.34%	38.43%	39.68%
Instruction scalability	100.00%	98.33%	96.18%	94.12%	92.18%
Frequency scalability	100.00%	82.24%	55.94%	33.57%	26.52%

Table 2: Analysis done on Tue Sep 20 04:41:23 PM CEST 2022, par4121

Statistics about explicit tasks in parallel fraction					
Number of processors	1	4	8	12	16
Number of explicit tasks executed (total)	17920.0	71680.0	143360.0	215040.0	286720.0
LB (number of explicit tasks executed)	1.0	0.94	0.88	0.92	0.92
LB (time executing explicit tasks)	1.0	0.98	0.98	0.98	0.98
Time per explicit task (average us)	60.6	27.36	27.44	41.62	39.08
Overhead per explicit task (synch %)	0.16	10.75	12.99	35.67	71.15
Overhead per explicit task (sched %)	1.04	1.5	2.02	3.96	5.56
Number of taskwait/taskgroup (total)	1792.0	1792.0	1792.0	1792.0	1792.0

Table 3: Analysis done on Tue Sep 20 04:41:23 PM CEST 2022, par4121

Fig. 3.1: Resultados de modelfactors v0

¿Es la escalabilidad apropiada?
Según los datos mostrados por la primera tabla, la escalabilidad parece ser insatisfactoria, puesto que a partir de la ejecución con 4 procesadores el tiempo de ejecución empieza a aumentar, parejamente con la disminución del *Speedup* y de la eficiencia.

¿Es el overhead debido a la sincronización despreciable?
Según los datos obtenidos no es despreciable, ya que a partir de los 8 núcleos la eficiencia de paralelización empieza a bajar sensiblemente, desde el 88,92% con 8 procesadores a 73,10% con 12 y baja drásticamente con 16 procesadores hasta el 57,97%.

¿Está afectando el overhead a la ejecución por tarea explícita?
Como se puede observar en la tercera tabla (Fig. 3.1 - Tabla 3), *el overhead por tarea explícita por sincronización* afecta mínimamente a la ejecución con un procesador, pero según aumentamos el número de procesadores con los que ejecutamos el código, este *overhead* aumenta en un porcentaje considerable.
A partir de los 4 núcleos empieza a crecer drásticamente, llegando a un porcentaje inaceptable de 71,15% con 16 núcleos.

¿Cual es la fracción paralelizable para esta versión del programa?
83,7%.

¿Es la eficiencia para las regiones paralelas apropiada?
La eficiencia para las regiones paralelas es poco apropiada ya que esta baja rápidamente cuantos más procesadores utiliza, comenzado con una eficiencia del 98.8% con 1 núcleo llegando a una del 5,49% con 16 núcleos.

¿Cuál es el factor que más negativamente está influyendo?
Una estrategia de paralelización poco eficiente puede ser la razón de que existan grandes overheads por sincronización que causen su baja eficiencia con múltiples núcleos.

Example 3DFFT: Obtaining parallelisation details using Paraver

Hay dos factores clave que influyen en la escalabilidad general y el rendimiento final. Mirando las dos ventanas de líneas de tiempo podemos ver estos dos factores: Primeramente hay una función que no está paralelizada, además hay un estado de subproceso que predomina a lo largo de la ventana de la línea de tiempo.

El problema más importante parece ser el segundo, porque hay un fuerte problema de escalabilidad debido a la cantidad de tareas y sincronizaciones en el programa.

¿Piensas que este problema de overhead es constante o es en función del número de threads? ¿Por qué?

El problema de *overhead*, como podemos ver en los datos obtenidos, no es constante, crece en función del número de procesadores.
Esto se debe a que al aumentar el número de procesadores, se está incrementando el tiempo de sincronización a causa de los posibles mapeados del *data sharing*, así como la sincronización de tareas entre todos ellos.

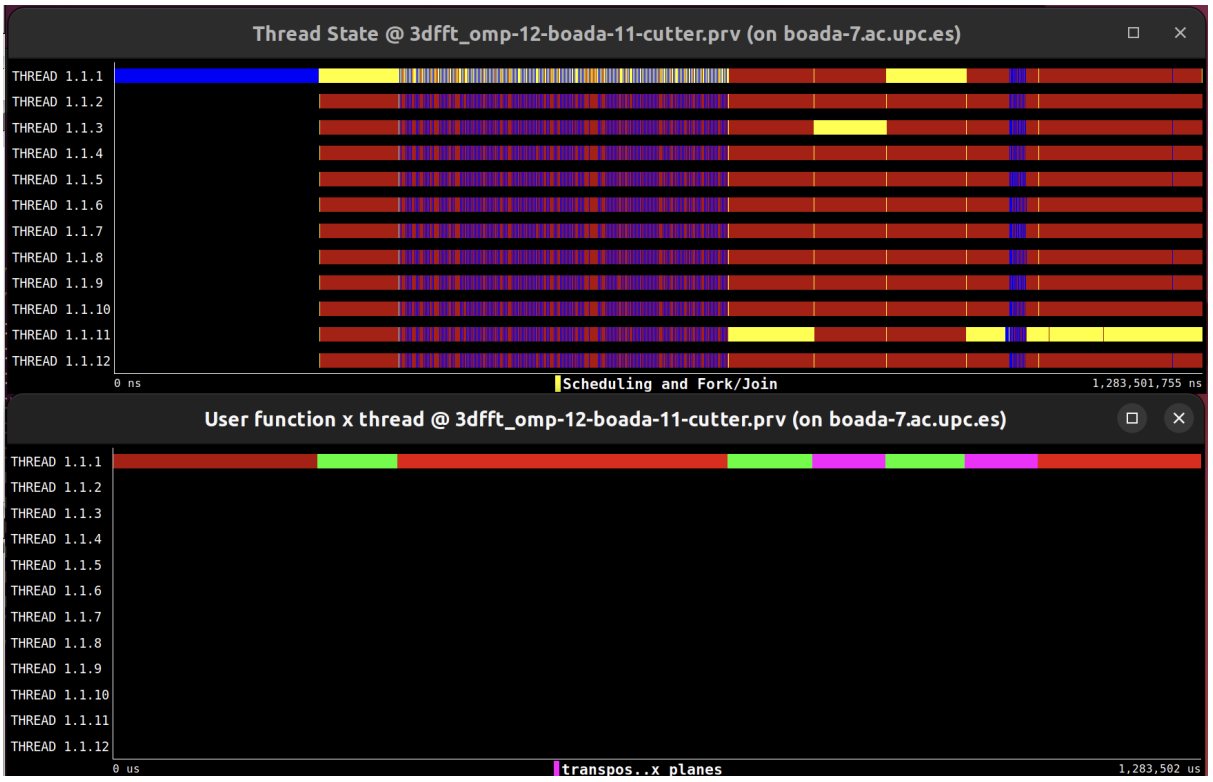


Fig. 3.2: Líneas temporales de la ejecución del programa

Implicit and Explicit tasks

¿Qué tipo de granularidad de tarea explícita crees que tienes: grano fino o grueso?

Como se puede observar en la línea temporal de las tareas explícitas ejecutadas, las tareas tienen una granularidad fina.

Fig. 3.3: Línea temporal de la distribución de las tareas implícitas y explícitas durante la ejecución

Example 3DFFT: Reducing Parallelisation Overheads and Analysis

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	1.25	0.57	0.45	0.40	0.38
Speedup	1.00	2.21	2.77	3.17	3.25
Efficiency	1.00	0.55	0.35	0.26	0.20

Table 1: Analysis done on Mon Sep 26 06:36:12 PM CEST 2022, par4121

Overview of the Efficiency metrics in parallel fraction, ϕ=83.27%					
Number of processors	1	4	8	12	16
Global efficiency	99.96%	77.67%	60.48%	50.85%	44.85%
Parallelization strategy efficiency	99.96%	96.65%	96.64%	96.97%	97.14%
Load balancing	100.00%	97.71%	98.12%	97.77%	98.50%
In execution efficiency	99.96%	98.91%	98.49%	99.19%	98.63%
Scalability for computation tasks	100.00%	80.36%	62.59%	52.44%	46.17%
IPC scalability	100.00%	80.68%	66.87%	57.68%	50.99%
Instruction scalability	100.00%	99.99%	99.98%	99.97%	99.96%
Frequency scalability	100.00%	99.61%	93.62%	90.95%	90.59%

Table 2: Analysis done on Mon Sep 26 06:36:12 PM CEST 2022, par4121

Statistics about explicit tasks in parallel fraction					
Number of processors	1	4	8	12	16
Number of explicit tasks executed (total)	70.0	280.0	560.0	840.0	1120.0
LB (number of explicit tasks executed)	1.0	0.97	0.91	0.92	0.8
LB (time executing explicit tasks)	1.0	0.99	0.99	0.98	0.99
Time per explicit task (average us)	14880.2	4628.96	2971.6	2364.22	2014.0
Overhead per explicit task (synch %)	0.0	3.39	3.35	2.97	2.71
Overhead per explicit task (sched %)	0.03	0.03	0.03	0.04	0.05
Number of taskwait/taskgroup (total)	7.0	7.0	7.0	7.0	7.0

Table 3: Analysis done on Mon Sep 26 06:36:12 PM CEST 2022, par4121

Fig. 3.4: Resultados modelfactors v1

¿Hemos mejorado la ejecución del programa? Ha aumentado el tiempo de ejecución?

Se ha reducido notablemente el tiempo de ejecución del programa, y la eficiencia de paralelización ha pasado de ser contraproducente a reducir el tiempo de ejecución a medida que se añaden más núcleos.

¿Observas alguna diferencia en los overheads de las tareas explícitas entre la versión anterior y la optimizada?
Observamos una mejora significativa en la cantidad de overheads respecto a la versión anterior, obteniendo en la segunda unos overheads mucho menores.

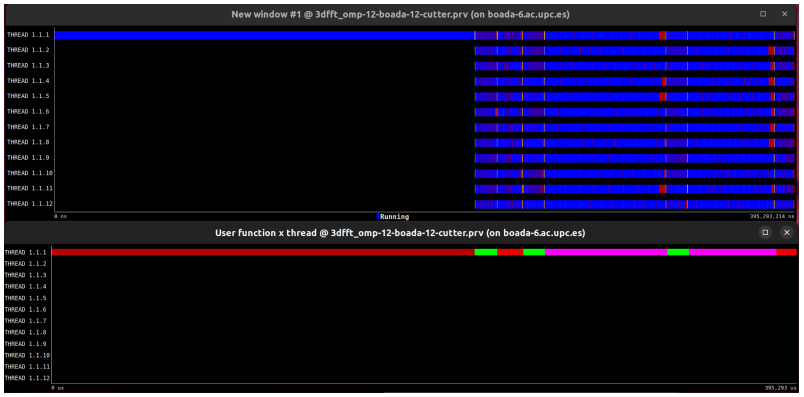


Fig. 3.5: Líneas temporales de la ejecución del programa optimizado

Podemos observar que el tiempo de sincronización entre las dos versiones (Fig. 3.2 y Fig 3.5) es muy diferente ya que en la primera ocupa un gran porcentaje del tiempo total mientras que en la segunda se ha reducido drásticamente debido a la nueva granularidad, lo que en consecuencia reduce el tiempo de ejecución.

Como comentario podemos apreciar que en la nueva versión es la función no paralelizada *init_complex_grid* la que ocupa más tiempo de ejecución dentro del tiempo total y limita el máximo speedup posible ya que no puede mejorar su tiempo.

- init_complex_grid Duration: 224,103.48 us

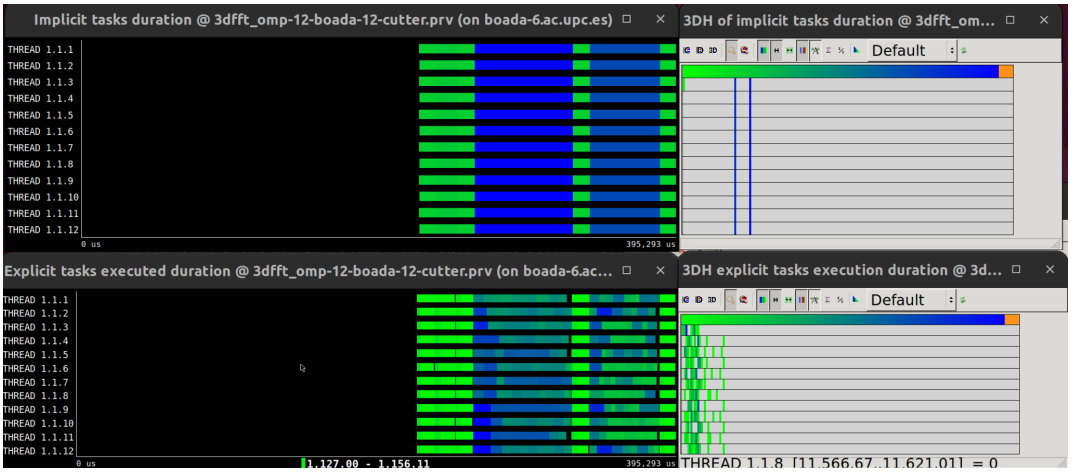


Fig. 3.6: Línea temporal de la distribución de las tareas implícitas y explícitas durante la ejecución optimizada

Como podemos ver en las respectivas líneas temporales (Fig. 3.6), existe una reducción de duración de las tareas implícitas y explícitas entre las dos versiones.

Example 3DFFT: Improving φ and Analysis

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	1.29	0.41	0.25	0.21	0.26
Speedup	1.00	3.14	5.07	6.01	4.95
Efficiency	1.00	0.79	0.63	0.50	0.31

Table 1: Analysis done on Mon Sep 26 08:08:05 PM CEST 2022, par4121

Overview of the Efficiency metrics in parallel fraction, ϕ=99.97%					
Number of processors	1	4	8	12	16
Global efficiency	99.83%	78.52%	63.32%	50.10%	30.90%
Parallelization strategy efficiency	99.83%	95.16%	92.03%	81.36%	55.73%
Load balancing	100.00%	97.20%	96.19%	94.91%	95.78%
In execution efficiency	99.83%	97.90%	95.68%	85.73%	58.18%
Scalability for computation tasks	100.00%	82.52%	68.80%	61.58%	55.45%
IPC scalability	100.00%	84.86%	72.86%	67.35%	60.84%
Instruction scalability	100.00%	99.80%	99.54%	99.28%	99.03%
Frequency scalability	100.00%	97.43%	94.86%	92.09%	92.02%

Table 2: Analysis done on Mon Sep 26 08:08:05 PM CEST 2022, par4121

Statistics about explicit tasks in parallel fraction					
Number of processors	1	4	8	12	16
Number of explicit tasks executed (total)	2630.0	10520.0	21040.0	31560.0	42080.0
LB (number of explicit tasks executed)	1.0	0.99	0.91	0.86	0.85
LB (time executing explicit tasks)	1.0	0.98	0.99	0.97	0.97
Time per explicit task (average us)	489.78	148.38	88.98	66.27	55.2
Overhead per explicit task (synch %)	0.02	4.77	8.02	19.44	73.78
Overhead per explicit task (sched %)	0.15	0.3	0.6	2.49	5.64
Number of taskwait/taskgroup (total)	263.0	263.0	263.0	263.0	263.0

Table 3: Analysis done on Mon Sep 26 08:08:05 PM CEST 2022, par4121

Fig. 3.7: Resultados modelfactors v2

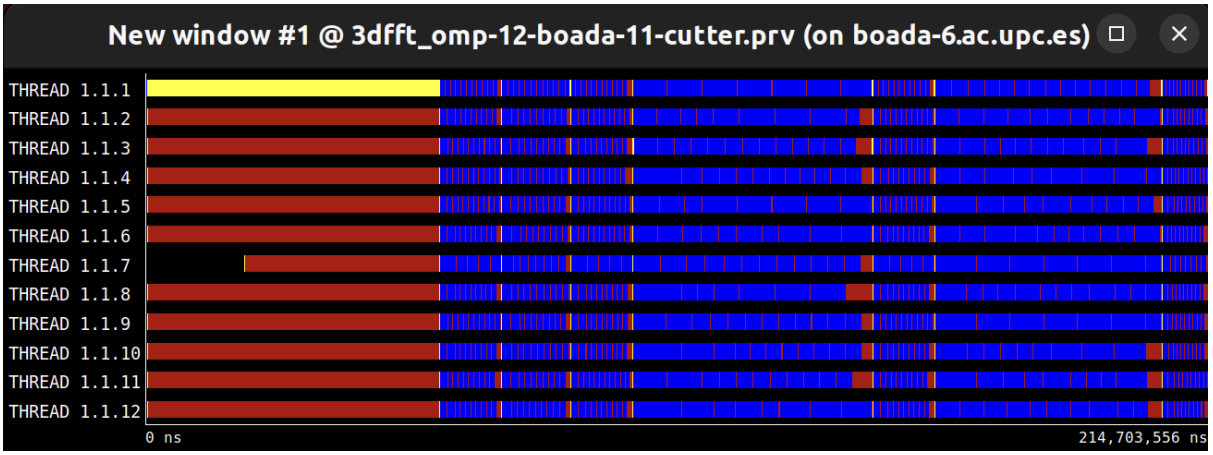


Fig. 3.8: Línea temporal ultima versión

Es en la última versión donde vemos el mejor tiempo de entre todas las versiones, un aumento del speedup y eficiencia (sobre todo con más threads) debido a que la función *init_complex_grid* deja de ejecutarse secuencialmente para pasar a ejecutarse paralelamente. Pasando ahora a un 99,97% de fracción paralelizable del programa.

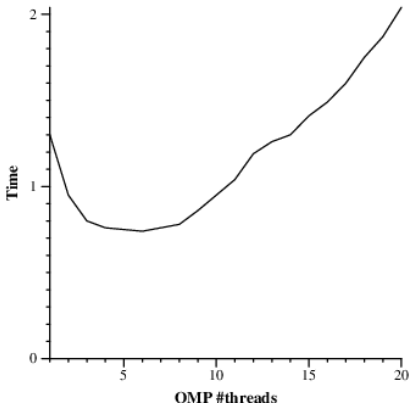
Fig. 3.9: v2 comparada con v0

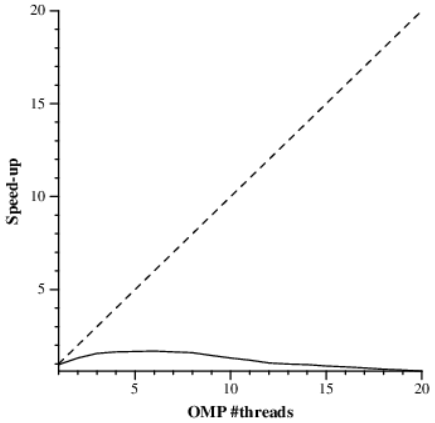
Version	φ	ideal S_{12}	T_1 (s)	T_{12} (s)	real S_{12}
initial version in 3dfft omp.c	0,83	5,88	1,31	1,28	1,02
new version with reduced parallelisation overheads	0,83	5,88	1.25	0,40	3,17
final version with improved φ	0,99	100	1.29	0,21	6,01

Fig. 3.10: Tabla comparativa versiones

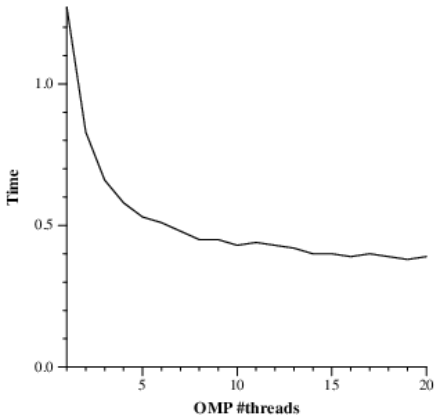
Se han usado los datos de *modelfactor* para completar la tabla, el Speedup ideal se ha calculado con: $S = 1 / (1-\varphi)$.

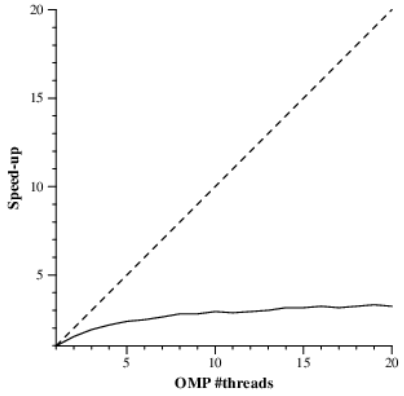
Initial version in 3dfft omp.c



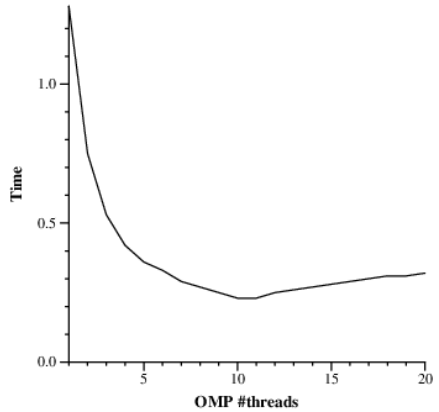


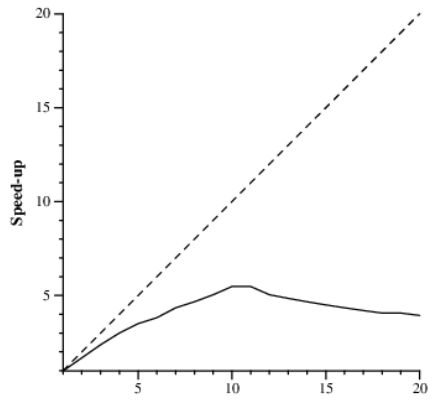
New version with reduced parallelisation overheads





Final version with improved φ





Finalmente obtenemos las gráficas de strong scalability de todas las versiones. Con la ayuda de los gráficos podemos observar mejor el aumento de eficiencia en cada una de las versiones.
Se puede ver como el número de threads útiles en cada versión va aumentando. Mientras que en la primera versión el tiempo aumenta drásticamente a partir de los 6 threads, en las siguientes, aparte de ser un tiempo bastante inferior, se estabiliza, indicándonos que el programa es más eficiente y paralelizable.

Aunque no lo parezca debido al número de páginas, el recuento de palabras es el siguiente: