

Informe

Laboratorio 1 de PAR

Fall 2022-2023



Paula Barrachina Cáceres (*par4109*)
Marc Castro Chavez (*par4111*)

Índice

1. Experimental setup	2
Procedimiento:	2
1.1 Node architecture and memory	2
1.4. Compilation and execution of OpenMP programs	3
1.5. Strong vs. weak scalability	4
Strong Scalability	4
Weak Scalability	5
2. Systematically analysing task decompositions with Tareador	6
Procedimiento:	6
2.3 Exploring new task decompositions for 3DFFT	6
Versión 1	6
Versión 2	7
Versión 3	8
Versión 4	8
Versión 5	9
3. Understanding the execution of OpenMP programs	11
Procedimiento:	11
4.1 Discovering model factors: Overall analysis	11
4.2 Discovering Paraver (Part I): execution trace analysis	12
4.3 Discovering Paraver (Part II): understanding the parallel execution	12

1. Experimental setup

Data: 06/09/2022

En esta primera sesión el objetivo principal era entender cómo está estructurado boada y familiarizarnos con el entorno de este.

Procedimiento:

1.1 Node architecture and memory

Para investigar la arquitectura de los nodos de **boada-11** a **boada-14** hemos ejecutado los comandos *lscpu* i *lstopo*. Con los datos proporcionados por los comandos hemos llenado la tabla, algunos datos no eran explícitos pero se podían intuir a partir de otros.

	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	3200 MHz
L1-I cache size (per-core)	640 KB
L1-D cache size (per-core)	640 KB
L2 cache size (per-core)	20 MB
Last-level cache size (per-socket)	27.5 MB
Main memory size (per socket)	47 GB
Main memory size (per node)	94 GB

Tabla 1: Tabla descriptiva de la arquitectura de los servidores de **boada-11** a **boada-14**

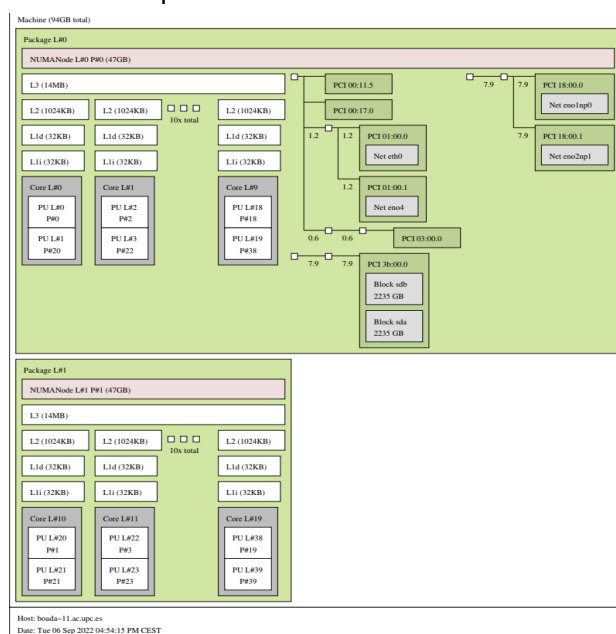


Figura 1: Diagrama de la arquitectura de **boada-11**

1.4. Compilation and execution of OpenMP programs

Para familiarizarnos con la ejecución y compilación en boada hemos estudiado el programa **pi_omp.c**. Del cual hemos obtenido los siguientes datos:

# threads	Interactive: Timing information				Queued: Timing information			
	user	system	elapsed	% of CPU	user	system	elapsed	% of CPU
1	2'36	0.00	0:02.37	99%	0.68	0.00	0:00.70	98%
4	2'39	0.03	0:01.22	199%	0.70	0.00	0:00.19	360%
8	2'39	0.04	0:01.22	199%	0.74	0.01	0:00.11	658%
16	2'42	0.11	0:01.29	199%	0.79	0.00	0:00.09	1111%
20	2'47	0.10	0:01.29	199%	0.84	0.00	0:00.08	1013%

Tabla 2: Resultados de la ejecución del programa **pi_omp.c** en interactivo y en queue.

El resultado del script nos indicaba el usuario, el system time, el elapsed time y el porcentaje de CPU usado por el programa.

Si nos fijamos en la tabla 2, hecha con los datos obtenidos, podemos observar que en la ejecución iterativa por más que aumentemos el número de threads no mejora el tiempo. Sin embargo, si nos fijamos en la columna de los porcentajes de CPU usados vemos que en un momento pasa de 99% a 199%, eso significa que se utilizan los dos threads de cada core.

El porcentaje de CPU utilizado no puede incrementar por encima de este límite ya que para hacerlo serían necesarios más cores y al ser un programa no paralelizable en múltiples cores, no es posible que su ejecución sea más rápida de lo que es.

En cuanto a la ejecución en cola, vemos que al encolar las tareas a medida que aumentamos el número de threads el tiempo de ejecución se ve reducido.

Además el porcentaje de CPU utilizado también aumenta hasta superar de largo el límite de 200% que limita la versión de un solo core. Esto es gracias al hecho de que al encolar la ejecución permite planificar un cálculo en paralelo del programa y podemos superar el límite.

1.5. Strong vs. weak scalability

Strong Scalability

Consiste en que el número de threads se aumenta pero dejamos la medida del problema fija reduciendo el trabajo que ha de hacer cada procesador. Esto significa que podemos llegar a un punto donde por más que añadamos más procesadores no cambiará el tiempo de ejecución en casi nada o nada, ya que habremos llegado a la máxima paralelización debido a la dominación de overheads de la creación y sincronización de las tareas.

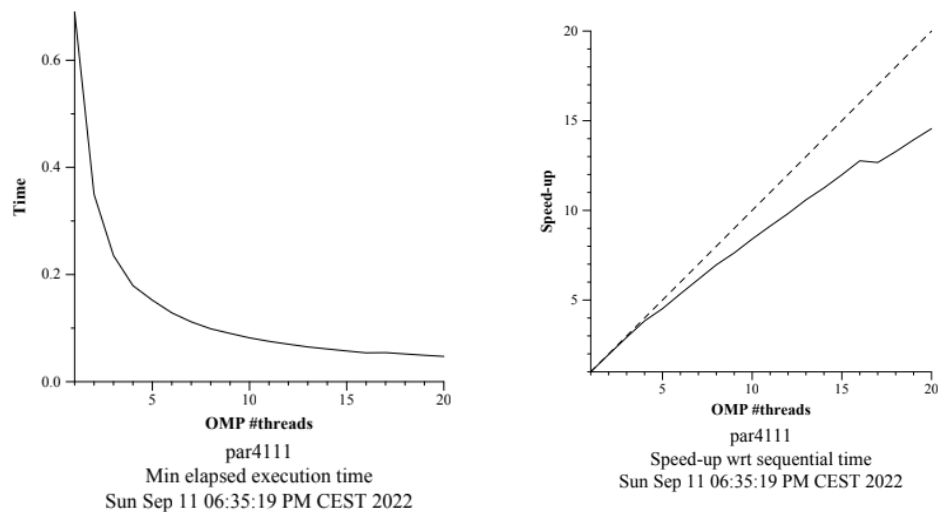


Figura 2: Diagrama de Strong Scalability 20 threads.

En los gráficos se puede ver como con un problema con 1.000.000.000 iteraciones a medida que se aumentan los threads el tiempo se reduce logarítmicamente y en la gráfica del speed-up se puede apreciar que del 15 al 20 la línea empieza a ser más horizontal.

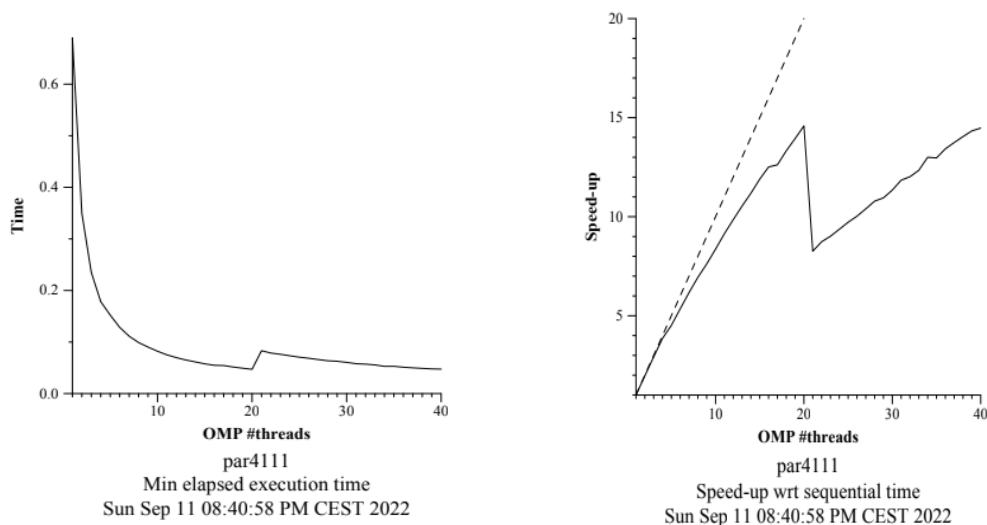


Figura 3: Diagrama de Strong Scalability 40 threads.

Al aumentar el número de threads hasta 40 vemos que hay un comportamiento extraño, eso es debido a que con 12 13 14 threads el coste de hacer **hyper-threading** es más alto que el beneficio de este. Pero si seguimos añadiendo cores llega un punto donde volvemos a ver la reducción del tiempo.

Weak Scalability

Se hace lo contrario a la Strong Scalability. Ganamos paralelización según vamos aumentando la medida del problema de manera proporcional al número de threads. De esta manera deberíamos ver un speed-up más o menos constante según incrementa la carga de trabajo que estamos realizando.

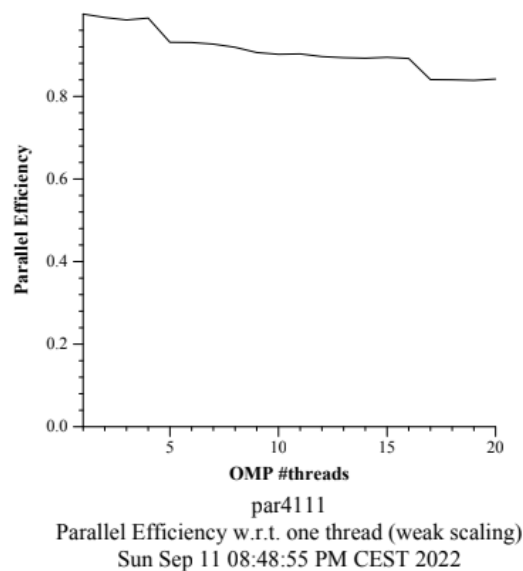


Figura 4: Diagrama de Strong Scalability 20 threads.

Como podemos observar en el gráfico (figura 4) una vez llegamos a los 5 threads decremента de una manera significativa y continua decremētando poco a poco hasta llegar a los 15 threads donde vuelve a bajar de manera considerable. Es decir tiene una tendencia a ir bajando cada vez más según aumentamos el número de los threads.

2. Systematically analysing task decompositions with Tareador

Data: 13/09/2022

En esta sesión el objetivo principal era familiarizarse con el Tareador y entender cómo ir aumentando la granularidad del código **3dfft** dividiendo las tareas.

Procedimiento:

2.3 Exploring new task decompositions for 3DFFT

A partir del código base facilitado en esta sesión, se nos pide programar un seguido de modificaciones para entender el efecto de la paralelización de las tareas de manera gráfica e interactiva.

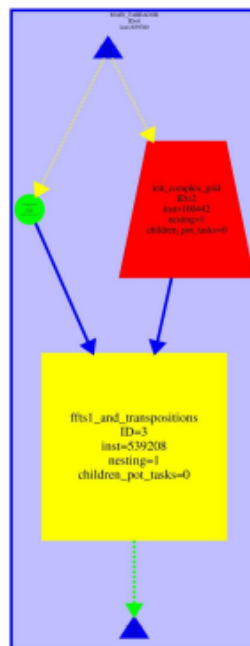


Figura 5: Representación con el Tareador de la distribución de tareas inicial.

Se nos pide realizar cinco versiones del código:

Versión 1

La primera modificación nos pide granular la tarea **ffts1_and_transpositions** para que no sea una única tarea de 539208 instrucciones, si no varias de menos instrucciones. Esta modificación, en cuanto a tiempo, no significa nada ya que no se paraleliza solo se divide para que la tarea no sea de un tamaño tan grande, de ahí que los datos sean igual que de manera secuencial, véase en la tabla 5.



Figura 6: Representación con el Treadador de la distribución de la versión 1.

Versión 2

En esta segunda modificación partimos de la versión anterior y en este caso modificamos la función **ffts1_planes** para que se ejecuten paralelamente 10 tareas. En esta versión sí que se nota un cambio en cuanto al tiempo debido a la paralelización, véase en la tabla 5.

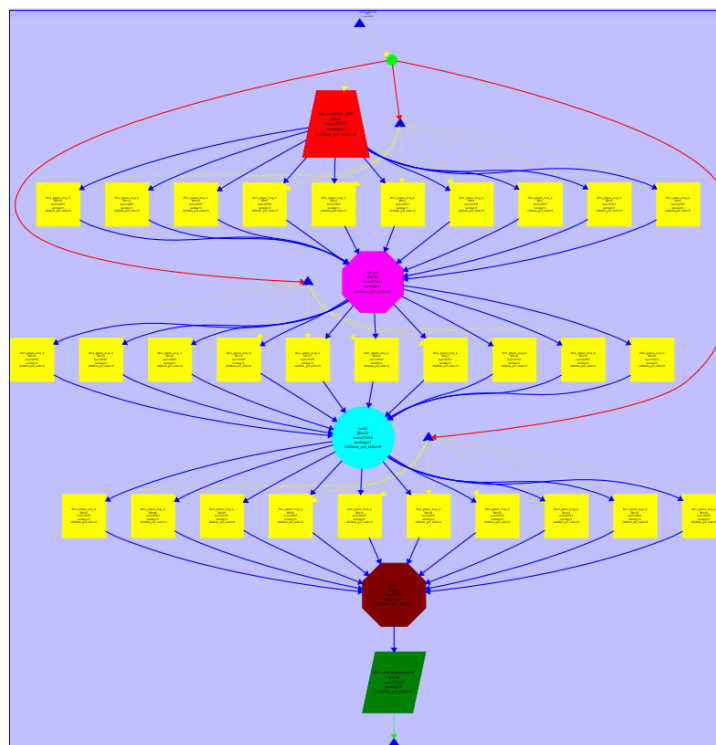


Figura 7: Representación con el Treadador de la distribución de la versión 2.

Versión 3

Siguiendo con las modificaciones anteriores en este apartado hemos sustituido las tareas realizadas en las funciones ***transpose_xy_planes*** y ***transpose_zxx_planes*** por una granularidad más fina. Con esto hemos conseguido que las 4 tareas más costosas se hayan convertido en 10 tareas paralelizadas. De esta manera conseguimos una reducción del tiempo aún mayor, véase en la tabla 5.

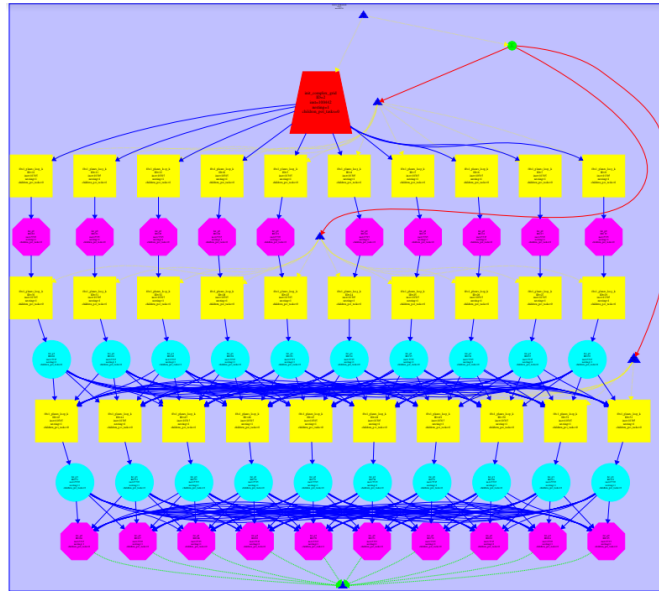


Figura 8: Representación con el Tareador de la distribución de la versión 3.

Versión 4

En esta cuarta modificación sustituimos la última función restante en el main ***init_complex_grid***, de manera que ahora tiene un tratamiento más granular. Hasta ahora se había ejecutado como una única función y al ser paralelizada vemos que el tiempo se reduce de una manera considerable.

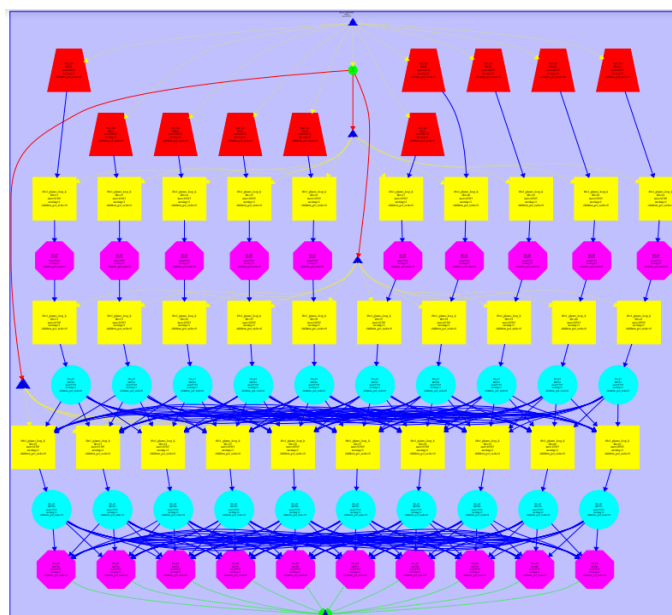


Figura 9: Representación con el Tareador de la distribución de la versión 4.

Número de cores	Time (ns)
1	639 780 001
2	320 310 001
4	165 389 001
8	91 496 001
16	64 018 001
32	64 018 00

Tabla 3: Tabla escalabilidad de la versión 4.

Versión 5

En esta última modificación partimos de la versión anterior y en este caso modificamos la función ***ffts1_planes***.

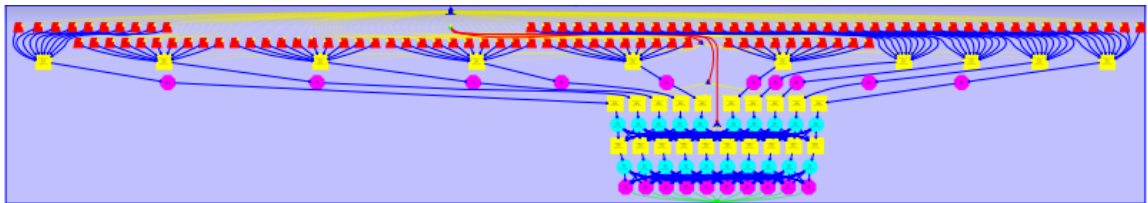


Figura 10: Representación con el Tareador de la distribución de la versión 5.

Número de cores	Time (ns)
1	639 780 001
2	320 081 001
4	165 721 001
8	094 020 001
16	060 913 001
32	057 928 001
128	055 820 001

Tabla 4: Tabla escalabilidad de la versión 5.

```

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

    for (k = 0; k < N; k++) {
        //tareador_start_task("init_fat");
        for (j = 0; j < N; j++) {
            tareador_start_task("init_fat");
            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_fat");
    }
}

```

Figura 11: Fragmento de código de la versión 5.

Para conseguir una granularidad más fina hemos cambiado la función ***init_complex_grid***, haciendo que las tareas se generen en el segundo bucle en vez del primero. Aun así, hay formas de reducirla más, como generando la tarea en el bucle más interno de la función o incluso aún más si cambiamos también las otras dos funciones ***transpose_xy_planes*** y ***transpose_zx_planes*** poniendo el generador de tareas en el segundo o tercer bucle.

Hemos decidido aplicar la primera opción ya que solo se nos pedía una granularidad más fina a la anterior y el resultado es mucho más comprensible que las otras posibilidades ya que estas generaban una representación con el Tareador de dimensiones excesivamente grandes y se dificulta la comprensión.

Con todas las modificaciones hechas y a partir de los datos que nos ha aportado el Tareador hemos rellenado la siguiente tabla:

Version	T_1	T_∞	Parallelism
seq	0.639 s	0.639 s	1
v1	0.639 s	0.639 s	1
v2	0.639 s	0.361 s	1.7
v3	0.639 s	0.154 s	4.14
v4	0.639 s	0.064 s	9.98
v5	0.639 s	0.055 s	11.61

Tabla 5: Tabla de los tiempos con los datos del Tareador.

En ella se puede apreciar la mejora de tiempo gracias a la paralelización.

3. Understanding the execution of OpenMP programs

Data: 13/09/2022

En esta última sesión de la práctica hemos acabado de profundizar en la paralelización del programa **3dfft_omp.c**.

Procedimiento:

4.1 Discovering model factors: Overall analysis

En la versión inicial del código **3dfft_omp.c** la granularidad al empezar es la misma que la de la versión 4 de la sesión anterior. En la primera ejecución sin modificar el programa, obtenemos las siguientes tablas.

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	1.34	0.78	0.80	1.29	1.49
Speedup	1.00	1.71	1.68	1.04	0.90
Efficiency	1.00	0.43	0.21	0.09	0.06

Table 1: Analysis done on Tue Sep 20 04:27:54 PM CEST 2022, par4111

Overview of the Efficiency metrics in parallel fraction, $\phi=83.00\%$					
Number of processors	1	4	8	12	16
Global efficiency	98.75%	49.32%	24.22%	8.74%	5.50%
Parallelization strategy efficiency	98.75%	89.23%	86.78%	70.85%	55.89%
Load balancing	100.00%	97.69%	97.74%	97.34%	97.49%
In execution efficiency	98.75%	91.34%	88.78%	72.79%	57.33%
Scalability for computation tasks	100.00%	55.28%	27.91%	12.34%	9.84%
IPC scalability	100.00%	69.54%	51.64%	38.31%	40.28%
Instruction scalability	100.00%	98.32%	96.18%	94.14%	92.16%
Frequency scalability	100.00%	80.84%	56.20%	34.21%	26.50%

Table 2: Analysis done on Tue Sep 20 04:27:54 PM CEST 2022, par4111

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	0.85	0.75	0.72	0.8
LB (time executing explicit tasks)	1.0	0.98	0.98	0.98	0.98
Time per explicit task (average us)	61.04	27.62	27.35	41.26	38.82
Overhead per explicit task (synch %)	0.16	10.56	13.13	37.07	73.18
Overhead per explicit task (sched %)	1.1	1.49	2.07	4.04	5.72
Number of taskwait/taskgroup (total)	1792.0	1792.0	1792.0	1792.0	1792.0

Table 3: Analysis done on Tue Sep 20 04:27:54 PM CEST 2022, par4111

Tabla 6: Tablas de la versión inicial del código **3dfft_omp.c**.

De las tres tablas obtenemos diferente información. A partir de estos datos vemos que la escalabilidad no es la adecuada y va bajando a medida que añadimos procesadores al igual que la eficiencia y vemos que su fracción paralela es de un 83%.

Si miramos el tiempo de ejecución en la primera tabla vemos que no mejora, si no que, empeora al pasar de 1 a 16 procesadores esto nos indica que el tiempo de overhead no es negligible y que nos afecta gravemente como se puede ver en los datos de la tercera tabla.

4.2 Discovering Paraver (Part I): execution trace analysis

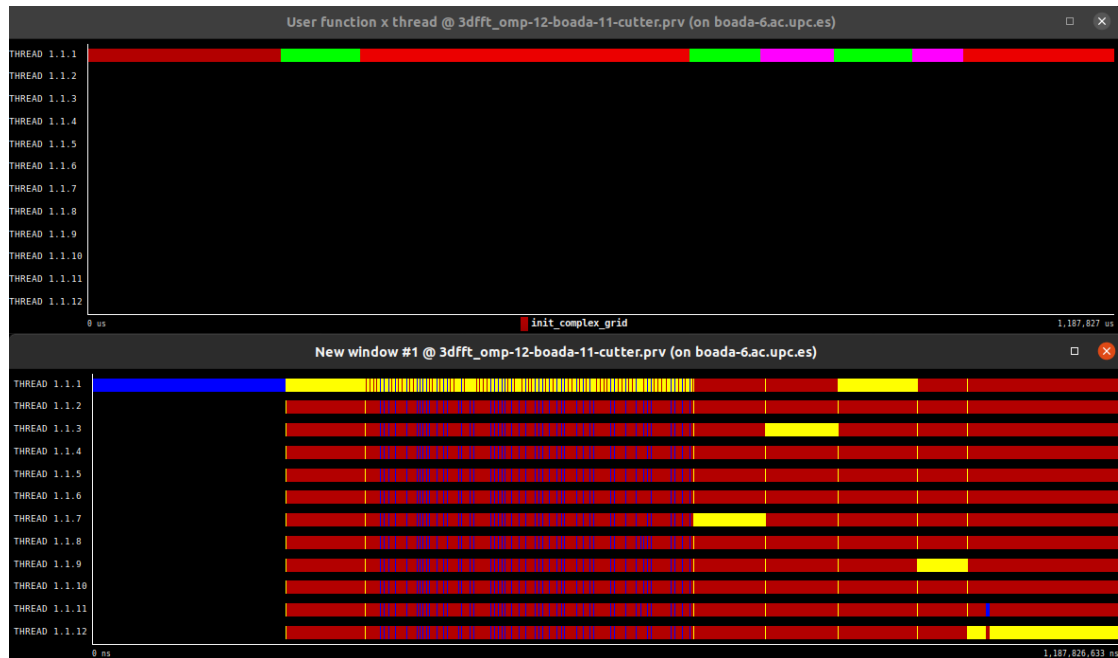


Figura 12: Tabla user functions en comparativa a la primera version del programa.

Mirando las tablas vemos que hay una función que no está paralelizada (*init_complex*) además vemos que hay un thread predominante, el número 1, y esto es debido al problema de escalabilidad mencionado anteriormente y el número de tareas.

Gracias a las tablas anteriores, concretamente a la tercera, vemos que el aumento del overhead va ligado al aumento del número de threads.

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

2D thread state profile @ 3dfft_omp-12-boada-11-cutter.prv (on boada-7.ac.upc.es)			
	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	69.98 %	10.46 %	19.56 %
THREAD 1.1.2	68.89 %	31.10 %	0.00 %
THREAD 1.1.3	-	-	-
THREAD 1.1.4	-	-	-
THREAD 1.1.5	-	-	-
THREAD 1.1.6	-	-	-
THREAD 1.1.7	-	-	-
THREAD 1.1.8	-	-	-
THREAD 1.1.9	-	-	-
THREAD 1.1.10	-	-	-
THREAD 1.1.11	-	-	-
THREAD 1.1.12	-	-	-
Total	138.87 %	41.57 %	19.56 %
Average	69.44 %	20.78 %	9.78 %
Maximum	69.98 %	31.10 %	19.56 %
Minimum	68.89 %	10.46 %	0.00 %

Figura 13: Histograma primera versión del código.

Como se puede observar en la figura en los histogramas vemos que la granularidad de nuestro programa es de *coarse grain*.

Para la segunda versión hemos añadido más granularidad al modificar el código moviendo la instrucción **#pragma omp taskloop** del bucle interno a antes de empezar el bucle para incluir el bucle externo en esta. Así conseguimos aumentar la granularidad y reducir el overhead causado por el paralelismo.

Y al hacer esta ejecución obtenemos la siguientes tablas:

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	1.28	0.56	0.46	0.41	0.37
Speedup	1.00	2.30	2.81	3.13	3.44
Efficiency	1.00	0.57	0.35	0.26	0.21

Table 1: Analysis done on Tue Sep 20 05:11:34 PM CEST 2022, par4111

Overview of the Efficiency metrics in parallel fraction, $\phi=82.36\%$					
Number of processors	1	4	8	12	16
Global efficiency	99.96%	78.03%	60.13%	52.23%	46.46%
Parallelization strategy efficiency	99.96%	96.76%	97.08%	95.95%	97.43%
Load balancing	100.00%	97.10%	97.71%	96.80%	98.53%
In execution efficiency	99.96%	99.64%	99.36%	99.12%	98.89%
Scalability for computation tasks	100.00%	80.65%	61.94%	54.43%	47.69%
IPC scalability	100.00%	81.65%	65.95%	59.85%	52.48%
Instruction scalability	100.00%	99.99%	99.98%	99.97%	99.96%
Frequency scalability	100.00%	98.78%	93.93%	90.99%	90.91%

Table 2: Analysis done on Tue Sep 20 05:11:34 PM CEST 2022, par4111

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	0.93	0.9	0.92	0.8
LB (time executing explicit tasks)	1.0	0.97	0.98	0.97	0.99
Time per explicit task (average us)	15044.32	4663.41	3035.92	2302.77	1971.39
Overhead per explicit task (synch %)	0.0	3.29	2.88	4.05	2.42
Overhead per explicit task (sched %)	0.03	0.02	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 Tue Sep 20 05:11:34 PM CEST 2022, par4111

Tabla 7: Tablas de la versión inicial del código **3dfft_omp.c**.

En ellas podemos ver que tanto el tiempo de ejecución como la eficiencia han mejorado respecto a la primera versión, aun así, aunque el overhead se ve reducido de manera importante aún se podría conseguir un mayor speedup ya que falta por paralelizar una función, por eso el speedup en 12 procesadores no es lo bueno que podría ser .

Además en esta versión del programa la fracción paralela se ve reducida a 82% casi nada teniendo en cuenta la mejora que hay respecto al primer programa.

Para esta última versión con tal de reducir el tiempo de overhead, hemos añadido en el bucle interno de la función ***init_complex*** la instrucción ***#pragma omp taskloop*** para así conseguir una mayor mayor eficiencia y acabar de paralelizar la parte que faltaba del programa.

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	1.29	0.41	0.25	0.24	0.28
Speedup	1.00	3.16	5.20	5.36	4.58
Efficiency	1.00	0.79	0.65	0.45	0.29

Table 1: Analysis done on Tue Sep 20 05:42:31 PM CEST 2022, par4111

Overview of the Efficiency metrics in parallel fraction, $\phi=99.97\%$					
Number of processors	1	4	8	12	16
Global efficiency	99.85%	78.83%	65.00%	44.64%	28.65%
Parallelization strategy efficiency	99.85%	94.97%	92.55%	70.99%	50.80%
Load balancing	100.00%	97.66%	97.22%	95.02%	95.36%
In execution efficiency	99.85%	97.25%	95.20%	74.71%	53.27%
Scalability for computation tasks	100.00%	83.01%	70.23%	62.88%	56.40%
IPC scalability	100.00%	85.48%	76.80%	70.75%	63.58%
Instruction scalability	100.00%	99.80%	99.54%	99.29%	99.03%
Frequency scalability	100.00%	97.30%	91.87%	89.52%	89.57%

Table 2: Analysis done on Tue Sep 20 05:42:31 PM CEST 2022, par4111

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	0.94	0.96	0.88	0.84
LB (time executing explicit tasks)	1.0	0.99	0.98	0.96	0.96
Time per explicit task (average us)	489.69	147.48	87.15	64.89	54.25
Overhead per explicit task (synch %)	0.01	4.94	7.26	36.54	90.12
Overhead per explicit task (sched %)	0.13	0.34	0.75	4.28	6.71
Number of taskwait/taskgroup (total)	263.0	263.0	263.0	263.0	263.0

Table 3: Analysis done on Tue Sep 20 05:42:31 PM CEST 2022, par4111

Tabla 8: Tablas de la versión inicial del código ***3dfft_omp.c***.

En esta última versión vemos una mejora en cuanto a tiempo de ejecución y como la fracción paralela está a un 99'97% casi al 100% indicando que ya no hay funciones sin paralelizar, si no, una parte muy escasa del código. Sin embargo vemos como la eficiencia se ve reducida y el overhead nos impide un speedup mayor.

Una vez hemos hecho las tres versiones del programa hemos completado la siguiente tabla con los datos obtenidos de las diferentes versiones de la sesión y así poder hacer una comparativa general de los tres códigos.


Version		Ideal S ₁₂	T ₁	T ₁₂	real S ₁₂
initial version	0.83	4.18	1.335	1.28	1.04
new version	0.82	4.02	1.279	0.408	3.13
final version	0.99	10.81	1.290	0.240	5.38

Tabla 9: Tabla resumen de las mejoras del código **3dfft_omp.c**.

Las fórmulas que hemos usado para completar la tabla son:

$$T_p = T_{seq} + T_{par}$$

$$S_{12 \text{ Ideal}} = T_1/T_{12}$$

$$S_{12 \text{ real}} = 1/(1-\phi + (\phi/12))$$



Figura 14: Comparativa de las tres versiones del programa con 12 threads de misma escala temporal.

Como se puede apreciar en la comparativa global, tanto en la tabla 9 como en la figura 14, la evolución del código es considerable en cuanto a tiempo. En cuanto al speedup también hay un cambio significativo aunque mejora respecto al inicio y se consigue un mejor tiempo gracias a ello, aun así solo se llega a la mitad del speedup ideal debido al overhead del programa y no es posible aumentarlo más ya que en nuestra última versión el porcentaje de paralelización es del 99,97% indicando que no hay más fragmento de código paralelizar.

Esto nos demuestra que por mucho que paralelizemos nuestro código teniendo en cuenta el speedup ideal, este se va a ver afectado de manera importante por el overhead del programa. Además a medida que vamos paralelizando y aumentando los threads la eficiencia va disminuyendo de manera considerable.

En definitiva la paralelización es útil para reducir el tiempo de ejecución y mejorar la eficiencia del programa pero llega un punto donde por más granularidad que introduzcas al programa la mejora podría no compensar los problemas causados por dicha paralelización. Es decir llega un momento donde paralelizar más no sale a cuenta.