PAR Laboratory Assignment

ÍNDICE

Session 1: Experimental Setup **2**

Node architecture and memory 2

Serial compilation and execution 4

Compilation and execution of OpenMP programs 4

Strong vs. weak scalability 5

Session 2: Systematically analysing task decompositions with Tareador **7**

Exploring new task decompositions for 3DFFT 7

Versión 1 8

Versión 2 9

Versión 3 9

Versión 4 10

Versión 5 11

Session 3: Understanding the execution of OpenMP programs **12**

Initial version 13

Improving φ 13

Reducing parallelisation overheads 14

Conclusiones 17

Lab 1: Experimental Setup and tools

Session 1: Experimental Setup

El objetivo de la primera sesión es familiarizarnos con el hardware y software que utilizaremos a lo largo del semestre en las clases de laboratorio de PAR. Trabajaremos en boada, un servidor multiprocesador situado en el departamento de Arquitectura de Computadores. Este servidor tiene diferentes nodos: utilizaremos boada-1 para ejecutar interactivamente o boada-2 a boada-8 para ejecutar con el sistema de colas.

Node architecture and memory

Lo primero que hicimos fue investigar la arquitectura disponible de los diferentes nodos en boada. Para hacer esto ejecutamos in
teractivamente los comandos *lscpu* y *lstopo* para obtener información sobre el hardware en el servidor:

fig. 1: Tabla información del hardware de los nodos boada 1-4

	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	2395 MHz
L1-I cache size (per-core)	32 KB
L1-D cache size (per-core)	32 KB
L2 cache size (per-core)	256 KB
Last-level cache size (per-socket)	12288 K → 12 MB
Main memory size (per socket)	23 GB
Main memory size (per node)	12 GB

Para mostrar visualmente la arquitectura de un nodo ejecutamos el comando *lstopo* con la opción *--of fig map.fig* para crear una imagen en formato *.fig*. Esta imagen nos da una idea de la estructura que tiene un nodo. Para visualizarla utilizamos el comando *xfig map.fig*.

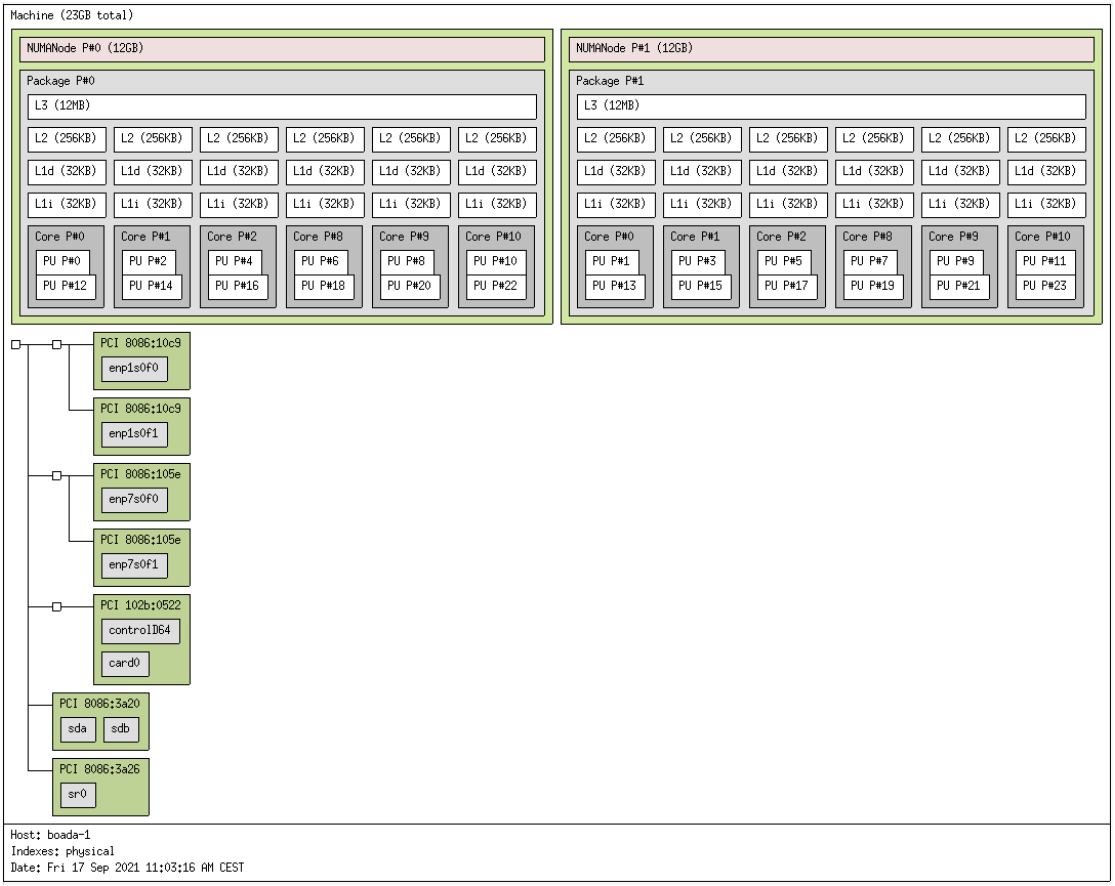


fig. 2: Arquitectura Boada 1-4

Serial compilation and execution

El objetivo de este apartado era ejecutar un programa que calcula el número pi a partir de un algoritmo concreto que nos daba el enunciado. Para ejecutarlo utilizamos dos ficheros diferentes:

- **run-seq.sh:** Se ejecuta en el boada-1 de manera interactiva.
- **submit-seg.sh:** Se ejecuta en los nodos boada-2 a boada-8 en la cola.

Compilation and execution of OpenMP programs

En este apartado vimos cómo compilar y ejecutar programas paralelos en *OpenMP*. El programa que nos da el enunciado se trata de un algoritmo que calcula el número pi. Para ejecutarlo, lo hemos hecho de dos formas: de forma interactiva usando el *script* run-omp.sh con 1, 2, 4 y 8 procesadores y de forma aislada con el sistema de colas usando el *script* submit-omp.sh.
Con el modo interactivo vimos que el número de procesadores se especifica con la variable OMP_NUM_THREADS y el resultado de la ejecución con 1.000.000.000 de iteraciones fue el siguiente:

fig. 3: Tabla diferencia entre ejecuciones interactivas o en cola

# threads	Interactive				Queued			
	user	system	elapsed	% of CPU	user	system	elapsed	% of CPU
1	7,97	0,00	0:03.99	199	3,93	0,01	0:04.00	98
2	7,97	0,00	0:03.99	199	3.95	0,00	0:02.00	197
4	7,98	0,00	0:03.99	199	3,98	0,00	0:01.01	392
8	7,97	0,00	0:03.99	199	4,17	0,01	0:00.54	763

Mientras que al ejecutarlo de forma interactiva los resultados se mantuvieron constantes, al ejecutarlo de forma aislada vimos cómo a medida que se añadían *threads*, se reducía tiempo de ejecución, y el % de CPU crecía más de un 90% de la última ejecución, sin llegar 100% a causa de los *overheads.*

Strong vs. weak scalability

Cuando en paralelismo hablamos de escalabilidad, nos referimos a la facilidad con la que podemos conseguir un mayor poder de cómputo aumentando recursos.
Para evaluarla, se tienen en cuenta dos factores:

- **Strong scalability**
  Capacidad del sistema de reducir tiempo de ejecución al añadir potencia de procesamiento.

- **Weak scalability**
  Capacidad del sistema de mantener el tiempo de ejecución aumentando la potencia computacional

En esta sección estudiamos la escalabilidad de la versión paralelizable del programa *pi_omp.*
Para ello pusimos en la cola de ejecución el programa `submit-strong-omp.sh` usando el comando `sbatch -p execution ./submit-strong-omp.sh.`
Este programa ejecuta *pi_omp* varias veces aumentando el número de threads y grafica el resultado.
Al acabar la ejecución, utilizamos el comando `gs` y obtuvimos las siguientes gráficas:

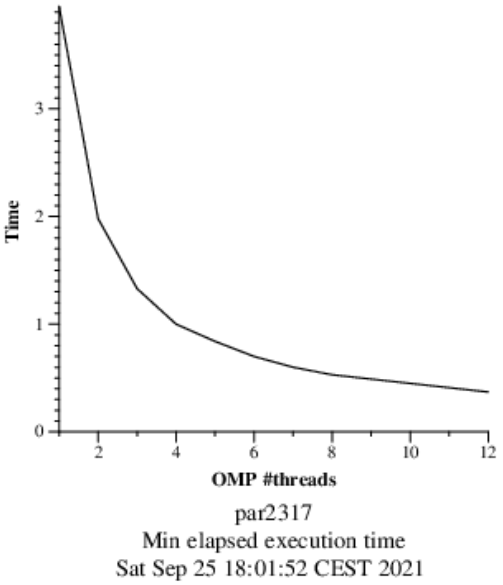


fig. 4: Postscript strong scalability tiempo

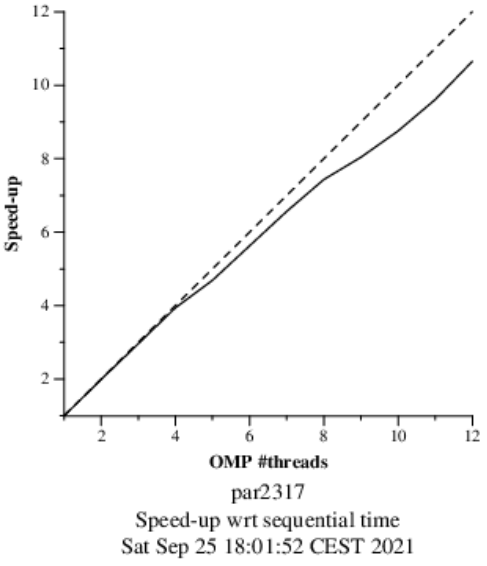


fig. 5: Postscript strong scalability speed-up

Podemos ver en la gráfica del tiempo de ejecución, como este se reduce de forma logarítmica, cosa que nos hizo pensar que habría un número de threads a partir del cual dejaría de reducirse el tiempo de ejecución.
En la gráfica del Speed-up podemos observar que, aunque cada vez se distancia más de la recta ideal, aumenta de forma prácticamente lineal.
Después editamos el archivo e *`submit-strong-omp.sh`* ampliando el número máximo de threads (*np_NMAX=24*) con los que se ejecutaría el programa y volvimos a ejecutar el script, en esta ocasión obtuvimos las siguientes gráficas:

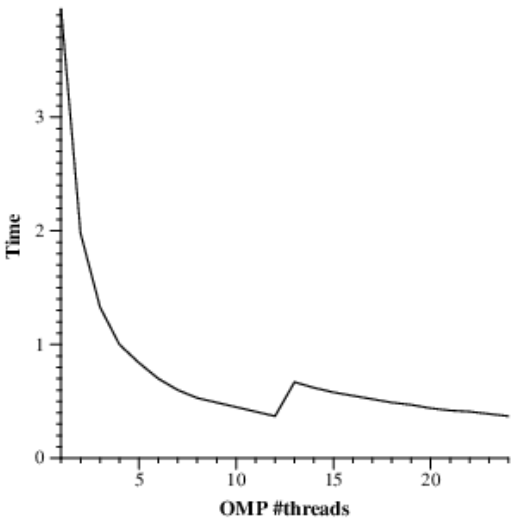


fig. 6: Postscript strong scalability tiempo (np_NMAX=24)

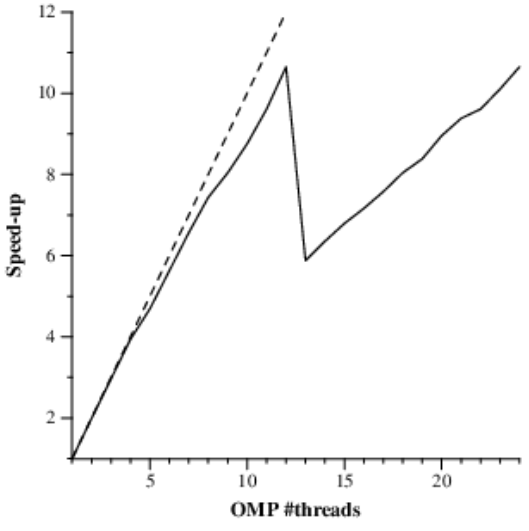


fig. 7: Postscript strong scalability speed-up (np_MAX=24)

Podemos ver en la gráfica del tiempo de ejecución, como este se reduce de forma logarítmica hasta llegar a 12 threads, punto a partir del cual el tiempo aumenta.
En la gráfica del Speed-up podemos observar que aumenta de forma prácticamente lineal hasta llegar a los 12 threads, después disminuye.
Esta bajada de rendimiento se debe a que se está forzando una paralelización innecesaria que añade al tiempo de ejecución actual el tiempo de los overheads sin aumentar el paralelismo.

Para poner a prueba la *weak scalability* del programa ejecutamos el script `submit-weak-omp.sh` y como anteriormente usamos el comando `gs` para conseguir la gráfica:

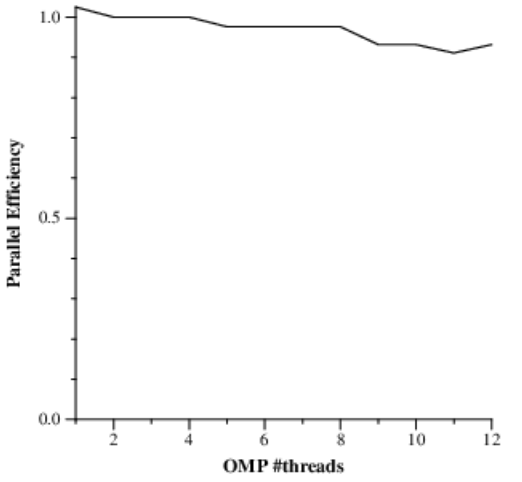


fig. 8: Postscript weak scalability paralelismo

Aquí se muestra como el paralelismo del sistema no es perfecto y va aumentando a medida que se aumentan los threads debido a los *overheads*.

A partir del comportamiento observado en las gráficas, podemos concluir que aumentar el número de threads por sí solo no implica una mejora en el rendimiento, hay que tener en cuenta los *overheads* y como de paralelizable es el programa

Session 2: Systematically analysing task decompositions with *Tareador*

El objetivo de esta segunda sesión es ver el funcionamiento de *Tareador* , una herramienta que sirve para analizar el posible paralelismo obtenido cuando se aplica una determinada estrategia de descomposición de tareas al código secuencial que queremos paralelizar. En este entorno podemos simular la ejecución de un programa utilizando el número de *cores* que convengan.
En esta sesión trabajamos únicamente con el fichero 3dfft_tar.c que contiene un código con varias funciones y muchos bucles.

Exploring new task decompositions for *3DFFT*

Primero compilamos y ejecutamos el código de 3dfft_tar.c con *Tareador*. Para hacer esto usamos el script run-tareador.sh con el comando `./run-tareador.sh 3dfft_tar` y obtenemos el siguiente grafo de dependencias:

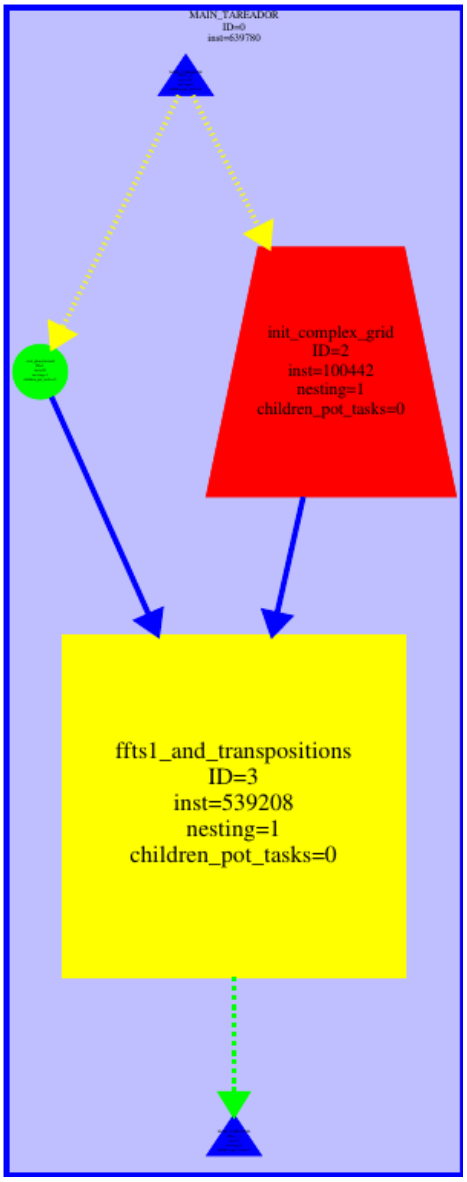


fig. 9: Grafo de dependencias de la versión secuencial

Para ver el paralelismo con diferente número cores, simulamos la ejecución con la herramienta Paraver. Vemos que con 1 core el tiempo de ejecución (T1) es de 639780001 ns. Con 1 core se ejecuta linealmente y, por lo tanto, no se produce paralelismo. El tiempo de ejecución con "infinitos procesadores" (T∞), es decir, el tiempo con el número máximo de cores el qual se limita el tiempo aunque aumentemos más procesadores, vemos que es de 639780001 ns, por lo tanto el programa no aprovecha nada del paralelismo de la arquitectura. El enunciado nos pide modificar el código de 3dfft_tar.c para hacer 5 versiones diferentes y analizar el paralelismo en cada una. De estas versiones obtuvimos los siguientes resultados representados en la tabla siguiente:

fig. 10: Tabla T_1, T_∞, Paralelismo por versiones

Version	T_1 (ns)	T_∞ (ns)	Parallelism
seq	639780001	639780001	1
v1	639780001	639780001	1
v2	639780001	361420001	1.77
v3	639780001	154354001	4.15
v4	639780001	64029001	9.99
v5	639780001	55820001	11,45

Versión 1

En esta versión modificamos el código reemplazando la tarea *ffts1_and_transpositions* añadiendo una tarea para cada una de las funciones del código. Este cambio no provoca ninguna alteración en los tiempos de ejecución y, por lo tanto, tampoco en el nivel de paralelismo, puesto que cada función tiene que esperar a la función anterior para ejecutarse ya que dependen de datos que se modifican en la anterior función.
En la figura 6 podemos ver gráficamente como en esta versión cada función depende los datos de su anterior y, por lo tanto el paralelismo no varía.

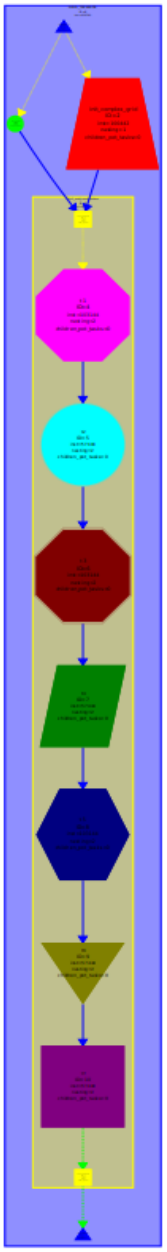


fig. 11: Grafo de dependencias de la versión 1

Versión 2

En la versión 2, modificamos el código de la función *ffts1_planes* creando nuevas tareas por cada iteración del bucle k, gracias a esto, la función se descompone y se ejecutan sus diferentes tareas en paralelo, reduciendo así su tiempo de ejecución prácticamente a la mitad.
En este caso, a diferencia de la *versión 1,* si que tenemos partes paralelizadas, que aparecen representadas por las filas de bloques amarillos de la figura 7.

Versión 3

Partiendo del código de la versión 2, reemplazamos el código de las funciones *transpose_xy_planes* y *transpose_zx_planes* y creamos tareas en cada una de las iteraciones de los bucles k para poder reducir la granularidad y dividir la carga de procesamiento en diferentes tareas.
Gracias a este aumento en la paralelización, el tiempo de ejecución se ve reducido respecto a las versiones anteriores.
En la figura 8 se muestra

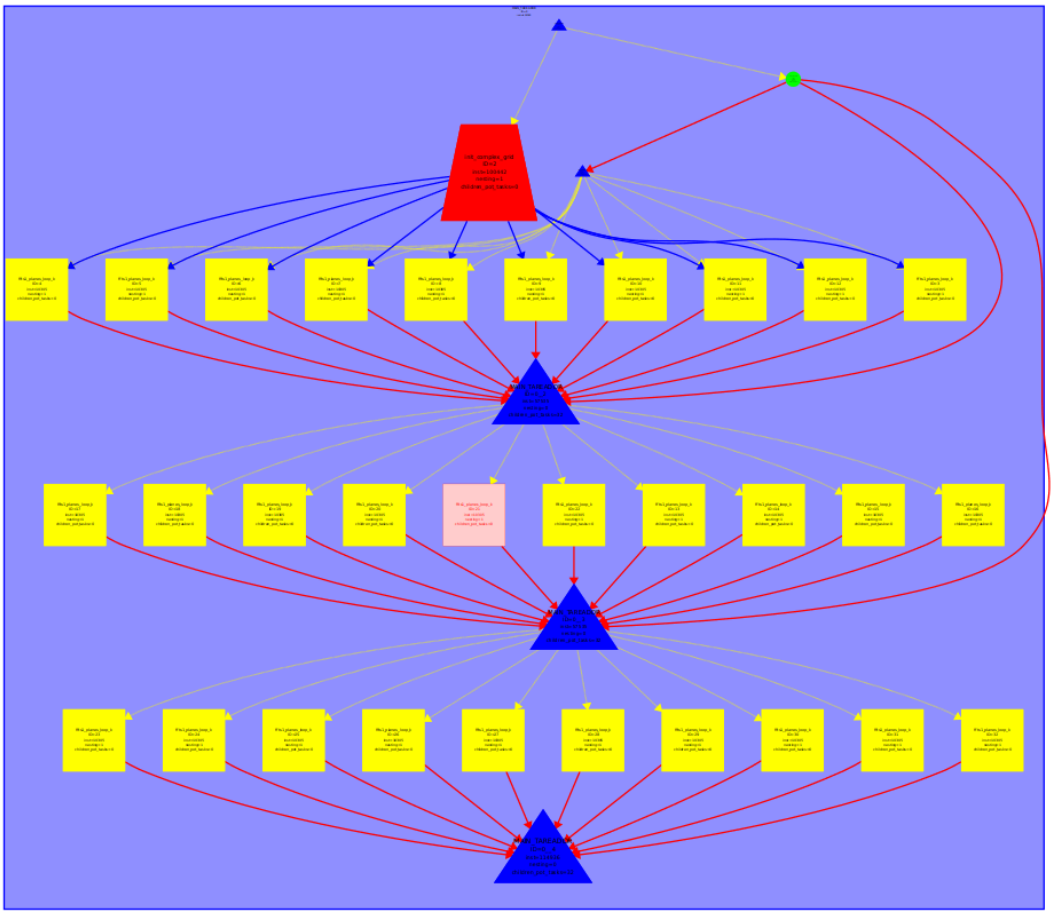
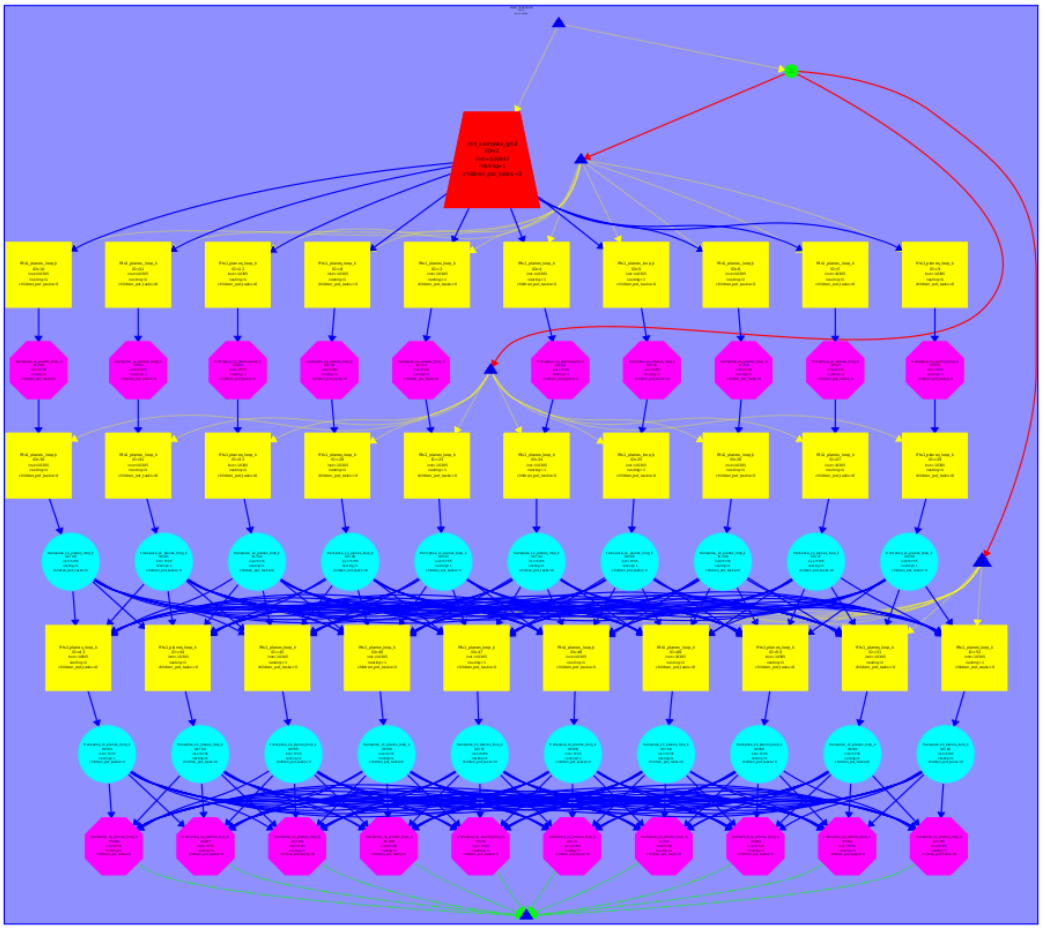


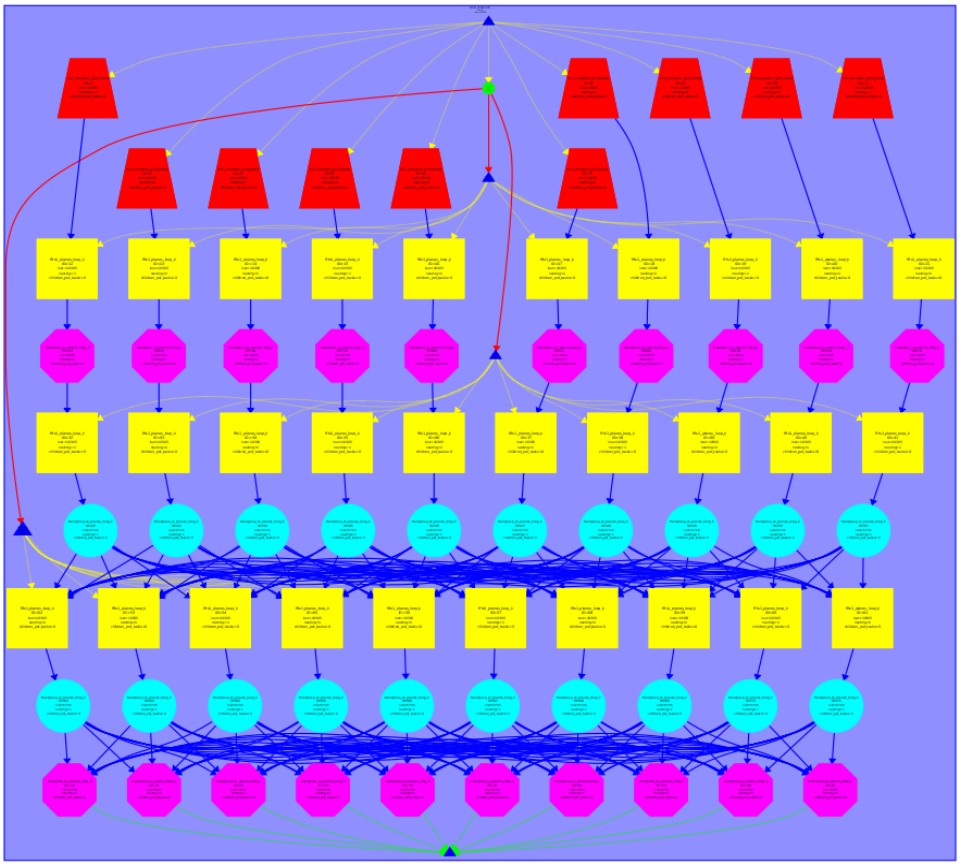
fig 12: Grafo de dependencias de la versión 2



Versión 4

En esta versión modificamos el código de la versión 3 y aplicamos la misma granularización en el bucle exterior (bucle k) de la función *init_complex_grid*. El tiempo de ejecución infinito es de 64029001 ns, es decir, menos de la mitad que el tiempo infinito de la versión anterior y, por lo tanto, el paralelismo es más del doble respecto al anterior.
La siguiente gráfica muestra el tiempo de ejecución si cambiamos el número de procesadores (1, 2, 4, 8, 16, 32). Vemos que al llegar a 16 procesadores el tiempo es constante aunque aumentemos el número de *threads*, es decir, este tiempo es T∞.

fig. 14: Tiempo de ejecución cambiando el número de procesadores (versión 4)



Versión 5

En esta última versión modificamos el código de la versión 4 cambiando la granularización del bucle exterior k al bucle j en la función *init_complex_grid*. Primero probamos de hacerlo en el bucle i, ya que pensamos que sería el nivel de granulación que permitiría reducir el tiempo al mínimo, pero vimos que tardaba demasiado en ejecutarse ya que lo hacía en cada iteración del bucle más interno, así que decidimos hacer la granulación en el bucle j.
El siguiente gráfico (fig. 12) nos muestra, al igual que la versión anterior, el tiempo de ejecución si cambiamos el número de procesadores (1, 2, 4, 8, 16, 32).

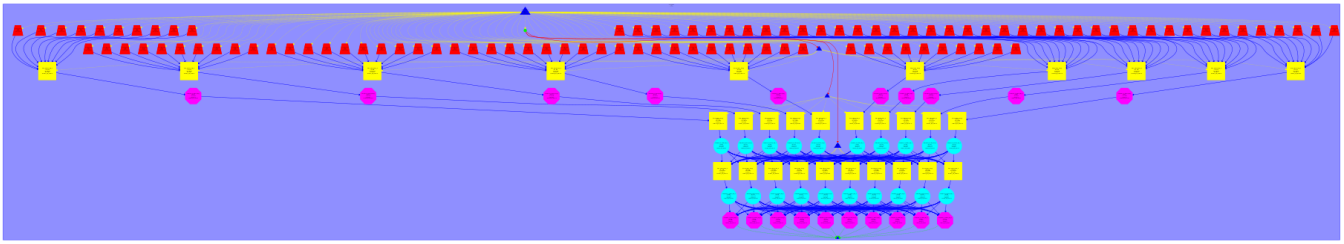


fig. 16: Grafo de dependencias de la versión 5

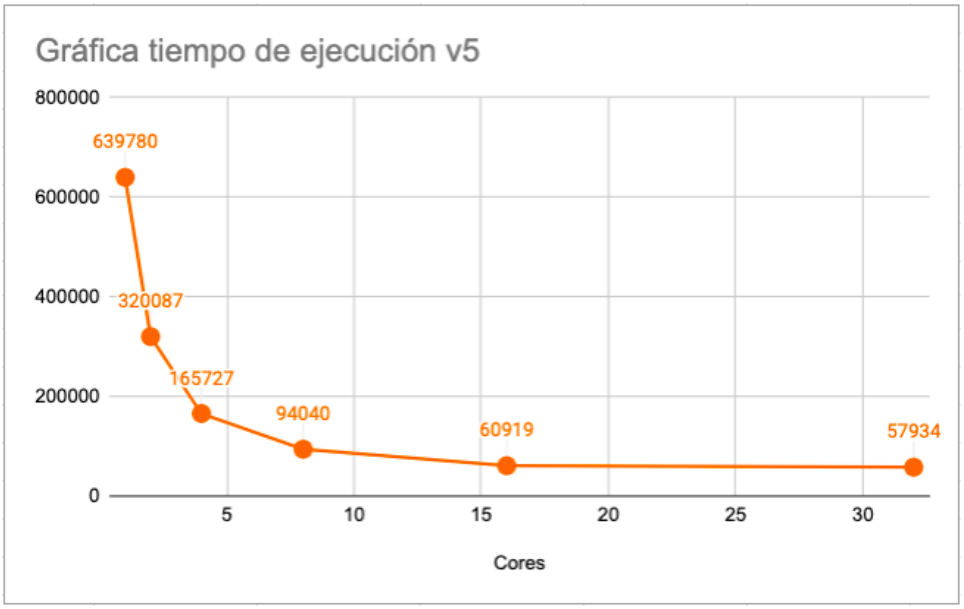


fig. 17: Tiempo de ejecución cambiando el número de procesadores (versión 5)

Vemos que el gráfico es muy parecido al de la versión 4, pero hay alguna pequeña diferencia si nos fijamos bien en los valores. En esta última versión al crear tareas con una menor granulación vemos que es capaz de escalar más *threads* que la anterior versión, por lo que al llegar a 16 procesadores el tiempo de ejecución no es constante del todo como antes.

Session 3: Understanding the execution of *OpenMP* programs

El objetivo de esta última sesión es familiarizarnos con el entorno de *Paraver*, una herramienta que nos permite recoger información y visualizar la ejecución de un programa paralelo en *OpenMP*.
En esta sesión trabajamos únicamente con el fichero *3dfft_omp.c* y lo ejecutamos con el script sumbit-extrae.sh con 1 y 8 procesadores. Lo ejecutamos con el comando *sbatch* y este genera 3 ficheros: .prv, .pcf, .row. De estos abriremos con el *Paraver* el .prv para ver gráficamente cuando el programa se ejecuta paralelamente o secuencialmente.
La siguiente tabla muestra los resultados que hemos obtenido en las 3 versiones del fichero 3dfft_omp.c que hemos ejecutado:

fig. 18: Gráfico de tiempo de la versión inicial

Version	φ	**ideal** S_8	T_1 (ns)	T_8(ns)	**real** S_8
initial version in 3dfft omp.c	0,70	3,365247531	2530946327	1461185608	1,732118297
new version with improved φ	0,82	5,697024501	2472595587	1131048199	2,186109831
final version with reduced parallelisation overheads	0,91	11,59770233	2441782127	664255285	3,675969438

Hemos calculado cada uno de los parámetros a partir de la información que nos proporciona *Paraver* de las siguientes fórmulas:
$T_1=T_{seq}+T_{par}$ $S_P=S_1/S_P$ $\varphi=T_{par}/T_1$ $S_\infty=1/(1-\varphi)$

Initial version

Al ejecutar la versión inicial del fichero hemos obtenido los resultados de la primera fila de la tabla anterior. En esta versión el rendimiento real y el ideal son mucho menores a las otras versiones ya que la función *init_complex_grid* no se paraleliza.
El gráfico de tiempo obtenido con 8 threads es el siguiente:

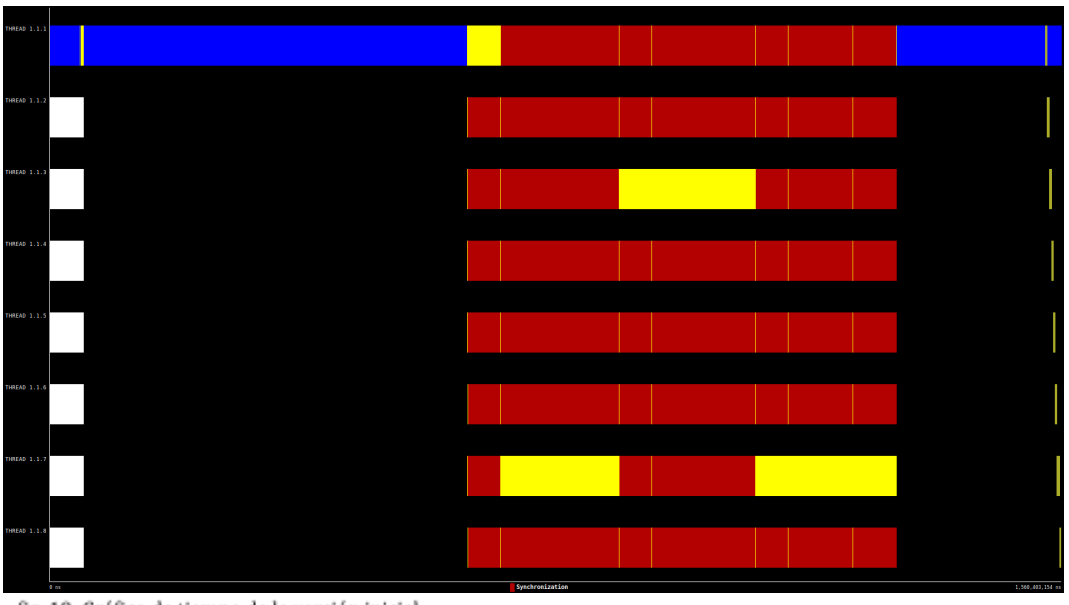


fig. 19: Gráfico de tiempo de la versión inicial

Si vamos la opción *Flags* y seleccionamos la opción *Parallel functions*, *Paraver* nos muestra el periodo de tiempo donde el programa se ejecuta en paralelo, es decir, podemos saber con facilidad la T_{par}

Improving φ

En este caso descomentamos las líneas de código que permiten paralelizar la función *init_complex_grid,* calculamos los diferentes parámetros para rellenar la tabla y obtuvimos el siguiente gráfico con *paraver* :

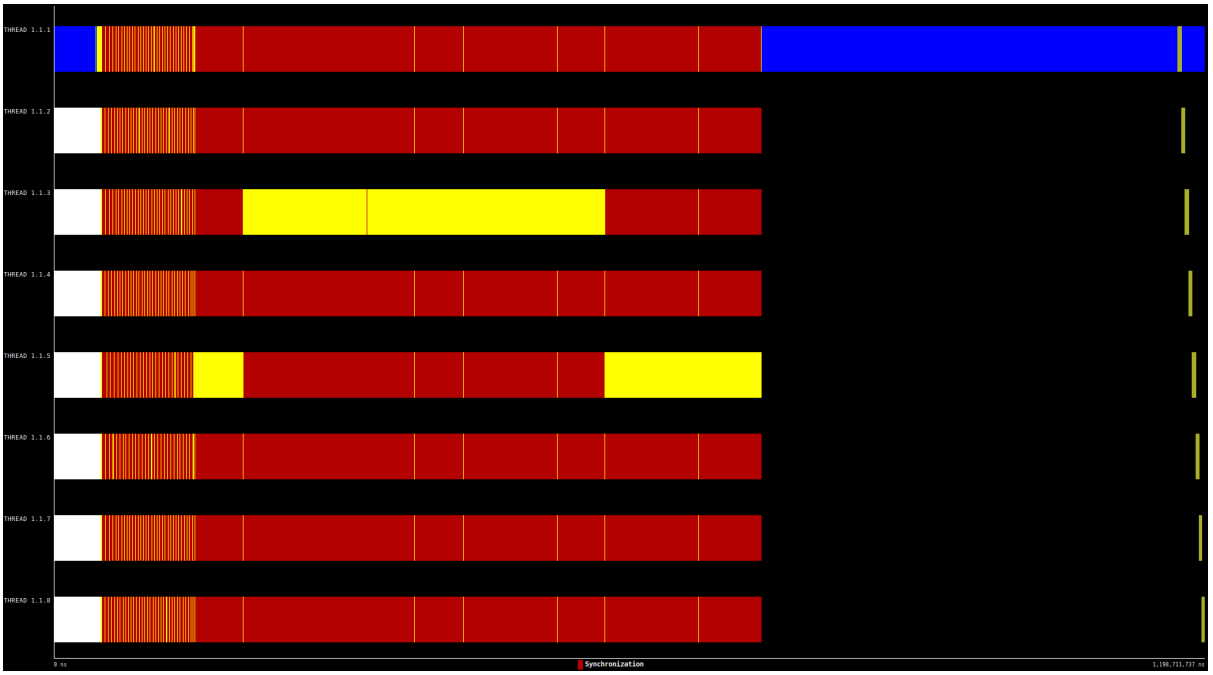


fig. 20: Gráfico de tiempo de la versión φ mejorada

Observamos que en este caso el porcentaje de tiempo que se ejecuta en paralelo es mayor que en la versión anterior y aunque no coincide con el ideal por culpa de los overheads, vemos una mejora en la eficiencia.

Reducing parallelisation overheads

En esta última versión del fichero *3dfft_omp.c* exploramos la posibilidad de aumentar la granulación de las tareas, activando el *taskloop* en cada una de las 4 funciones. Con esto obtenemos una mejora importante de rendimiento y de tiempo. Aún así, el φ sigue siendo parecido al de la versión anterior.
A continuación se muestra el gráfico de tiempo de la ejecución del código con 8 threads.

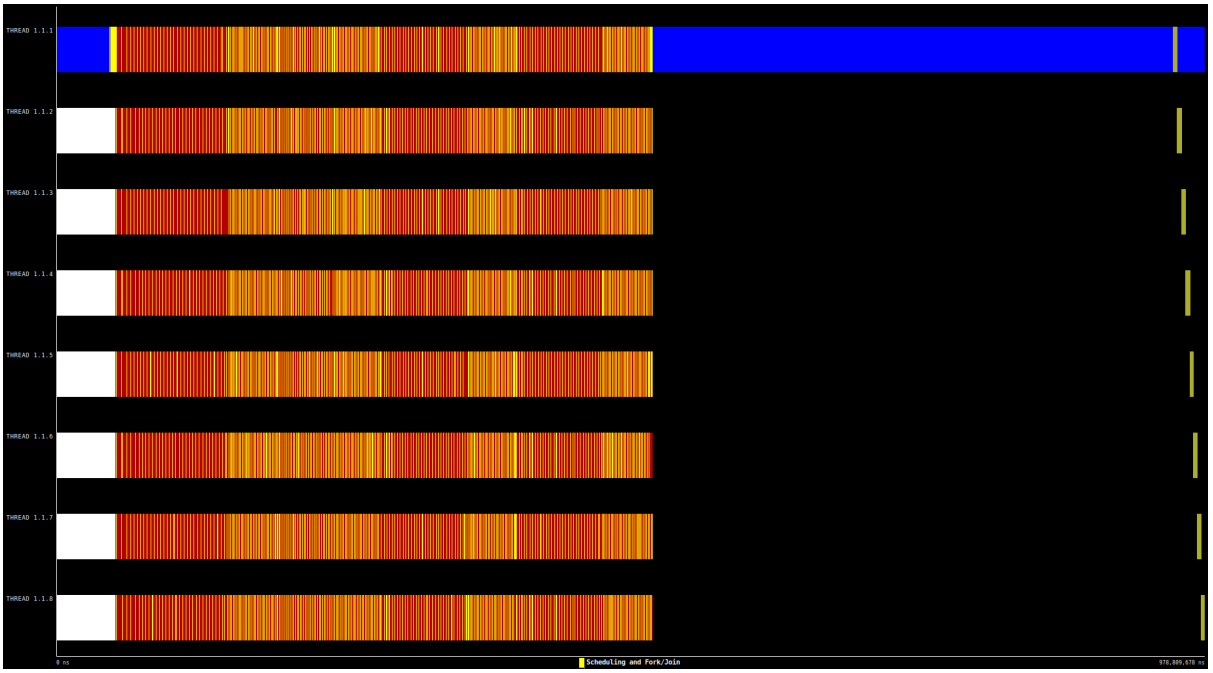


fig. 21: Gráfico de tiempo de la versión final

Por último, hemos obtenido las gráficas de strong scalability de cada una de las versiones:

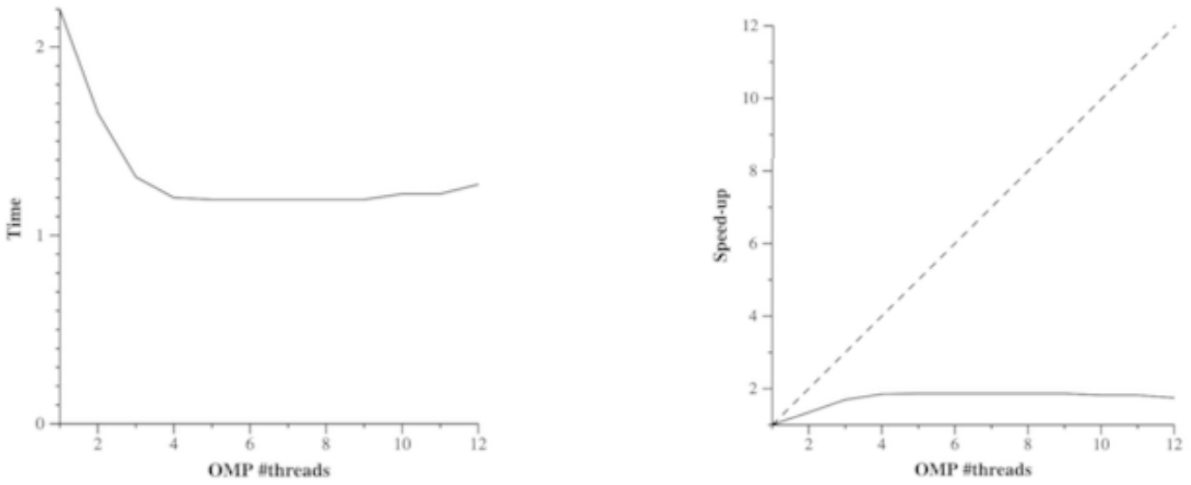


fig. 22: Strong scalability de la versión inicial

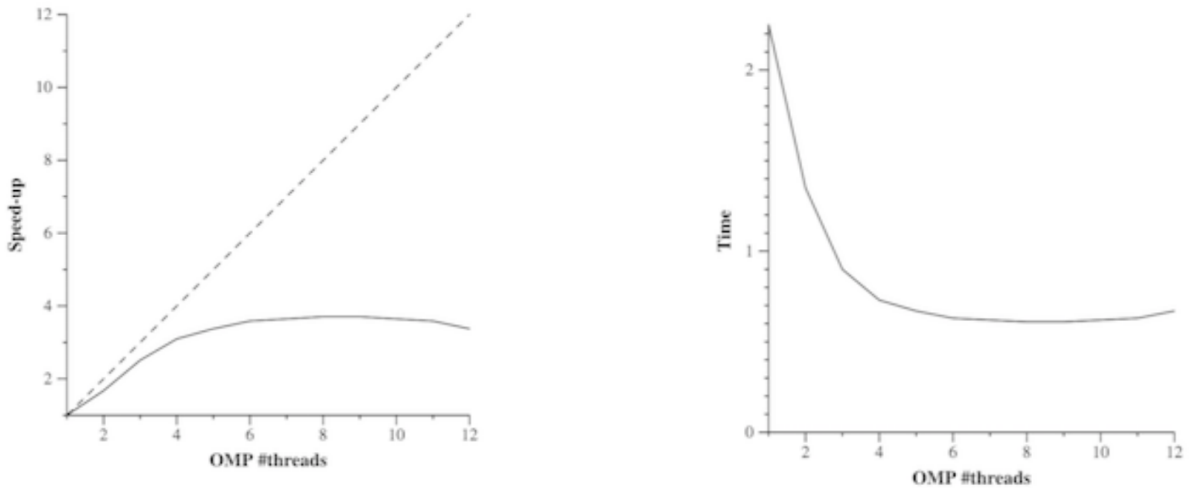


fig. 23: Strong scalability de la versión φ mejorada

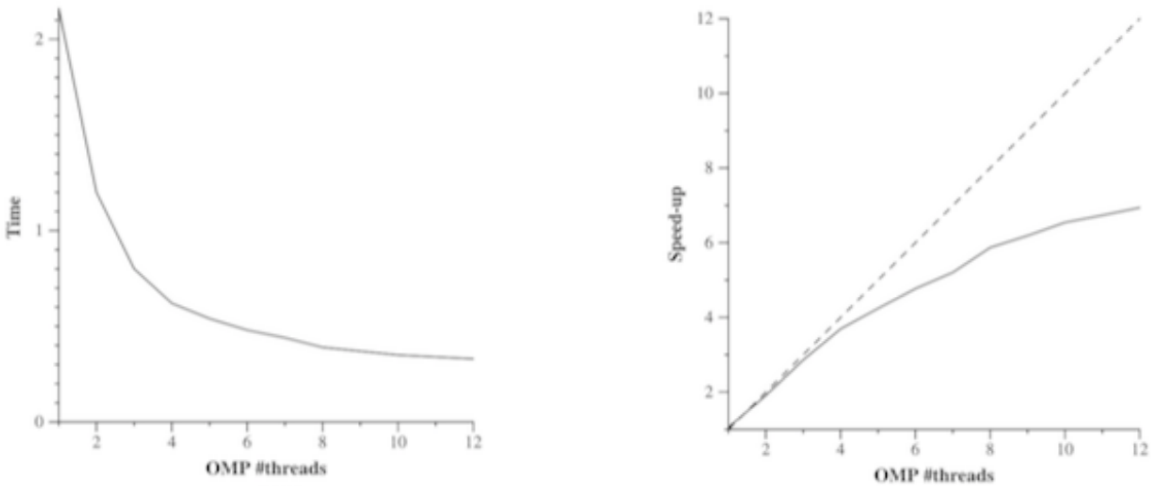


fig. 24: Strong scalability de la versión final

Gracias a los gráficos, podemos ver representada la mejora en eficiencia de cada una de las versiones.

Se aprecia un aumento en el número de threads útiles en cada nueva versión.

Mientras que en la versión inicial, el tiempo de ejecución y el speed-up se estabilizan con 4 threads, tanto en la segunda como en la tercera este número aumenta, cosa que nos indica que el programa es más paralelizable y más eficiente.

Conclusiones

En la primera parte de esta entrega, nos hemos informado sobre la arquitectura de *boada* y hemos explorado algunas de las opciones y funciones que nos ofrece este entorno en el que trabajaremos durante el curso. También usamos herramientas de visualización que nos han ayudado a ver los cambios en la ejecución de algunos programas, al aplicar diferentes grados de paralelismo. Gracias a estas herramientas pudimos analizar de forma visual la escalabilidad de un programa graficando su comportamiento al ejecutarlo varias veces con diferente número de threads.
Hemos llegado a la conclusión de que aumentar el número de threads sin tener en cuenta el grado de paralelismo de un programa puede llevar a usar recursos innecesarios e incluso aumentar el tiempo de ejecución. También hemos podido apreciar el gran impacto que tiene el paralelismo en la eficiencia de un programa.

En la segunda sesión seguimos trabajando en el estudio de la paralelización de los programas, pero esta vez centrándonos más en las dependencias que ocurren en la ejecución.
Partiendo de una versión secuencial, fuimos aumentando la complejidad del programa paralelizandolo y graficamos los grafos de dependencias usando el programa *Tareador:*
Con este software pudimos simular la ejecución del código y ver cómo el sistema operativo repartiría la carga de trabajo entre los diferentes threads, así como su grafo de dependencias.
En esta sección experimentamos con la granularidad de las tareas y los cambios que suponía en los grafos de dependencia y tiempos de ejecución. Observando los resultados dedujimos que es fundamental escoger de forma inteligente el nivel de granularidad y las partes del programa que se pretenden paralelizar.

Finalmente, en la tercera parte, incorporamos una nueva herramienta de visualización llamada *Paraver,* con la que generamos líneas de tiempo en las que podíamos ver, representado con barras de colores, el trabajo realizado por cada thread en cada instante.
Al igual que en las otras sesiones, hicimos diferentes modificaciones al código con tal de mejorar su eficiencia y esta vez, además, sacamos alguna información extra, como los tiempos de ejecución en paralelo y secuencial, mediante la nueva aplicación.
En esta última parte, entendimos cómo se ejecutaban los programas *OpenMP* y nos familiarizamos con una herramienta de análisis que nos permitió simular ejecuciones de programas paralelos y visualizarlos.