PAR Laboratory Assignment
Laboratori 1: Experimental Setup and tools

Professor:
Jordi Tubella

Alumnes:
Eric Hurtado
Jordi Pitarch

Índice

1. Introducción

Los diferentes objetivos que pretende este primer laboratorio de la asignatura de Paralelismo son:

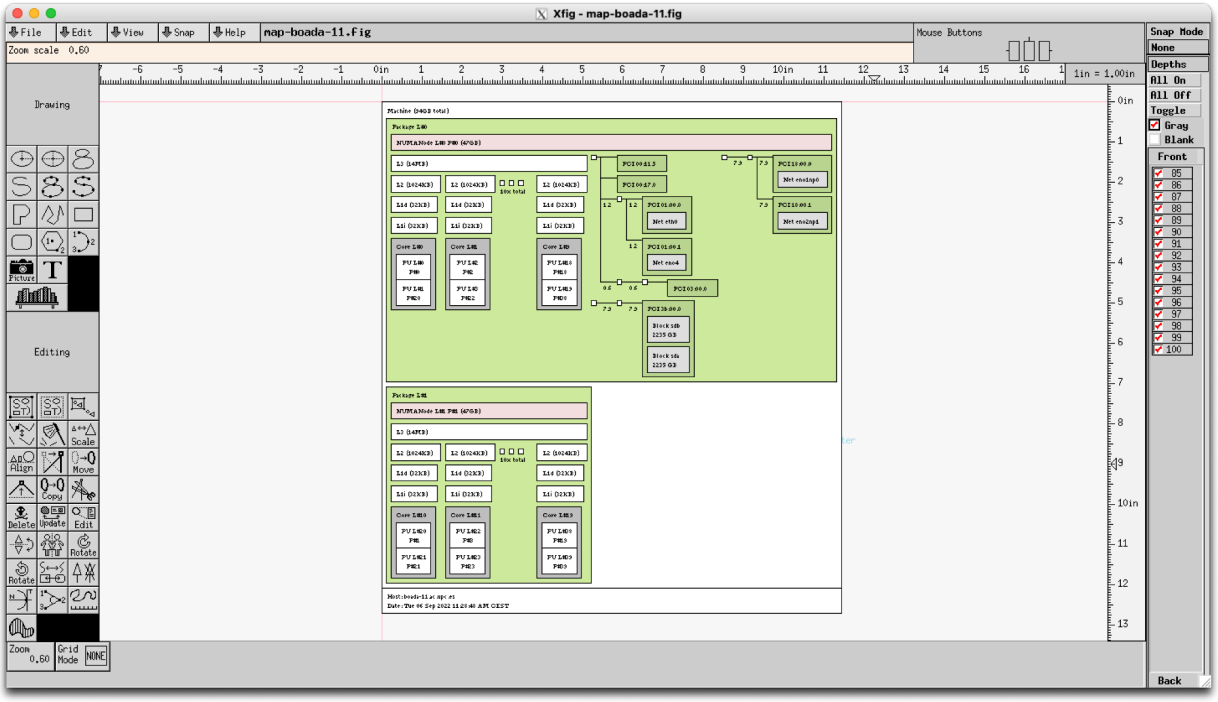
1. Descubrir la arquitectura de los diferentes nodos del “boada” y entender sus principales parámetros.
2. Compilar y ejecutar secuencialmente y código paralelo en OpenMP en nuestros nodos
3. Aprender a usar “Tareador”
4. Aprender a usar “Modelfactors”
5. Aprender a usar “Paraver”
6. Aplicar paralizaciones mediante las herramientas anteriores

2. Experimental setup

2.1 Node architecture and memory

La primera tarea a hacer ha sido investigar la arquitectura de los nodos disponibles en boada insertando el comando **sbatch submit-arch.sh** que pone en cola un script que ejecuta los comandos **lscpu** y **lstopo** para obtener la información sobre el hardware.

	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	2400MHz
L1-I cache size (per-core)	32KB
L1-D cache size (per-core)	32KB
L2 cache size (per-core)	1024KB
Last-level cache size (per-socket)	14MB
Main memory size (per socket)	47GB
Main memory size (per node)	94GB



Arquitectura del boada

2.2 Execution modes: interactive VS queued

Aquí se muestra que hay dos formas de ejecutar programas en boada, poniéndolo en cola o bien interactivamente.

La primera opción se puede llevar a cabo con el comando **sbatch [-p partition] ./submit-xxxx.sh**, aquí el programa se ejecuta cuando hay algún nodo disponible.

La otra opción la podemos realizar a partir del comando **./run-xxxx.sh** y aquí, a diferencia de la opción anterior, el programa empieza a ejecutarse inmediatamente compartiendo recursos con otros.

2.3 Serial compilation and execution

En este apartado entenderemos cómo se compila y ejecuta una aplicación secuencial con el código que se nos ha dado, un programa bastante simple que calcula en número pi de forma secuencial.
Para ello ejecutamos el programa interactivamente con el comando **run-seq.sh** con los argumentos apropiados, o bien lo ponemos en cola con el comando **submit-seq.sh** con los argumentos apropiados y esto nos dará información como el tiempo del CPU o el porcentaje utilizado.

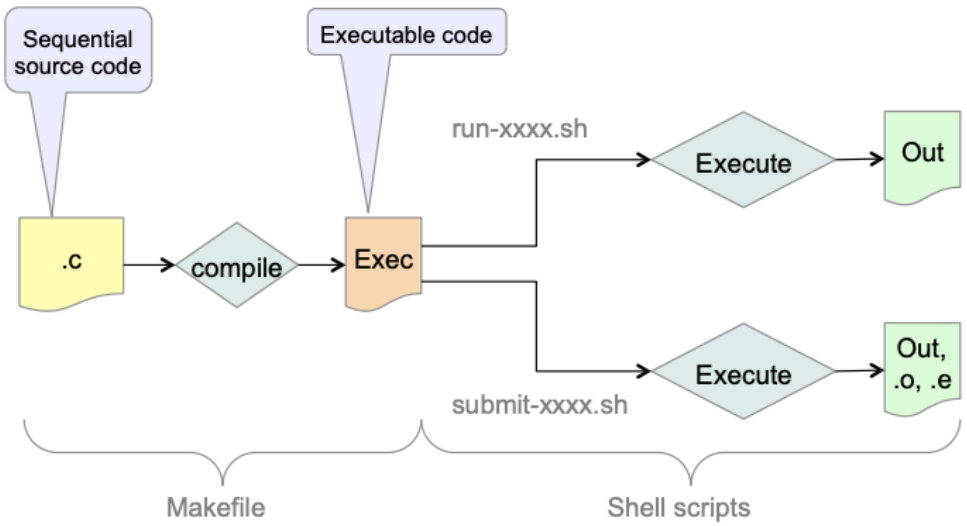


Gráfico de cómo se compila y ejecuta de forma secuencial

2.4 Compilation and execution of OpenMP programs

OpenMP es un programa de programación paralela, que nos permite usar memoria compartida.

Para poder observar las diferencias podemos ejecutar el programa de pi, tanto en modo secuencial, como en modo paralelo, para ver las diferencias.

En la siguiente tabla calculamos el user y system CPU time, el elapsed time y el % de CPU utilizados tanto en interactiva, como en cola, con el número de threads utilizados. La gran diferencia que se puede observar es que se utiliza un mayor % de CPU en el caso de la cola y tanto el usuario, el system y el elapsed time es menor para la cola.

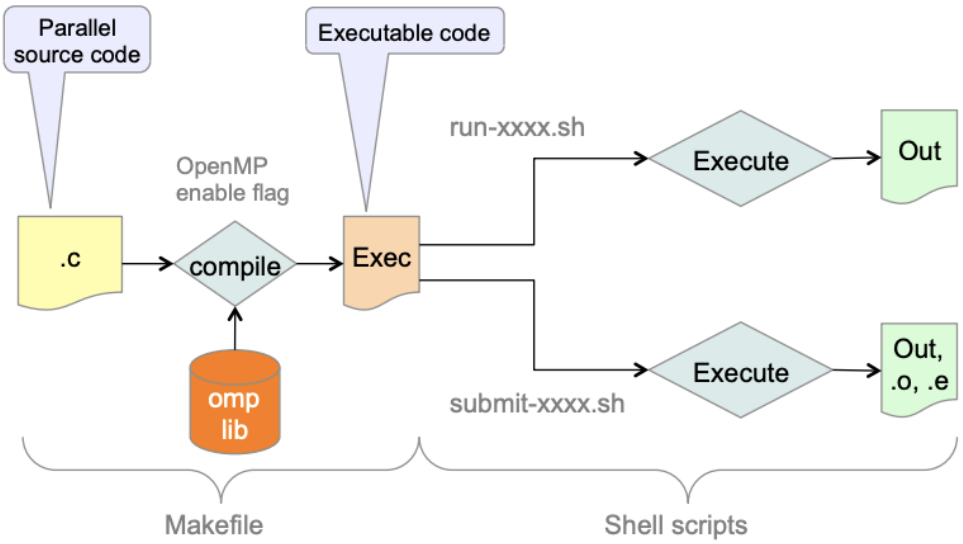


Gráfico de cómo se compila y ejecuta programas con OpenMP

Threads	Interactiva				Cola			
	user	system	elapsed	%CPU	user	system	elapsed	%cpu
1	2,36	0,00	0:02.37	99%	0,69	0	0:00,7	98%
4	2.4	0,02	0:01,21	199%	0,7	0	0:00,19	363%
8	2,38	0,05	0:01,22	199%	0,75	0	0:00,11	657%
16	2,42	0,10	0:01,26	199%	0,78	0,01	0:00,07	1112%
20	2,47	0,14	0:01,31	199%	0,83	0	0:00,06	1333%

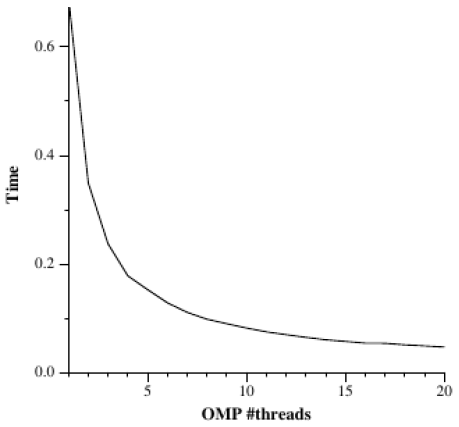
2.5 Strong vs. weak scalability

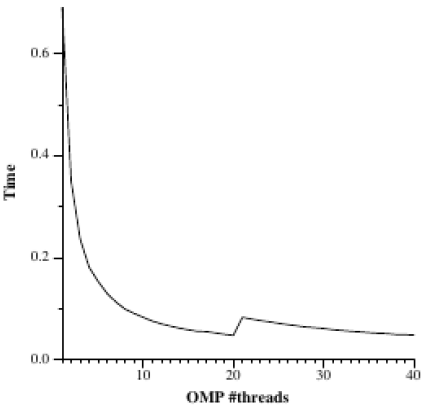
Cuando queremos hablar de escalabilidad en relación al paralelismo de un programa, se refiere a la facilidad de conseguir un mejor tiempo de ejecución aumentando el número de recursos disponibles.

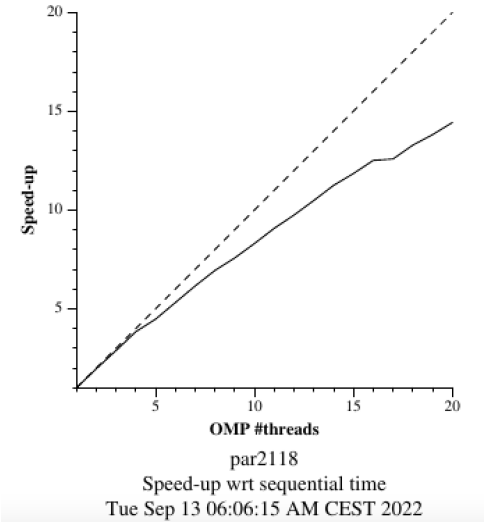
En esta sección del laboratorio, nos hemos basado en la versión paralelizable de *pi_omp*. Mediante la comanda de "**sbatch -p execution ./submit-strong-omp.sh**", y la posterior utilización del comando gs, hemos obtenido diferentes gráficas que nos ayudan a observar esta escalabilidad.

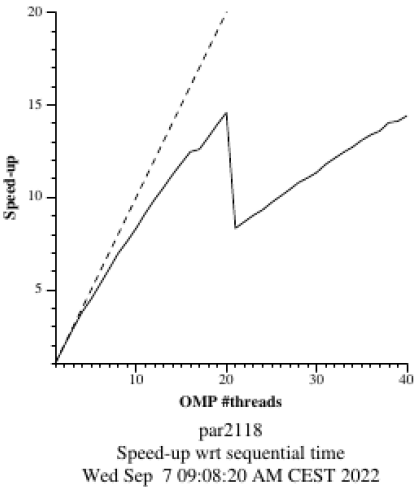
En la primera columna, en el primer gráfico, podemos observar que el tiempo de ejecución se va reduciendo de forma logarítmica. Mientras que en la segunda gráfica, la del speedup, podemos observar como cada vez se separa más de la línea discontinua, pero esta es casi lineal.

En la segunda columna, hemos aumentado el número de threads hasta 40 (np_MAX=40). Por lo que podemos observar en las gráficas, el tiempo de ejecución baja hasta llegar a los 20 threads, punto en el cual sube, para volver a bajar más lento. En lo que respecta al speedup, pasa algo parecido, debido a que este baja al llegar a los 20 threads, pero después vuelve a subir.

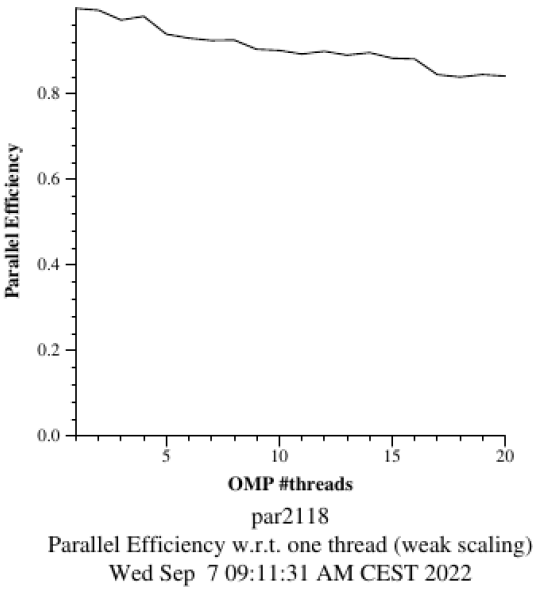








Por otra parte, hemos decidido poner a prueba también el "**submit-weak-omp.sh**". Como se puede apreciar en la gráfica posterior, el paralelismo del sistema no es perfecto y va aumentando a medida que se aumentan los threads debido a los overheads.



Gracias a las gráficas anteriores, hemos podido concluir que maximizar el número de threads únicamente, no implica una mejora del rendimiento del programa, debido a que hay que tener en cuenta overheads y como de paralelizable es el programa.

3. Systematically analysing task decompositions with Tareador

Para analizar las descomposiciones de tareas usaremos la herramienta Tareador que nos ayuda a observar entre otras cosas el grafo de dependencias, la visualización de los datos… lo que nos permite conocer la mejora aplicada a un código secuencial que queremos paralelizar.
Para familiarizarnos con el software trabajaremos con el código de *3dfft_tar.c*, proporcionado por la asignatura. En la tabla se muestran los datos recogidos de las diferentes versiones del código grácias a la herramienta mencionada anteriormente.

Versión	T_1	T_∞	Parallelism
v0	639.780.001	639.707.001	1,0001
v1	639.780.001	639.707.001	1,0001
v2	639.780.001	361.472.001	1,7699
v3	639.780.001	154.939.001	4,1292
v4	639.780.001	64.614.001	9,9015
v5	639.780.001	56.416.001	11,3403

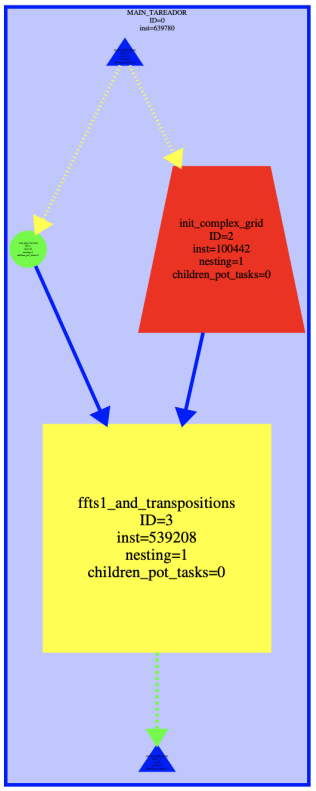
En la tabla vemos que T1 tendrá siempre el mismo valor, ya que al utilizar un solo procesador, el código se ejecutará siempre de forma secuencial. Por lo que al resto respecta los datos ya varían según la versión:

Versión 0:

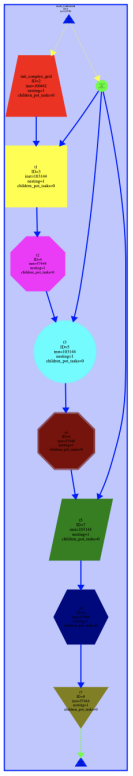
En la versión original del código, el Tareador comienza antes de las llamadas a las funciones y termina después de ellas.
En el grafo de dependencias se puede observar que existen dos caminos, sin embargo el tiempo de T∞ es prácticamente igual al de T1.

Versión 1:

En la versión 1 se modifica el código generando una tarea por cada función llamada por el main, esta modificación no provoca cambios en el tiempo de ejecución.



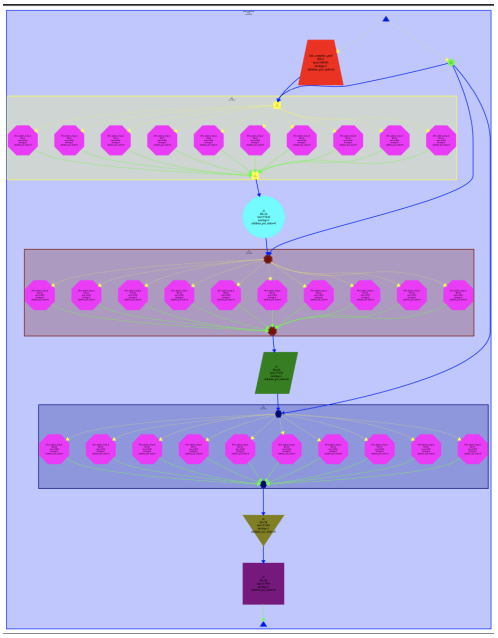
Versión 0



Versión 1

Versión 2:

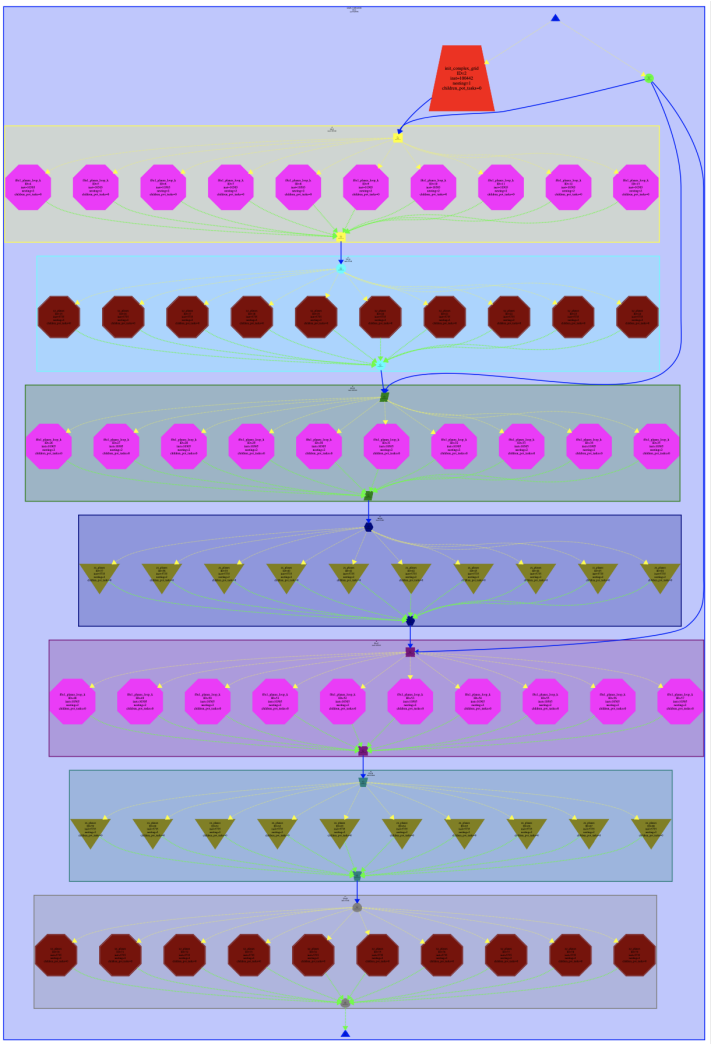
En la versión 2 se modifica un bucle de la función *ffts1_planes* para que se ejecute en paralelo, esto provoca que el T∞ se reduzca en casi la mitad de tiempo que en la versión anterior.



Versión 2

Versión 3:

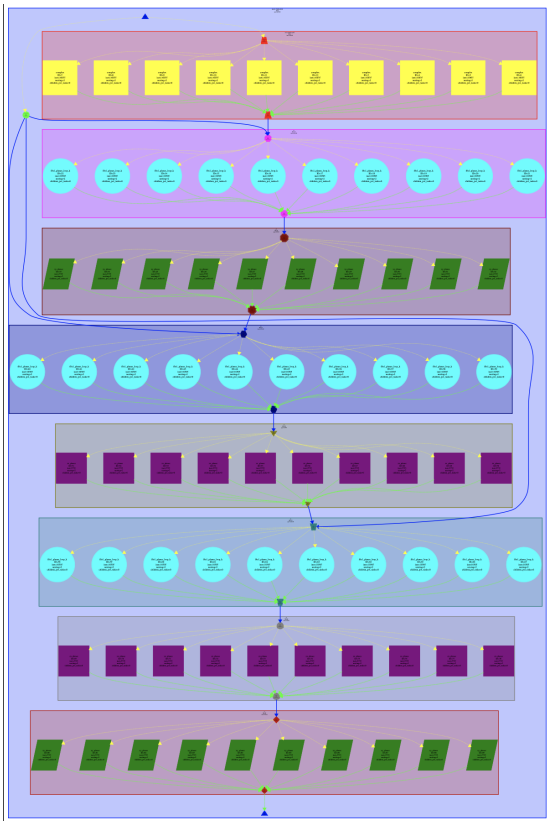
En la versión 3 hacemos lo mismo que hemos hecho en la 2 con el bucle de la función *ffts1_planes* pero esta vez también en las funciones *transpose_xy_planes* y *transpose_zx_planes*, esto ha provocado que el tiempo vuelva a reducirse en casi la mitad respecto a la versión anterior como puede observarse en la tabla.



Versión 3

Versión 4:

En la versión 4 hacemos que todas las tareas que se ejecutan dentro del bucle llamado "*init_complex_grid*", en el que tiene de variable auxiliar la variable "k", se ejecute cada una en una task diferente. Gracias a este cambio conseguimos mejorar casi 3 veces el tiempo obtenido en la versión anterior.



Versión 4

Versión 5:

En la versión 5 hacemos lo mismo que en la anterior, con la diferencia que en lugar de hacerlo en el bucle más externo, lo hacemos en un bucle más interno, en este caso el de la "j".

También hemos decidido hacerlo en este debido a que al hacerlo en el último más interno, hubiéramos tenido demasiadas tareas y esto hubiera provocado overheads mayores. Como consecuencia podríamos haber obtenido incluso tiempos mayores de ejecución.

Versión 5

Versión 4 vs Versión 5:

En este último apartado de la sesión 2, nos hemos querido centrar en las diferencias entre estas dos últimas versiones de "*3dfft_tar.c*".

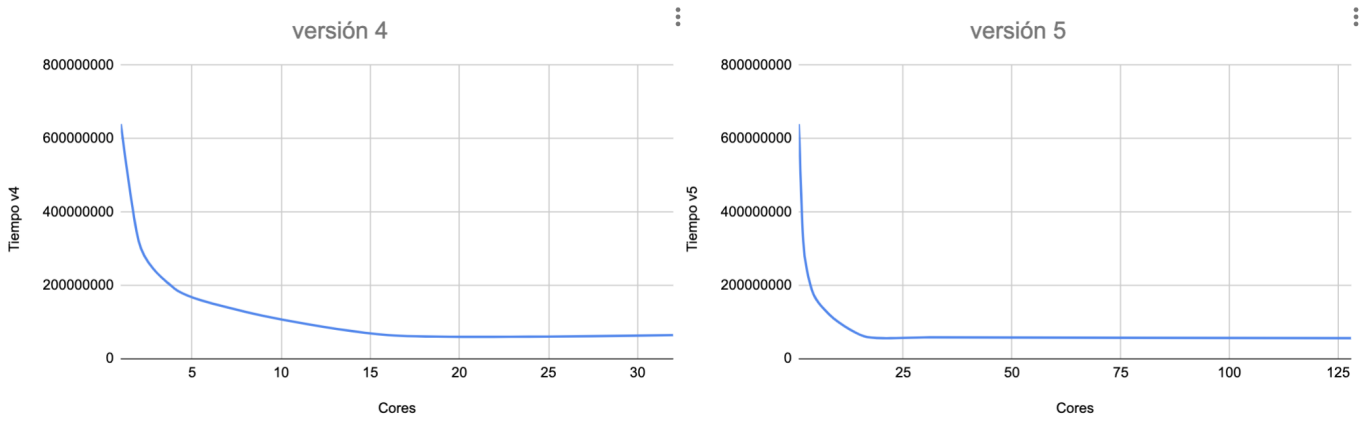
Por un lado vemos como vemos en el fragmento de excel, en la versión 4, a partir de 16 cores el tiempo que tarda en ejecutarse se mantiene constante debido a que el número de tareas concurrentes no es muy grande debido a que es el bucle exterior.

Por otra parte en la versión 5, podemos observar cómo al aumentar los cores, el tiempo disminuye, peró no de forma exponencial como los cores. En el fragmento del excel se puede observar como entre 32 y 128 cores la diferencia es mínima debido al overhead de producir más tareas.

Por último para comprobar lo que nos comentaste en clase de laboratorio, hemos decido comprobar los tiempos que se obtendrían en una versión en la cual el bucle más interno creará las tareas. Como se puede observar en el fragmento de excel, el tiempo en vez de reducir, llega un punto a partir de los 16 cores que es incluso mayor debido a los overheads de tener que crear las tareas.

Cores	Tiempo v4	Tiempo v5	Tiempo bucle interno
1	639780001	639780001	639780001
2	320257001	320682001	320275001
4	191882001	187156001	186328001
8	127992001	120878001	120530001
16	64616001	61501001	64004001
32	64614001	58516001	64004001
128	64614001	56416001	64004001

Fragmento de excel con los tiempos de v4, v5 y versión comentada en clase



4. Understanding the execution of OpenMP programs

En esta sesión de laboratorio, empezaremos a usar la herramienta *Paraver*, la cual nos permite poder visualizar y obtener información de las diferentes ejecuciones que realicemos en el boada.

Durante este apartado nos basaremos en el código del archivo *3dfft_omp.c*, el cual modificaremos a posterior para cumplir con los requisitos de cada apartado en relación con la granularidad y el porcentaje de programa paralelizable.

4.1 Obtaining parallelisation metrics using *modelfactors*

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	1.33	0.78	0.81	1.22	1.47
Speedup	1.00	1.70	1.64	1.09	0.90
Efficiency	1.00	0.42	0.20	0.09	0.06

Table 1: Analysis done on Tue Sep 20 10:29:54 AM CEST 2022, par2118

Overview of the Efficiency metrics in parallel fraction, ϕ=83.16%					
Number of processors	1	4	8	12	16
Global efficiency	98.81%	49.17%	23.87%	9.17%	5.55%
Parallelization strategy efficiency	98.81%	89.15%	87.04%	73.52%	56.33%
Load balancing	100.00%	98.30%	97.97%	98.04%	97.24%
In execution efficiency	98.81%	90.69%	88.84%	74.99%	57.92%
Scalability for computation tasks	100.00%	55.15%	27.43%	12.47%	9.86%
IPC scalability	100.00%	68.92%	51.54%	39.60%	39.29%
Instruction scalability	100.00%	98.32%	96.19%	94.12%	92.18%
Frequency scalability	100.00%	81.40%	55.33%	33.46%	27.23%

Table 2: Analysis done on Tue Sep 20 10:29:54 AM CEST 2022, par2118

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	0.91	0.85	0.87	0.72
LB (time executing explicit tasks)	1.0	0.99	0.98	0.98	0.98
Time per explicit task (average us)	60.82	27.58	27.74	40.68	38.58
Overhead per explicit task (synch %)	0.17	10.64	12.84	32.24	71.83
Overhead per explicit task (sched %)	1.03	1.51	2.02	3.74	5.67
Number of taskwait/taskgroup (total)	1792.0	1792.0	1792.0	1792.0	1792.0

Table 3: Analysis done on Tue Sep 20 10:29:54 AM CEST 2022, par2118

- **Is the scalability appropriate?**

  La escalabilidad de nuestro programa no es adecuada debido a que a medida que aumentamos el número de threads que ejecutan el programa, la eficiencia de cada

uno va disminuyendo. Incluso, como se puede observar en las tablas, el tiempo de ejecución del programa aumenta, con el incremento de threads ejecutándose.

- **Is the overhead due to synchronization neglegible?**
  El overhead debido a la sincronización en nuestro programa, se ve drásticamente decrementado a medida que aumentamos el número de threads que ejecuta nuestro programa. Este impacto se puede ver en la segunda tabla, en la fila de “In execution efficiency”, en la cual se puede observar un decremento del valor de un 98,81% hasta el 57,92 con 16 threads.

- **Is this overhead affecting the execution time per explicit task?**

  El overhead que hay en las tareas explícitas de nuestro programa también se ve afectado por el incremento del número de threads. Pero este cambio es mucho menos proporcionalmente que en el apartado anterior, debido a que en este solo baja un 0,02%.

- **Which is the parallel fraction (φ) for this version of the program?**
  La región paralela de nuestro programa es 83,16%. Este número es debido a que todo nuestro código es paralelizable, menos la primera parte de la ejecución.

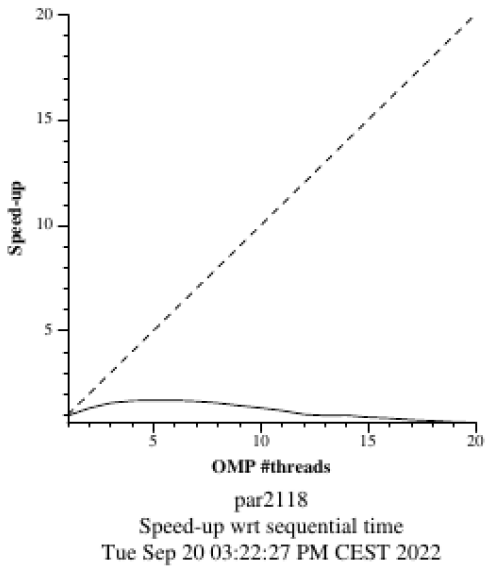
- **Is the efficiency for the parallel regions appropriate?**
  La eficiencia de paralelización de nuestro programa, va disminuyendo de forma progresiva a medida que aumenta el número de threads en ejecución. Como se puede observar en las tablas, nuestra eficiencia varía desde 98,81% con un solo thread, hasta un 56,33% con 16 threads.

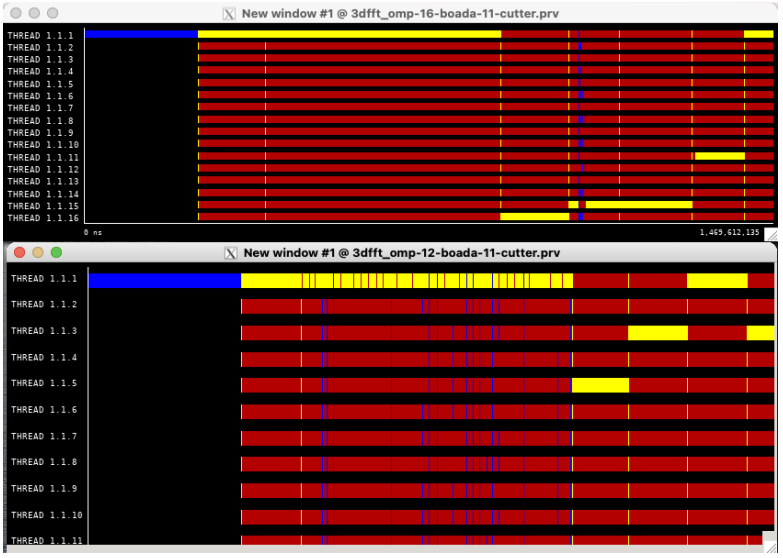
- **Which is the factor that is negatively influencing the most?**

El factor que más está influenciando negativamente a nuestro programa es la escalabilidad para tareas computacionales, debido a que esta cae de un 100% con un solo thread hasta el 9,86% con 16 threads. Por consecuencia estamos perdiendo más de 10 veces la escalabilidad inicial del programa en este tipo de tareas.

4.2 Discovering Paraver (Part I): execution trace analysis

There are two key factors that influence the overall scalability and final performance. Looking at the two timelines windows we can see these two factors: 1) there is one function that is not parallelised and 2) there is a thread state that is predominant along the timeline window. At this moment, the second one seems to more important because there is a strong scalability problem (slowdown from 12 threads) due to the number of tasks and synchronizations in the program. Do you think this overhead problem is constant or function of the number of threads? Why?





Como se puede apreciar en la gráfica superior, extraída con el comando "**sbatch -p execution ./submit-strong-omp.sh**", podemos observar un descenso constante del speedup a partir de la ejecución del programa de con 5 threads. Por tanto, este problema de escalabilidad a partir de los 12 threads es cada vez mayor, hasta llegar al punto mediante el cual con 20 threads el speedup disminuye demasiado. Este descenso debe ser debido a la creación masiva de tareas que hace que tengamos un overhead en nuestro programa mucho superior al tiempo de ejecución del mismo, y si tuvieramos mas threads, este aún sería mayor, debido a que se tendrían que crear aún más tareas.

Y como se puede apreciar en las gráficas extraídas de “**wxparaver**”, el tiempo de ejecución aumenta con 20 threads, por tanto podemos concluir que el overhead nos hace que nuestro programa vaya bastante más lento.

4.3 Discovering Paraver (Part II): understanding the parallel execution

Once you have seen the histograms and timeline windows of the explicit task durations, what kind of explicit task granularity do you think you have: fine or coarse grain?

Tal como se puede ver en el gráfico superior del “**wxparaver**”, se observan multitud de tareas creadas, entre las cuales cabe destacar las verde clarito, las cuales tienen una duración muy pequeña. Por consecuencia, debido al tamaño de las mismas, podemos concluir que nos encontramos ante un “fine granularity program”.

4.4 Reducing Parallelisation Overheads and Analysis

Have we improved the overall performance? Is there any slowdown? Do you observe any major difference on the overheads of the explicit tasks for the initial and optimized versions?

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	1.29	0.56	0.46	0.39	0.38
Speedup	1.00	2.30	2.81	3.27	3.37
Efficiency	1.00	0.57	0.35	0.27	0.21

Table 1: Analysis done on Wed Sep 21 09:21:42 AM CEST 2022, par2118

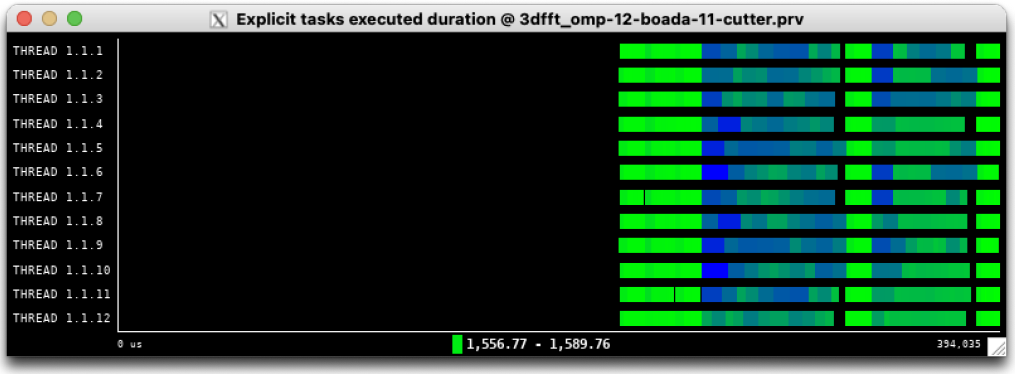
Overview of the Efficiency metrics in parallel fraction, ϕ=82.49%					
Number of processors	1	4	8	12	16
Global efficiency	99.96%	79.40%	60.41%	52.05%	47.00%
Parallelization strategy efficiency	99.96%	96.96%	96.61%	94.99%	96.76%
Load balancing	100.00%	97.59%	97.22%	95.97%	97.78%
In execution efficiency	99.96%	99.35%	99.37%	98.97%	98.96%
Scalability for computation tasks	100.00%	81.90%	62.53%	54.79%	48.57%
IPC scalability	100.00%	83.13%	66.36%	60.20%	53.57%
Instruction scalability	100.00%	99.99%	99.98%	99.97%	99.96%
Frequency scalability	100.00%	98.52%	94.24%	91.05%	90.71%

Table 2: Analysis done on Wed Sep 21 09:21:42 AM CEST 2022, par2118

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	0.96	0.93	0.8
LB (time executing explicit tasks)	1.0	0.99	0.98	0.97	0.98
Time per explicit task (average us)	15178.0	4633.12	3033.96	2308.04	1952.59
Overhead per explicit task (synch %)	0.0	3.07	3.38	5.1	3.12
Overhead per explicit task (sched %)	0.03	0.03	0.04	0.04	0.05
Number of taskwait/taskgroup (total)	7.0	7.0	7.0	7.0	7.0

Table 3: Analysis done on Wed Sep 21 09:21:42 AM CEST 2022, par2118

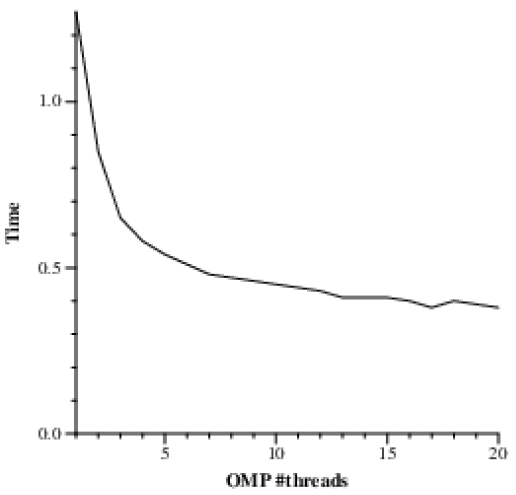
Como se puede observar en las tablas anteriores obtenidas del modelfactors, se puede observar un aumento considerable del rendimiento al aumentar el número de núcleos que usamos durante la ejecución de nuestro programa. Este descenso es debido a que al no tener tantas tareas, el overhead de creación/sincronización es menor y por tanto se reduce el tiempo de ejecución total del programa.

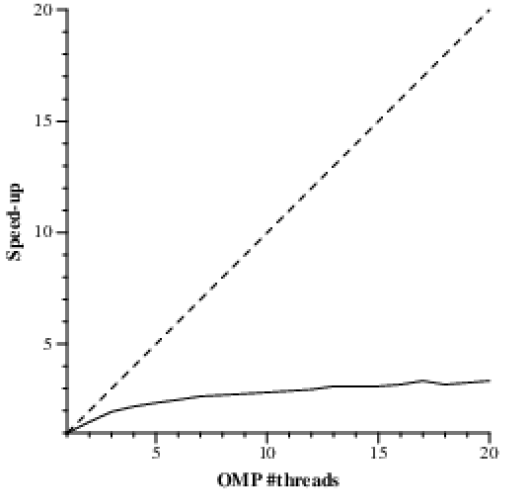


Como se puede observar en el diagrama anterior, el tiempo durante el cual una tarea se encuentra en ejecución es mucho mayor que en el apartado anterior, el cual era entre 8 y 14 ns. En este caso las tareas son 200 veces más grandes que en el caso anterior, por tanto la proporción de los overheads respecto al tiempo de ejecución es menor.

On the other hand, why do you think you can not achieve better speedup for 12 or more threads?

No podemos conseguir un mayor speedup a partir de los 12 threads, debido a que al no crear tantas tareas y estas ser de un tamaño mayor, hay momento de la ejecución en los cuales el programa se tiene que esperar a que acaben otras regiones del código y por tanto debido a las dependencias este no puede ser mayor.





En las gráficas superiores, podemos observar como el speedup a partir de los 12 threads no aumenta y como el tiempo mínimo de ejecución ya no disminuye tan rápidamente a partir de los 12 threads.

Do you observe any major difference on the duration of the implicit and explicit tasks for the initial and optimized versions? What is the function that is limiting the maximum speedup that you can achieve? Which functions in the program are or not parallelized? Which is the duration of the sequential execution for those user function not parallelised?

Como hemos comentado anteriormente, en la segunda versión del programa podemos observar como el tiempo de ejecución de las tareas ejecutadas por el programa se ha visto incrementado 200 veces.

La función que nos está limitando el speedup de nuestro programa és la ejecución del main de nuestro programa "*3dfft_omp.c*", el cual no está del todo paralelizado.

La función que no está nada paralelizada és "*init_complex_grid*", la cual es la primera en ejecutarse en nuestro programa. Teniendo en cuenta que nuestro programa con 12 threads

tarda 394034519 ns y que la región paralelizable en un 82,48%, nos da un tiempo na paralelizable de 69034847 ns, lo cual equivale a un 17,52% del tiempo de ejecución.

4.5 Improving ϕ and Analysis

Have the speed–up and efficiency metrics improved? What is the new value for ϕ? Compare the three version: initial, reducing overhead, and improving ϕ under the point of view of strong scalability.

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	1.30	0.39	0.25	0.19	0.15
Speedup	1.00	3.34	5.28	7.02	8.46
Efficiency	1.00	0.84	0.66	0.58	0.53

Table 1: Analysis done on Wed Sep 21 10:02:38 AM CEST 2022, par2118

Overview of the Efficiency metrics in parallel fraction, ϕ=99.95%					
Number of processors	1	4	8	12	16
Global efficiency	99.85%	83.46%	66.02%	58.51%	52.94%
Parallelization strategy efficiency	99.85%	96.78%	95.04%	93.21%	93.93%
Load balancing	100.00%	97.98%	97.56%	95.91%	97.75%
In execution efficiency	99.85%	98.77%	97.42%	97.18%	96.09%
Scalability for computation tasks	100.00%	86.23%	69.46%	62.78%	56.36%
IPC scalability	100.00%	87.23%	75.29%	70.66%	63.95%
Instruction scalability	100.00%	99.80%	99.54%	99.28%	99.03%
Frequency scalability	100.00%	99.06%	92.69%	89.48%	89.00%

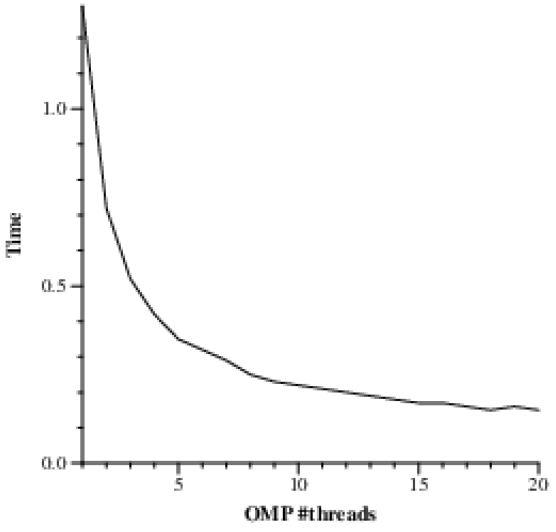
Table 2: Analysis done on Wed Sep 21 10:02:38 AM CEST 2022, par2118

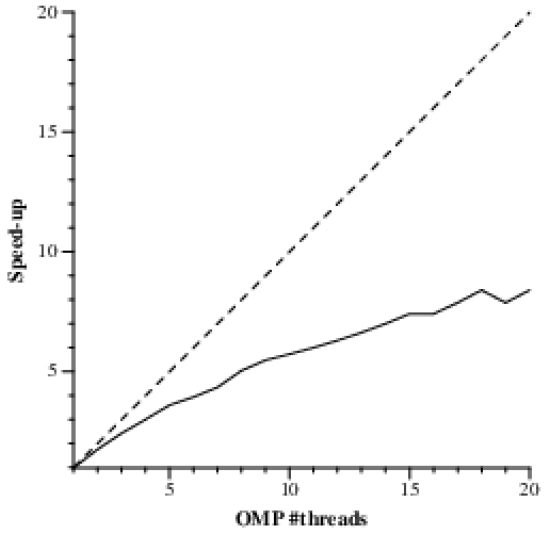
Statistics about explicit tasks in parallel fraction					
Number of processors	1	4	8	12	16
Number of explicit tasks executed (total)	2640.0	10560.0	21120.0	31680.0	42240.0
LB (number of explicit tasks executed)	1.0	0.96	0.96	0.96	0.93
LB (time executing explicit tasks)	1.0	0.98	0.98	0.97	0.98
Time per explicit task (average us)	490.71	142.31	88.46	65.3	54.69
Overhead per explicit task (synch %)	0.02	3.13	4.83	6.77	5.7
Overhead per explicit task (sched %)	0.13	0.17	0.34	0.44	0.67
Number of taskwait/taskgroup (total)	264.0	264.0	264.0	264.0	264.0

Table 3: Analysis done on Wed Sep 21 10:02:38 AM CEST 2022, par2118

Como se puede observar en las tablas superiores, el speedup de nuestro programa se ha visto incrementado de forma notable, gracias a la alta paralelización de nuestro programa.

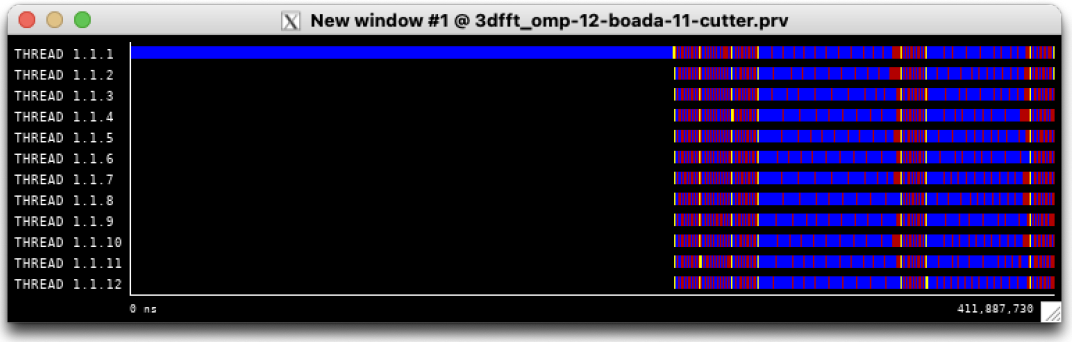
El nuevo valor de nuestro programa paralelizado es 99,95%.

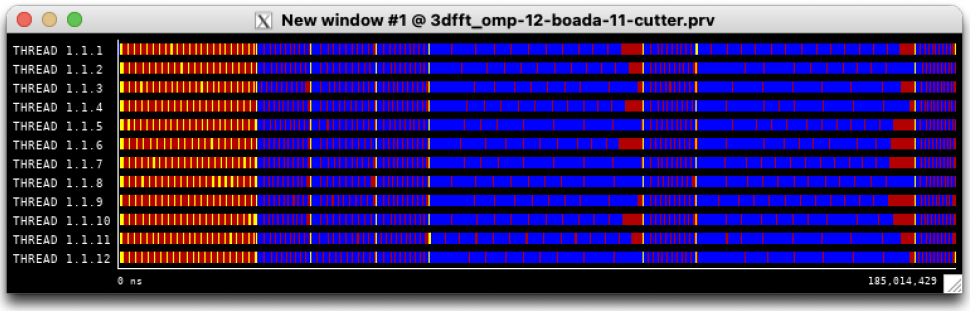




Como se puede observar en las gráficas de escalabilidad fuerte, se puede apreciar un aumento considerable del speedup a medida que aumentan los threads del programa y también podemos observar una disminución continuada del tiempo de ejecución a medida que se aumentan los threads.

Compare the three version: initial, reducing overhead, and improving ϕ using the timeline window and profile of thread states.





Como se puede observar en las ventanas temporales de las tres versiones del programa, se puede observar como entre la primera y la segunda hay un gran avance respecto al tiempo total de ejecución y esto es debido a la disminución considerable del número de tareas. Por otra parte, entre la segunda y la tercera versión, esta mejora se ve reflejada principalmente en el porcentaje de programa paralelizable, el cual se ve incrementado hasta prácticamente el 100% del tiempo total.

Para apreciar mejor todas las mejoras, hemos hecho rellenado la tabla con los valores de las diferentes versiones de nuestro programa.

Versión	ϕ	**Ideal S12**	**T1**	**T12**	**real S12**
initial version 3dfft_omp.c	83,16	5,9382	1,33	1,47	1,09
new version reducing overheads	82,48	5,7077	1,29	0,39	3,27
final version improved ϕ	99,95	2000	1,30	0,19	7,02

5. Conclusiones

Durante este primer laboratorio hemos aprendido en primer lugar sobre la arquitectura de los nodos sobre los que estamos trabajando, lo cual nos puede ser útil para futuros laboratorios. También hemos aprendido como el lugar donde se colocan las tareas en el código, influye en el rendimiento del programa, debido a que o puede que creemos demasiadas pocas tareas pal nombre de threads que tenemos o que creemos demasiadas tareas y el overhead sea mayor al tiempo de ejecución del mismo. Y por último hemos aprendido a usar las diferentes herramientas que nos proporcione el boada, entre ellas cabe destacar el modelfactor, paraver y tareador, las cuales nos han sido de gran utilidad durante la realización del laboratorio.