

# Entregable de Laboratori 1: Experimental setup and tools

GUILLEM GRÀCIA ANDREU, GERARD MADRID MIRÓ  
TARDOR 2021-22  
PAR3109

## Sessió 1:

### 1.1 Node architecture and memory

En aquest apartat es demana investigar i resoldre diferents dades de l'arquitectura dels nodes de Boada. Mitjançant l'ús de les comandes `lscpu` i `lstopo`, hem aconseguit obtenir un bolcat d'informació (Figura 1) sobre el node en que estavem treballant (en aquest cas boada-1):

	boada-1 to boada-4
<b>Number of sockets per node</b>	2
<b>Number of cores per socket</b>	6
<b>Number of threads per core</b>	2
<b>Maximum core frequency</b>	2395 MHz
<b>L1-I cache size (per core)</b>	32 kB
<b>L1-D cache size (per core)</b>	32 kB
<b>L2 cache size (per core)</b>	256 kB
<b>Last-level cache size (per socket)</b>	12288 kB
<b>Main memory size (per socket)</b>	12 GB
<b>Main memory size (per node)</b>	23 GB

Fig. 1: Taula de dades dels primers quatre nodes del servidor boada

Com podem observar, el procesador (Intel Xeon E5645) amb el que treballarem pot tenir un total de:

$$\text{Total Threads} = 2 \frac{\text{sockets}}{\text{node}} \cdot 6 \frac{\text{cores}}{\text{socket}} \cdot 2 \frac{\text{threads}}{\text{core}} = 24 \frac{\text{threads}}{\text{node}}$$

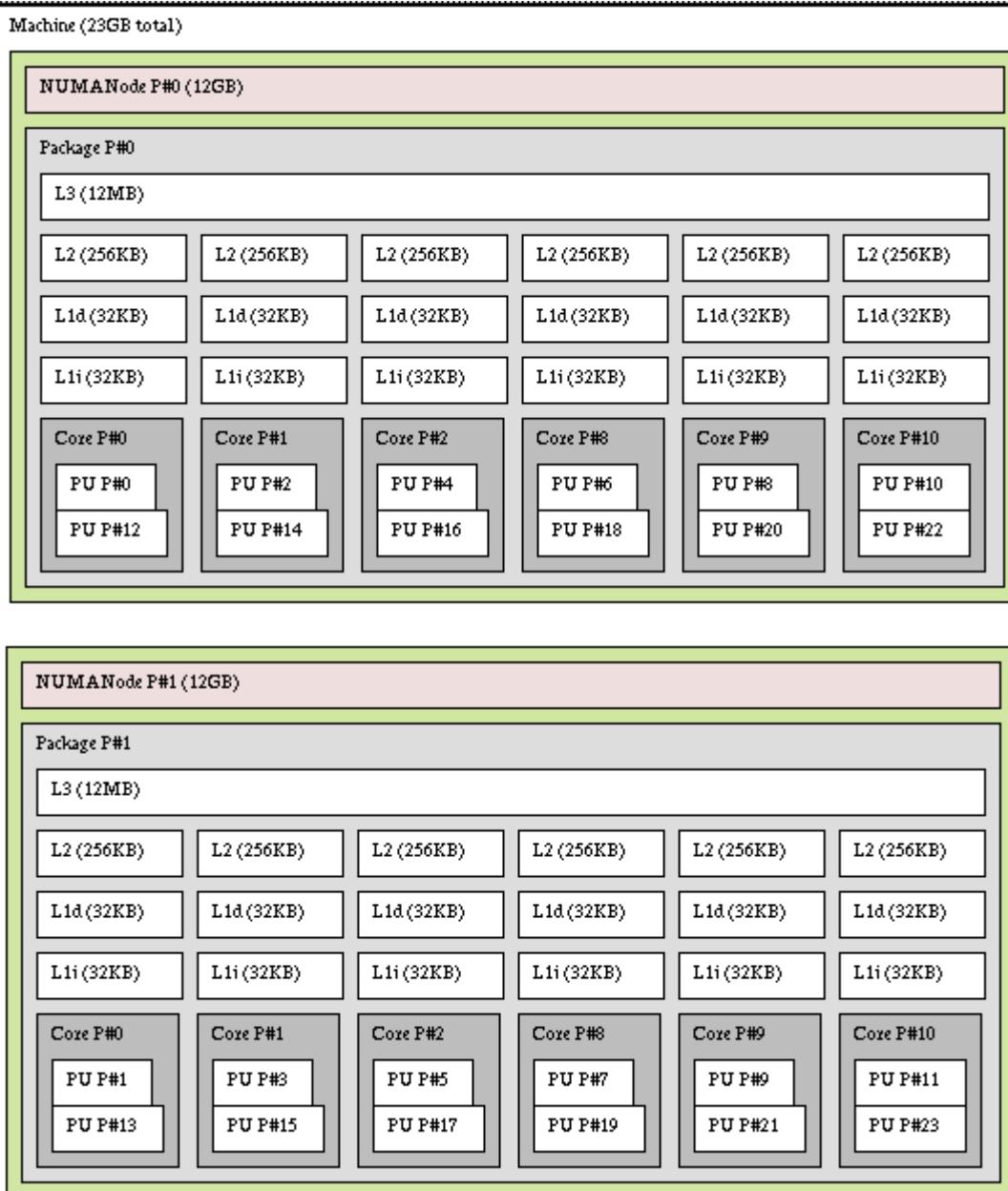
i pot treballar a un màxim de 2.4 GHz.

Com podem comprovar, els 4 primers nodes tenen la mateixa arquitectura, per tant, quan executem els programes, és igual a quin dels nodes s'executi ja que hauria de donar el mateix resultat.

Un cop hem obtingut les dades necessàries amb `lscpu` i `lstopo`, podem utilitzar nous paràmetres de la comanda `lstopo` per tal de veure l'arquitectura en un diagrama de format `.fig`:

```
lstopo --of format arxiuDestí
lstopo --of fig map.fig
```

Usant aquesta nova comanda obtenim la següent imatge (Imatge 1):



imatge 1: Diagrama d'arquitectura de boada-1..4.

La imatge (Imatge 1) descriu la memòria de cada un dels sockets de boada-1. Com hem dit abans, trobaríem la mateixa informació de boada-1 fins a boada-4 inclòs, ja que tenen la mateixa arquitectura. L'arquitectura de la memòria de cada socket és exactament igual llevat de la paritat dels blocs d'aquesta.

Podem comprovar que la màquina marca 23 GB total donat a que cada socket suma 11.5GB (que el diagrama arrodoneix a 12), sumant els 23 totals de la memòria.

## 1.2 Interactive modes

# threads	Interactive				Queued			
	user	system	elapsed	%CPU	user	system	elapsed	%CPU
1	3.94	0.00	3.94	99%	3.94	0.00	3.96	99%
2	7.99	0.00	4.00	199%	3.94	0.00	1.99	198%
4	7.98	0.03	4.01	199%	3.98	0.00	1.02	391%
8	7.98	0.06	4.02	199%	4.22	0.00	0.54	772%

Fig. 2: Taula de resultats de l'execució interactiva i encuada.

L'execució encuada (Figura 2) era resultat de la comanda `sbatch` amb els arguments necessaris. En aquest cas, la partició on s'executa el script i el fitxer executable del script:

```
sbatch partition submit-xxxx.sh
```

Per una altra banda, l'interactiva (Figura 2) era resultat de l'execució del fitxer executable del script, el programa a executar, el tamany de l'execució i el número de threads:

```
./run-xxxx.sh prog size n_th
```

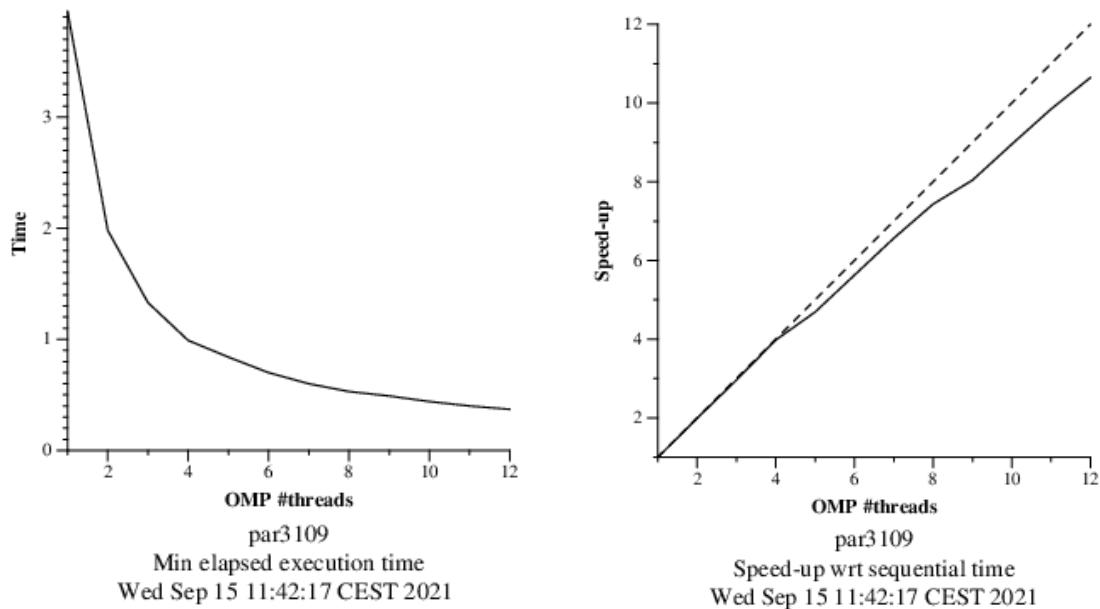
Com podem observar a la figura anterior (Figura 2), en el mode interactiu no marca diferència el número de threads que utilitzem i això és degut a que s'estan executant diferents programes a la vegada, no només el nostre, ja que estem fent la prova moltes persones alhora.

En canvi, en el mode encuat sí que veiem una millora substancial ja que es divideix el temps per la quantitat de threads en la que dividim el nostre programa.

### 1.3 Strong vs Weak scalability

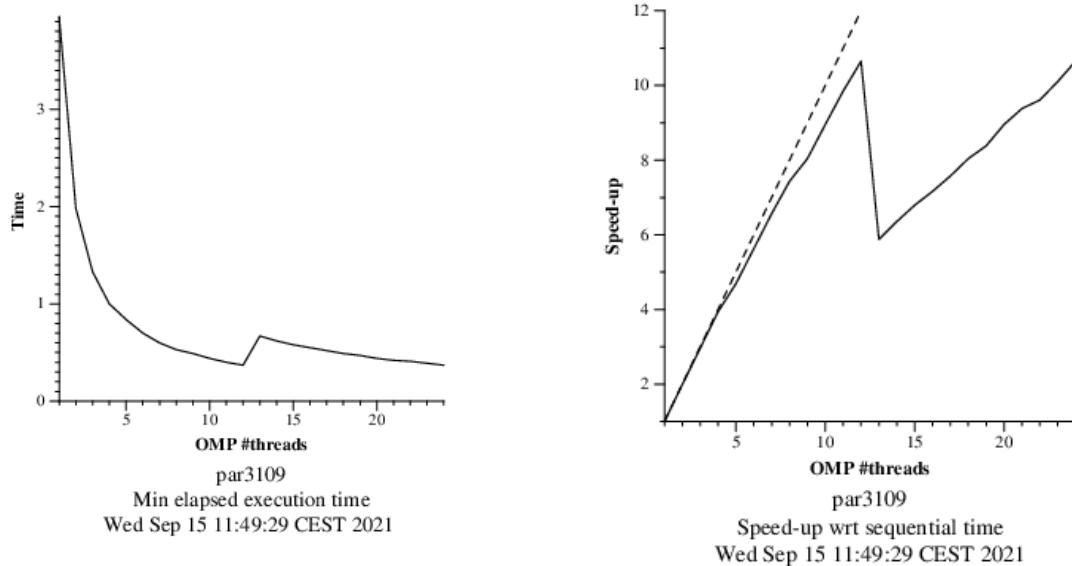
En aquest apartat considerem dos escenaris d'escalabilitat: escalabilitat forta i dèbil. L'escalabilitat forta redueix el temps d'execució d'un programa amb un tamany fixe mitjançant la divisió del programa per threads, mentre que la dèbil utilitza els threads per a augmentar el tamany del programa.

En el primer dels casos estudiats hem executat un programa amb escalabilitat forta. Podem observar als gràfics (Imatge 2) obtinguts que com més gran el número de threads, millors resultats obtenim en quant a temps d'execució. Això també es veu reflexat al gràfic de SpeedUp.



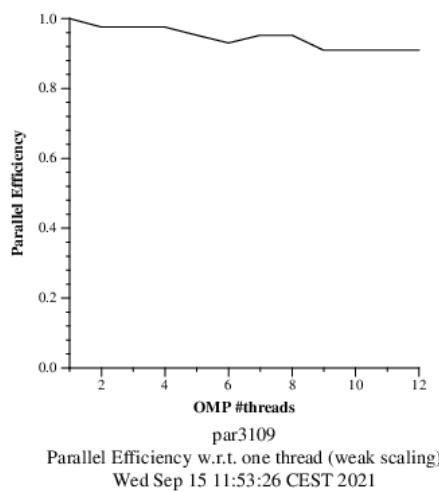
Imatge 2: Relació temps, speed-up i threads en l'execució.

En el següent cas hem provat d'augmentar el número de threads a 24. Podem observar (Imatge 3) una anomalia al gràfic al passar de 12 a 13 threads. L'esperat seria veure com segueix disminuint però, veiem com augmenta per després seguir disminuint. Això és degut a que el thread número 13 supera la relació 1-1 de threads per CPU, i una CPU s'haurà de repartir el 13è thread.



Imatge 3: Relació temps, speed-up i threads en l'execució.

En aquest últim cas, vam executar un programa amb escalabilitat dèbil. Podem observar (Imatge 4) com l'eficiència paralela es manté gairebé constant degut a que, per definició, un programa amb escalabilitat dèbil no redueix el temps d'execució per molt que s'executi de forma dividida en diferents threads.



Imatge 4: Relació eficiencia en paral·lel i threads en l'execució.

## Sessió 2

### 2.1 API de Tareador

Amb l'ajuda del Tareador hem pogut experimentar la millora substancial que proporciona paralelitzar programes. Aquest programa, genera un executable ja paral·lelitzat de la forma més eficient possible dins dels paràmetres de tasques que hem marcar dins del nostre codi gràcies a la seva senzilla API.

També ens proporciona una simulació que ens permet indicar el número de CPU màximes i evaluar el seu temps d'execució. Això ens ajuda a poder comparar el grau de paralelització la millora de rendiment dependent dels processos creats.

### 2.2 Descomposició de tasques i paral·lelisme

Version	$T_1$	$T_\infty$	Parallelism
seq	640	640	1.00
v1	640	640	1.00
v2	640	361	1.77
v3	640	155	4.13
v4	640	65	9.85
v5	640	7	91.43

Fig. 3: Taula de resultats de l'execució en diferents graus de granularitat.

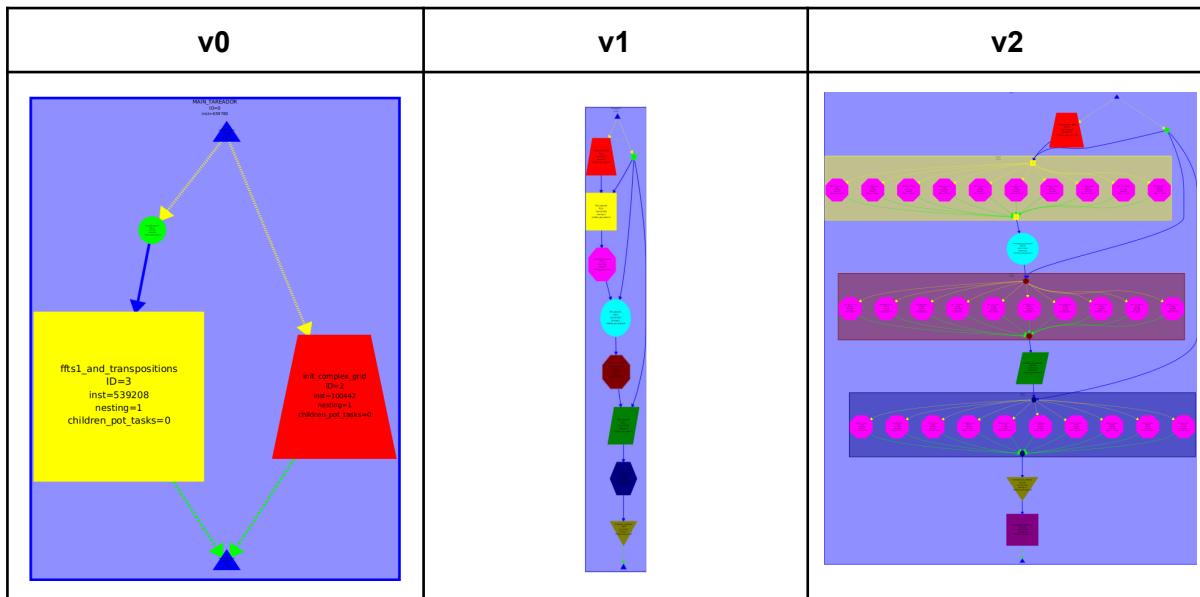
Com podem observar a la taula anterior (Figura 3), cada versió del codi per a 3D FFT (3D Fast Fourier Transform) va millorant el seu potencial paral·lelisme a mesura que apliquem de diferents formes la descomposició de tasques.

A la **versió inicial** (v0) hem executat i evaluat el programa base que se'n donava.

A la **versió v1** hem paral·lelitzat per tasques les 3 trucades a funcions dins de la tasca `ffts1` and `transpositions`, però no internament. Degut a això realment no ha hagut cap paral·lelisme ja que una depenia de l'anterior.

A la **versió v2** hem paral·lelitzat la `ffts1 planes` fent de forma paral·lela la del bucle extern `k`. Així hem obtingut finalment una millora de paral·lelisme de 1.77.

A les imatges següents podem veure la descomposició de tasques de cadascuna de les versions.

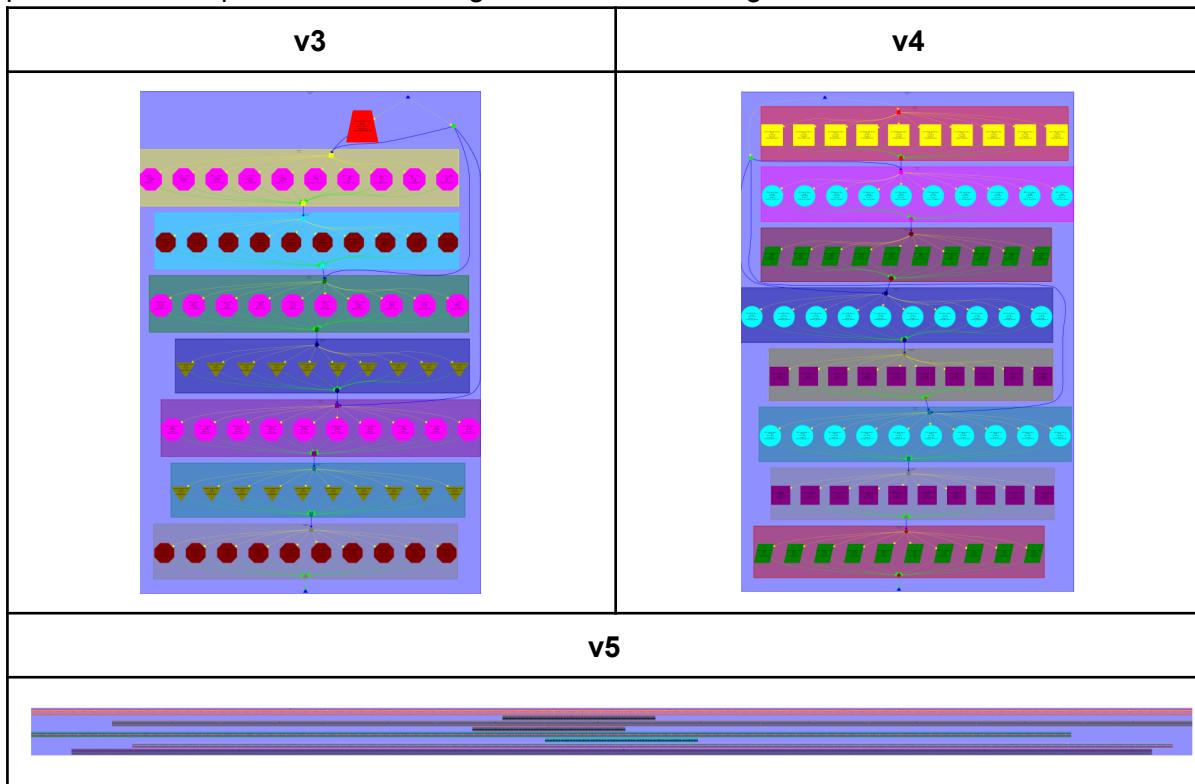


Imatge 5: Descomposició de tasques de les versions v0, v1 i v2.

A la **versió v3** hem paralelitzat transpose\_xy\_planes i transpose\_zx\_planes de la mateixa manera que a la versió 2, tot agregant la tasca al bucle extern k.

A la **versió v4** hem paral·lelitzat també el bucle exterior init\_complex\_grid.

Finalment a la **versió v5** hem paral·lelitzat tots els bucles que trobem a les funcions, interns i externs, per tal de trobar el màxim punt de millora. Hem trobat una possible millora de paral·lelisme de pràcticament 10 vegades la v4, és a dir, gairebé 100x la versió base.



Imatge 6: Descomposició de tasques de les versions v3, v4 i v5.

## 2.2.1 Relació v4 - v5

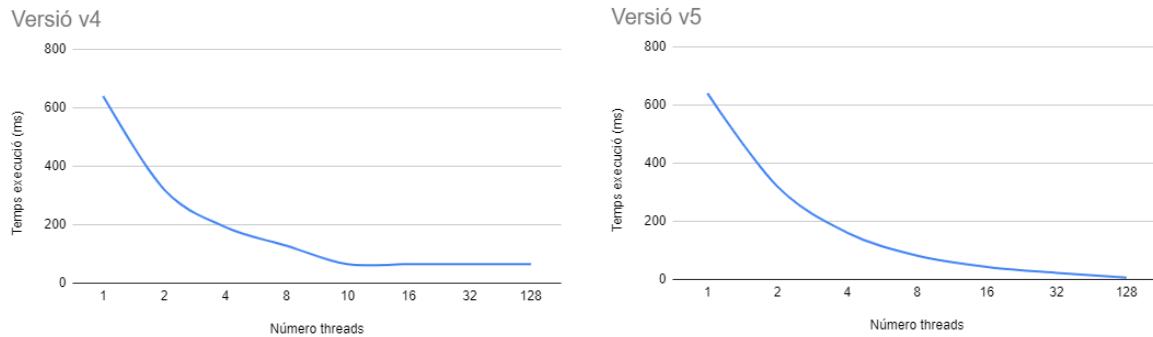
Analitzant el graf de descomposició tasques de la versió v4 (Imatge 6), hem observat com només feia 10 tasques en paral·lel degut a que només paral·lelitzavem els bucles externs. Al paral·lelitzar tots els bucles, hem obtingut moltes tasques simultànies i hem reduït el temps d'execució considerablement.

La següent taula (Figura 4) mostra el temps d'execució en milisegons (ms) de les dues versions v4 i v5 en funció del número de threads ( $X$  en  $T_X$ ). Podem observar que v4 no millora el seu temps un cop supera els 10 threads, mentres que v5 el millora de forma lineal.

Version	$T_1$	$T_2$	$T_4$	$T_8$	$T_{16}$	$T_{32}$
v4	640	320	192	128	65	65
v5	640	320	161	82	43	23

Fig. 4: Taula de resultats de l'execució amb diferents threads de v4 i v5.

En els dos gràfics següents (Imatge 7) podem comprovar com la versió v4 no redueix el temps d'execució passats els 10 threads mentre que la versió v5 segueix baixant de forma regular.



Imatge 7: Gràfics de temps per threads de v4 i v5.

# Sessió 3

Durant aquesta sessió, aprendrem a utilitzar Paraver, una interfície gràfica que ens permet visualitzar paràmetres d'execució dels nostres programes, mitjançant la generació de trances a través de la llibreria Extrae.

## 3.1 Obtenint les mètriques de paral·lelització de 3DFFT amb Paraver

Per tal d'obtenir les dades a analitzar, hem executat el codi de l'arxiu `3dfft_omp.c` amb l'script de Extrae per tal d'obtenir el fitxer `.prv` que ens permet obrir el programa Paraver i analitzar la traça.

Un cop obert el programa i seleccionada la traça, hem activat les flags i hem utilitzat la configuració més adequada per obtenir els valors de paral·lelisme:

`OMP_implicit_tasks_profile.cfg`

$$\Phi = T_{Par}/(T_{seq} + T_{par})$$

$$ideal S_8 = \frac{1}{(1-\Phi) + (\Phi/8)}$$

Version	$\Phi$	Ideal S8	T1 (s)	T8 (s)	Real S8
Initial version in <code>3dfft_omp.c</code>	0.60	2.11	2.74	1.57	1.75
New version with improved $\Phi$	0.88	4.35	2.54	1.02	2.49
Final version with reduced parallelisation overheads	0.88	4.35	2.28	0.7	3.26

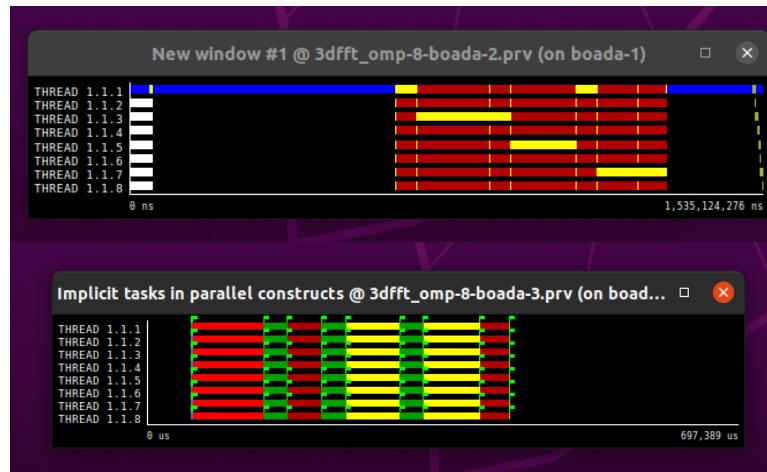
Fig. 5: Taula de resultats de l'execució amb 1 i 8 threads de les diferents versions de `3dfft_omp.c`.

Per tal d'obtenir les dades (Figura 5) hem hagut d'actuar diferent en funció de la fila de la taula que havíem d'analitzar, sempre fent servir dues versions: la executada en un thread, i la executada en 8 threads.

### 3.1.1 Initial version

En aquesta versió tractem les dades d'un codi que no està paral·lelitzat i que té per conseqüència un valor menor de  $\Phi$ .

Podem observar clarament (Imatge 8) les diferents seccions del programa gràcies a la configuració `implicit_tasks` dins Paraver.



Imatge 8: Timeline de la primera versió en vuit threads.

### 3.1.2 New version

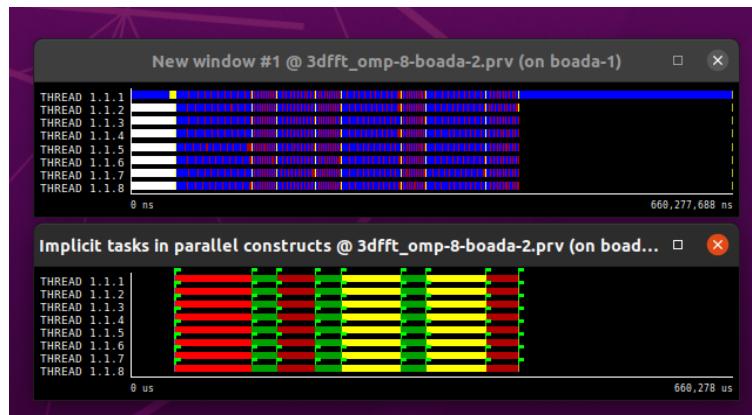
A la versió millorada del codi anterior hem introduït les comandes pragma per paral·lelitzar la funció `init_complex_grid` i com podem observar al gràfic (Imatge 9) hi ha una millora considerable en el temps d'execució paral·lel amb 8 threads ja que passem de 1570 a 1020 ms.



Imatge 9: Timeline de la versió millorada en vuit threads.

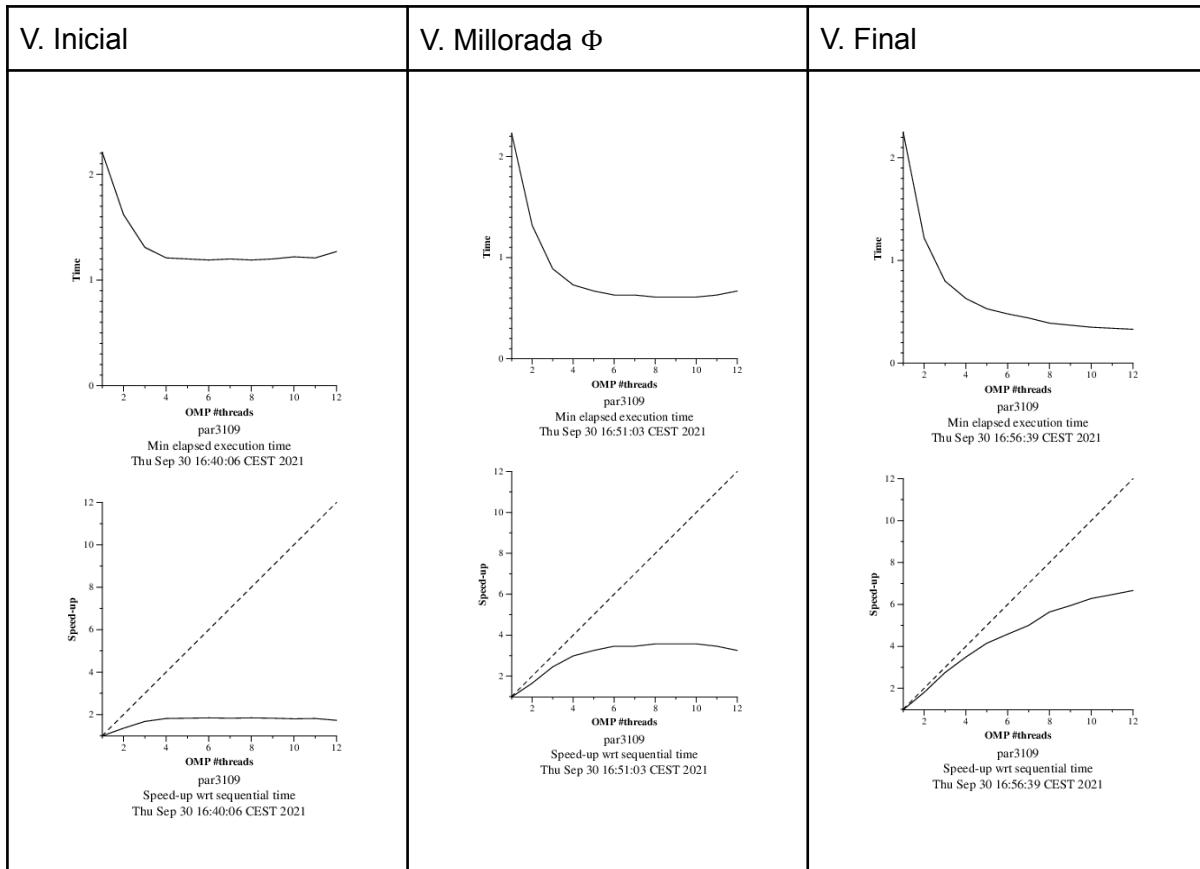
### 3.1.3 Final version

A la versió final del programa, hem augmentat la granularitat ja que hem passat de paral·lelitzar les iteracions més internes del bucle a les més externes. Això ha reduït els overheads i per tant, ha tornat a millorar el temps d'execució del programa sent ara 700ms de temps usant 8 threads en paral·lel.



imatge 10: Timeline de la versió final en vuit threads.

### 3.1.4 Comparant strong scalability



imatges 11,12 i 13 (d'esquerra a dreta): Relacions d'eficiència per threads en les tres versions del programa.

Com podem observar, la versió inicial (imatge 11) gairebé no té millora en el seu temps d'execució a partir de 4 threads i de fet, a partir de 12 comença a empitjorar lleugerament (com vist en l'apartat 1.3). Això comporta que el seu speed-up sigui gairebé constant a partir de 4 threads i un speedup total aproximat de 1.75x.

A la versió millorada (imatge 12), per una altra banda, podem observar un cas molt similar on a partir dels 4 threads es manté gairebé constant, però el seu speedup és aproximadament 3x.

Per últim, la versió final (imatge 13) no es queda constant a partir de 4 i va millorant més a poc a poc el seu temps d'execució. Podem veure que un thread més sempre comporta una millora (tot i que a partir de 10 threads sembla que no massa substancial). L'speed up arriba a 5x aproximadament als 12 threads i 4.5x als 8 threads.

## 4. Conclusió

Durant les tres sessions d'aquesta primera entrega de laboratori, hem après a utilitzar eines que ens seran útils durant el transcurs de l'assignatura. En primer lloc vam aprendre a connectar-nos al boada, i un cop a dins vam aprendre a fer servir les eines que hi trobem, com per exemple el Tareador i el Paraver.

Gràcies a aquestes eines hem aconseguit veure de forma gràfica i pràctica la paral·lelització de tasques i com aquesta afecta a l'eficiència d'un programa, entenent millor així els conceptes explicats a teoria.