PARAL·LELISME

Laboratori 1: Experimental setup and tools

PAR1105

Albert Borrellas Maldonado

Víctor Martínez Murillo

09/10/2019

Tardor 2019-2020

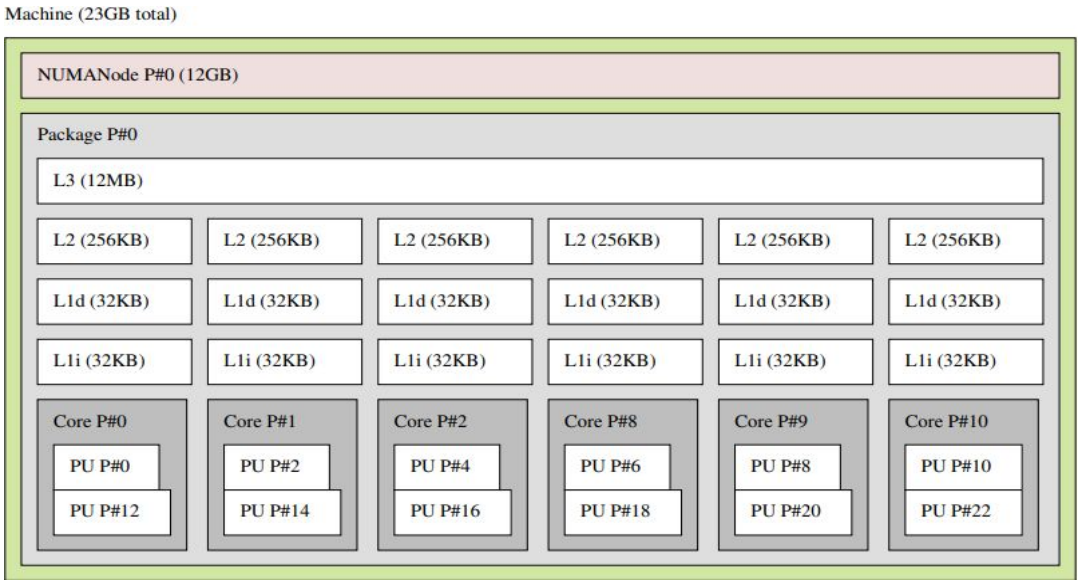
Sessió 1: Experimental Setup

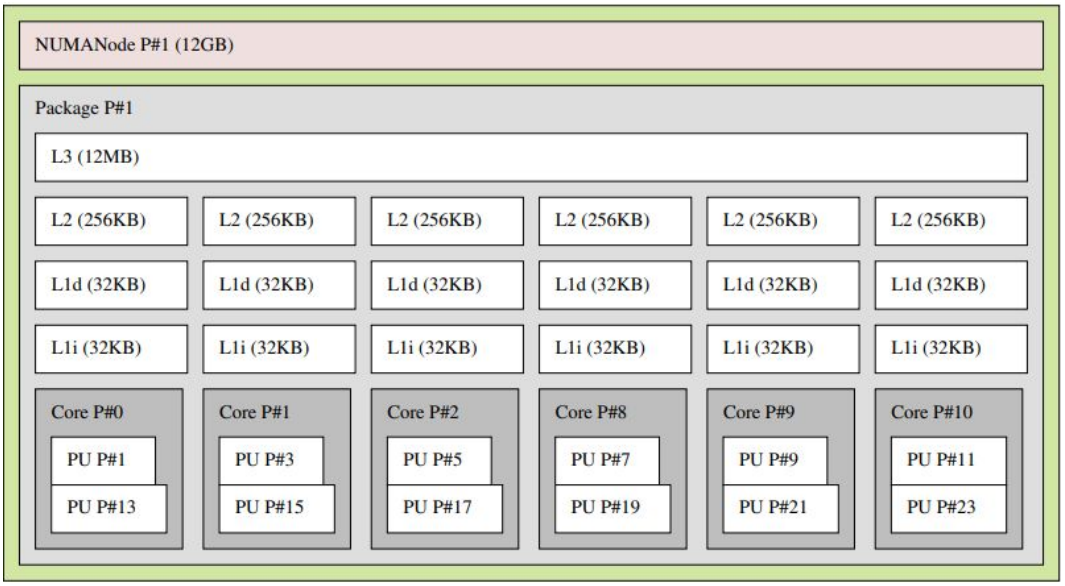
L'objectiu de la primera sessió és familiaritzar-nos amb l'entorn que treballarem a partir d'ara per dur a terme l'estudi del paral·lelisme. En el nostre *setup* treballarem amb un servidor multiprocessador anomenat Boada, que és un clúster dividit en 8 nodes amb diferents arquitectures. Per executar els nostres programes, ho farem interactivament (boada-1) o amb el sistema de cues (boada-2 a boada-8).

Node architecture and memory

Per visualitzar l'arquitectura de nodes usem les comandes *lscpu* (textualment) i *lstopo* (la imatge 1 s'obté afegint-li el flag *--of fig* a aquesta última).

L'arquitectura dels nodes boada-1 fins a boada-4 és exactament igual ja que disposen de la mateixa generació de processadors, en aquest cas, un *Intel Xeon E5645*.





Imatge 1: Arquitectura de memoria de boada-1 a boada-4. Particularment, boada-1.

El node boada-5 compta amb un processador *Intel Xeon E5-2620 v2 + Nvidia K40c* (unitat central + unitat gràfica). Finalment, els nodes boada-6 a boada-8 usen un processador *Intel Xeon E5-2609 v4.* Mitjançant les anteriors comandes completem la taula 1:

	boada-1 to boada-4	boada-5	boada-6 to boada-8
Number of sockets per node	2	2	2
Number of cores per socket	6	6	8
Number of threads per core	2	2	1
Maximum core frequency	2395 MHz	2600 MHz	1700 MHz
L1-I cache size (per-core)	32 KB	32 KB	32 KB
L1-D cache size (per-core)	32 KB	32 KB	32 KB
L2 cache size (per-core)	256 KB	256 KB	256 KB
Last-level cache size (per-socket)	12 MB	15 MB	15 MB
Main memory size (per socket)	12 GB	32 GB	16 GB
Main memory size (per node)	24 GB	64 GB	32 GB

Taula 1: Característiques de l'arquitectura del servidor Boada.

Com es pot observar, la major diferència és a la memòria principal (tant per socket, com per node) on destaca la gran capacitat que té el boada-5 degut a la doble unitat que monta.

Els últims nodes monten més cores per socket gràcies a la familia de processadors però, en canvi, perd en threads per core. També hi ha una gran disminució de la freqüència màxima respecte els altres.

Finalment, els primers tenen menys memòria a l'últim nivell de cache (L3), que és compartida per tot els cores.

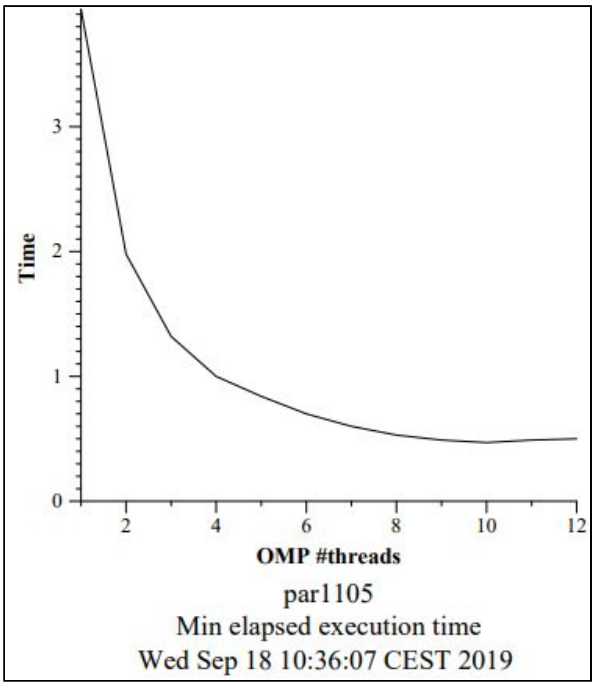
Strong vs. weak scalability

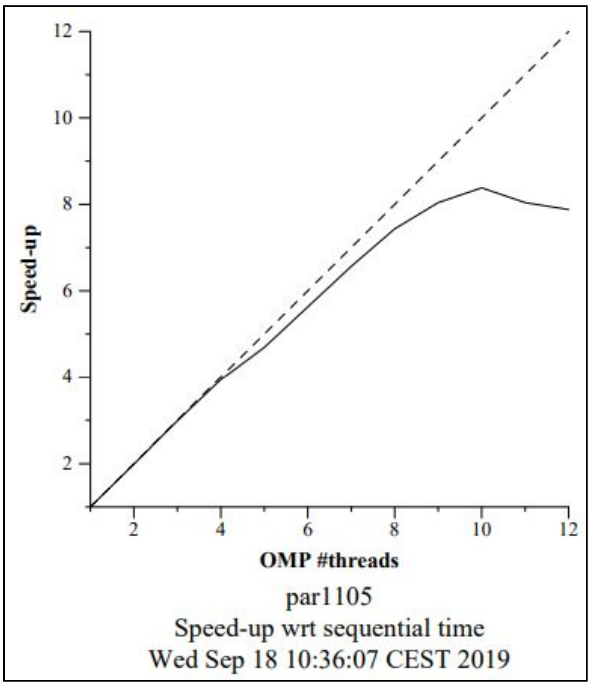
La mesura per excel·lència per medir el rendiment és l'anomenat *speed-up*. Un dels principals propòsits del paral·lelisme és millorar-lo. Tot i això, importen altres mesures, com la que anem a estudiar ara, la escalabilitat.

La escalabilitat és com es comporta la evolució del *speed-up* quan s'augmenta el nombre de processadors. Compararem la *strong* en contraposició a la *weak*.

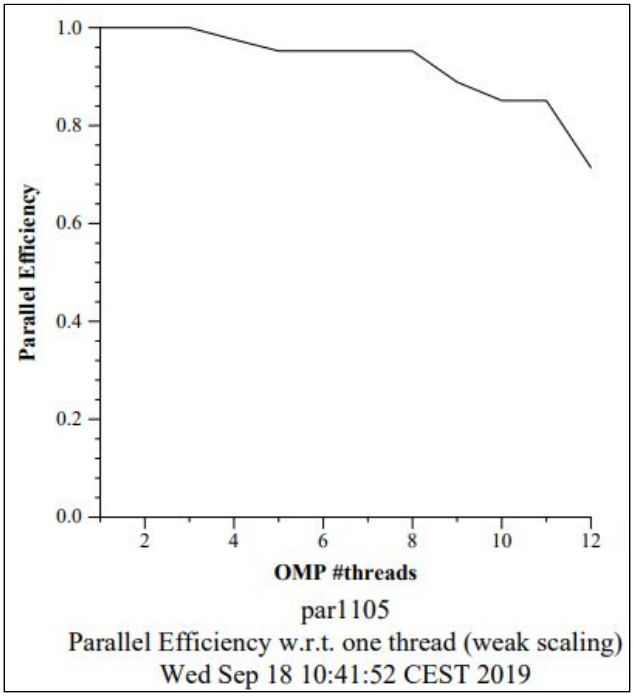
L'objectiu de la *strong scalability* és disminuir el temps d'execució d'un problema de mida fixa augmentant el número de processadors. En canvi, a la *weak scalability* importa mantenir el temps d'execució a mesura que un problema augmenta la seva mida mentre augmenta també el número de processadors (de forma proporcional respecte la mida) per aconseguir-ho.

Analitzem l'efecte de les dues al programa **pi_omp.c** que calcula el nombre pi de forma paral·lela i, gràcies als scripts ***submit-strong-omp.sh*** i ***submit-weak-omp.sh*** (enviats al sistema de cues), juntament a la comanda *gs*, obtenim els següents gràfics:





Imatges 2 i 3: Temps d'execució i speed-up de l'escalabilitat tipus *strong* (respecte el *#threads*).



Imatge 4: Eficiència (paral·lela) de la escalabilitat tipus *weak* (respecte el *#threads*).

A la imatge 3 podem veure que, efectivament, amb l'augment de *threads*, el *speed-up* comença augmentant de forma lineal (que seria el cas ideal) fins arribar als 4 threads. A partir d'aquí, es desvia del cas ideal fins arribar als 10 threads que és on aconsegueix el màxim *speed-up* i amb l'augment de *threads* només empitjora.

Conseqüentment, a la imatge 2, el temps disminueix de forma exponencial, fins arribar als 10 threads quan arriba al límit de la millora paral·lela.

A la imatge 4, veiem que amb l'augment de *thread*s la paral·lelització tampoc és ideal i la eficiència (*speed-up* entre #processadors) comença a baixar, entre altres coses degut als *overhead*s i sincronització entre threads que provoca el paral·lelisme.

Sessió 2: Systematically analysing task decompositions with Tareador

L'objectiu de la segona sessió és familiaritzar-nos amb l'eina *Tareador* que ens permet observar el potencial de millora d'una estratègia de descomposició de tasques aplicada al codi seqüencial que volem paral·lelitzar. *Tareador* només necessita que s'identifiquin les parts del codi que necessiten ser avaluades.

A partir del codi de **3dfft_tar.c** (*Fast Fourier Transform*), utilitzem el *Tareador* per saber quin temps d'execució tindríem seguint l'estratègia de paral·lelisme de les diferents versions amb un i infinits processadors ("infinits processadors" vol dir el màxim número de processadors fins que es limita la millora de temps tot i què augmentem els processadors).

Versió		T_1 (ns)	T_∞ (ns)	**Paral·lelisme**
seqüèncial		639.780.001	639.707.001	1
v1		639.780.001	639.707.001	1
v2		639.780.001	361.199.001	1,77
v3		639.780.001	154.354.001	4,15
v4		639.780.001	64.018.001	9,99
v5		639.780.001	55.820.001	11,46

Taula 2: Anàlisi del temps amb un processador, temps amb infinits processadors i del paral·lelisme de les diferents versions del codi 3dfft_tar.c .

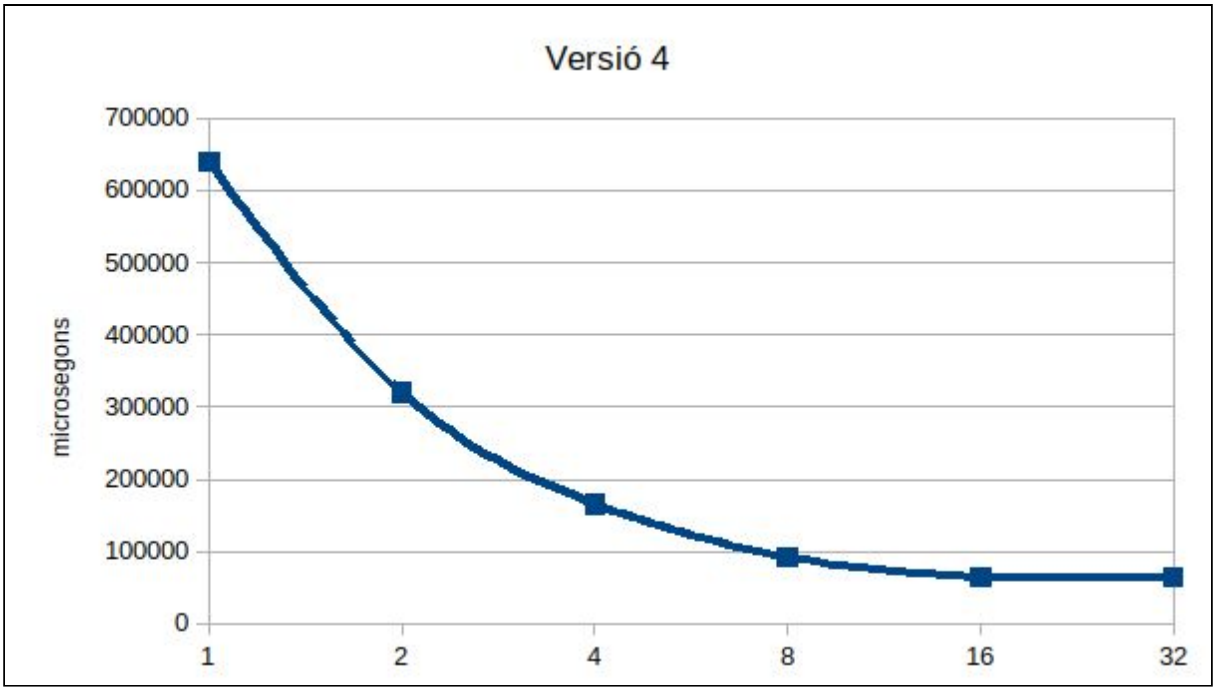
A la taula 2, en el primer que ens fixem és que T_1 sempre té el mateix valor, això és normal ja que encara que tinguem una estratègia de paral·lelització, si només utilitzem un processador sempre s'executarà de forma seqüencial.

A la versió 1, modifiquem la posició al codi de les comandes de paral·lelització: de quan es crida la funció cap a la primera línea de la funció, això no provoca canvis en el temps d'execució.

Tot comença a canviar a la versió 2, on fem que el bucle *for* extern (bucle-k) de dins de la funció **ffts1_planes** s'executi en paral·lel i observem una reducció del temps amb infinits processadors de fins a quasi la meitat respecte la v1. El que provoca aquest canvi dins del *for*, és la reducció de la granularitat de les tasques.

A la versió 3 (acumulant els canvis de la v2), reduïm la granularitat en els bucles externs de les funcions **transpose_xy_planes** i **transpose_zx_planes**. Com es reflecteix a la taula, el paral·lelisme ha augmentat considerablement, de 1,77 a 4,15 ja que ara hi ha tres funcions que paral·lelitzen cada iteració del bucle-k.

A la versió 4 (acumulant els canvis de la v3), fem la mateixa granularització amb l'última funció: **init_complex_grid**. Aquest cas duplica el paral·lelisme de l'anterior.



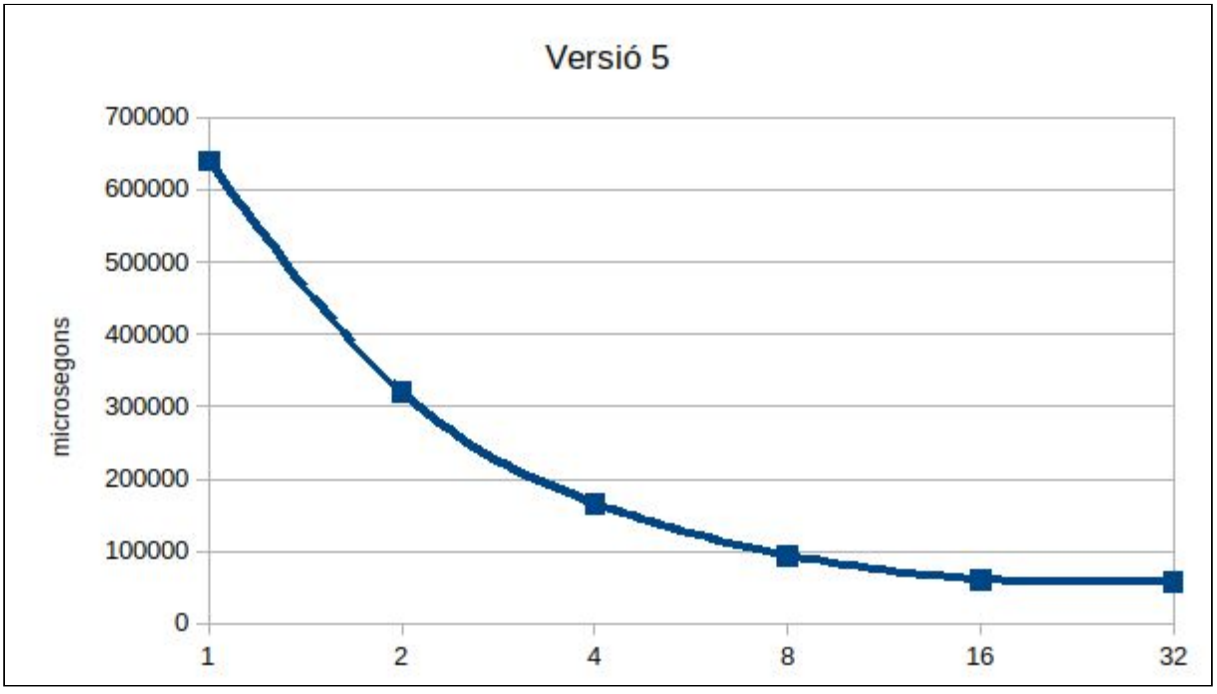
Imatge 5: Anàlisi temporal de la v4 canviant el #processadors d'1, 2, 4, 8, 16 i 32.

En aquesta gráfica podem veure com a mida que augmenten el número de processadors el temps disminueix fins al punt que el temps comença a ser constant, és a dir, fins que arribem a T_∞, en aquest cas, T_{16}.

Finalment, a la versió 5 (veure codi 1), hem agafat la v4 i em modificat la granularització del bucle-k al bucle-j. Es podria haver fet encara més petita, ficant-li la paral·lelització al bucle-i però tardaria massa ja que ho faria a cada iteració del bucle intern (tres *for*'s en total).

```
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");
    }
  }
```

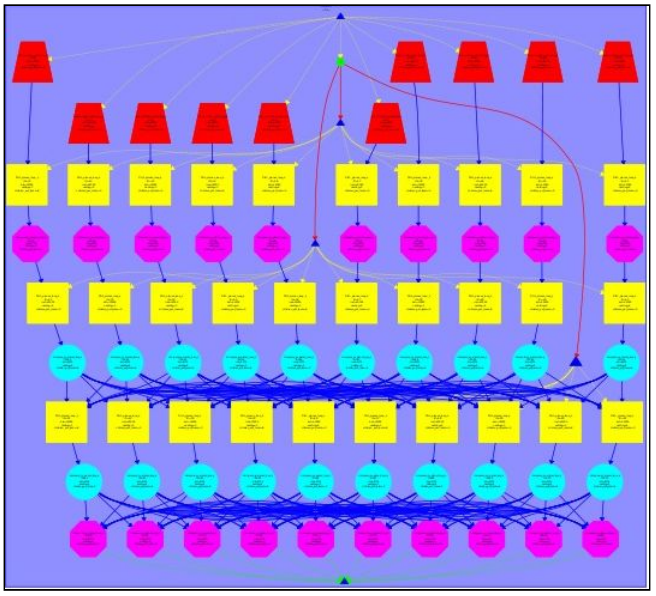
Codi 1: Versió 5, comandes de paral·lelisme al bucle-j.



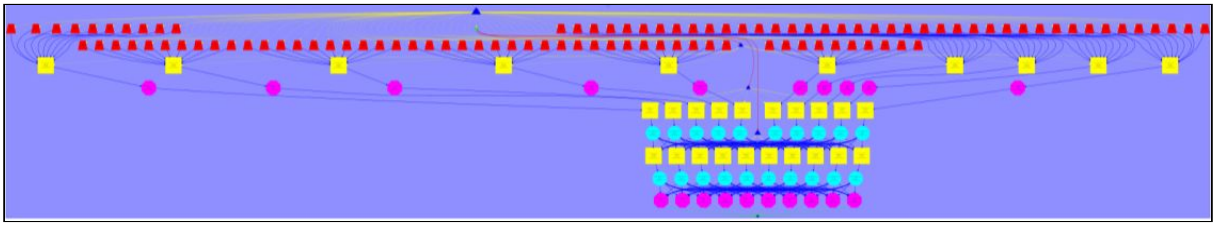
Imatge 6: Anàlisi temporal de la v5 canviant el #processadors d'1, 2, 4, 8, 16 i 32.

La imatge 6 resulta molt semblant a la imatge 5 però, efectivament, el gránul al ser més petit, millora la paral·lelització, i per això T_∞ ja no és 16. De fet tampoc és 32 (57.928.001 ns), sent T_∞ = 55.820.001 ns.

Ara veiem la descomposició en tasques que proporciona el *Tareador* mitjançant el graf de dependències, particularment de les versions v4 i v5 per poder comparar la influencia de la mida del gràNul:



Imatge 7: Descomposició en tasques de la v4 via *Tareador*.



Imatge 8: Descomposició en tasques de la v5 via *Tareador*.

Comparant les imatges 7 i 8, veiem que la escalabilitat a un nombre major de processadors de la v5 és superior a la de la v4 ja que, alhora, la v5 té més tasques en paral·lel que la v4, d'aquesta manera, per molt que la v4 tingui molts processadors assignats, per culpa de les dependències no aprofitarà tots els processadors per fer la paral·lelització.

Sessió 3: Understanding the execution of OpenMP programs

L'objectiu d'aquesta última sessió és familiaritzar-nos amb l'entorn *Paraver*, que ens permet reunir informació i visualitzar com s'executa un programa paral·lel en *OpenMP*. També es fa servir la llibreria *Extrae* que ens proporciona els arxius binaris amb tota la informació d'execució (anomenats *traces*) i després amb *Paraver* visualitzem la *trace* i així analitzem l'execució del programa.

Anem a analitzar la v5 de la sessió anterior, en aquest cas l'arxiu és el **3dfft_omp.c**.

La Taula 3 és un anàlisi d'aquest codi on l'executem amb el script **submit-omp-i.sh** amb un i vuit *threads*. Un cop executat ens genera les tres *traces* (.prv, .pcf, .row) de les quals utilitzarem el .prv. L'obrirem amb l'eina *wxparaver* i ens mostrarà un *timeline* amb el temps quan s'executa seqüencialment i paral·lelament. Utilitzant l'opció de flags juntament amb la configuració "parallel functions" que ens proporciona *Paraver* podem saber T_{par} (el temps que s'executa en paral·lel) i podrem obtenir totes les dades demanades amb les següents fórmules:

$$T_1 = T_{seq} + T_{par}$$

$$S_P = S_1 / S_P$$

$$\varphi = T_{par} / T_1$$

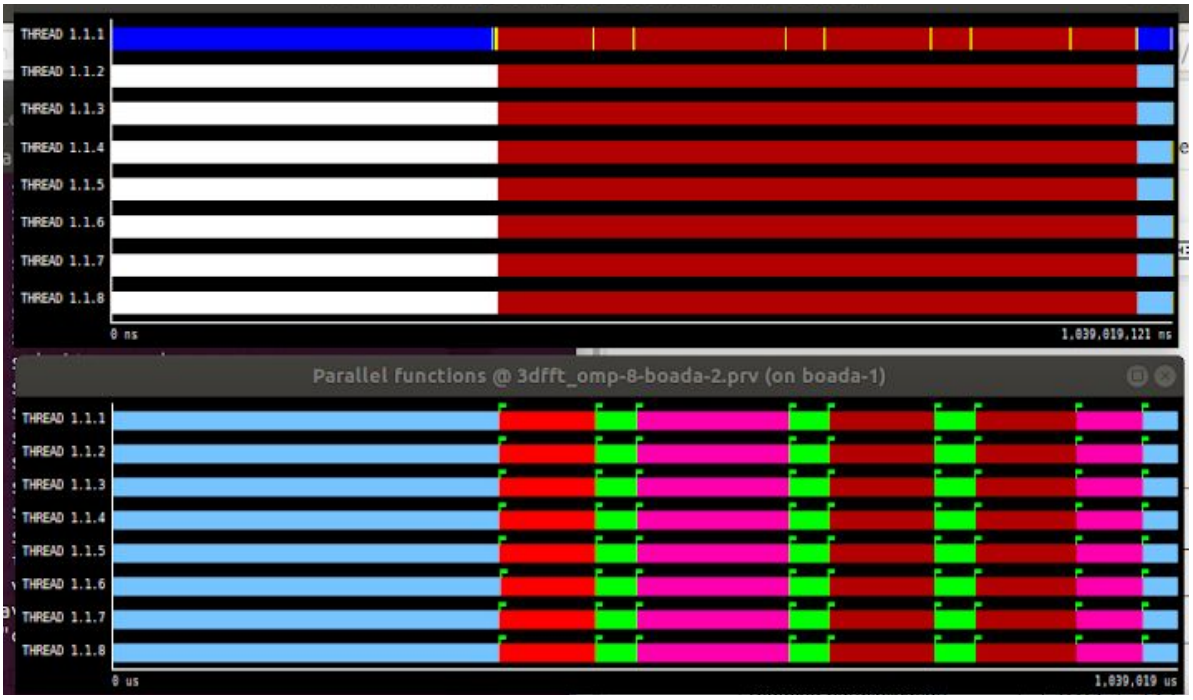
$$S_\infty = 1 / (1-\varphi)$$

Versió		φ	S_∞	T_1 (us)	T_8 (us)	S_8
initial version in 3dfft_omp.c		0,655	2,9	2.340.409,90	1.379.871,15	1,7
new version with improved φ		0,9	10,56	2.346.235,97	1.039.019,12	2,26
final version with reduced parallelisation overheads		0,89	9,21	2.305.497,84	602.368,88	3,83

Taula 3: Anàlisi de les mesures de rendiment i temporals amb un i vuit *threads*.

La primera fila són les dades de la versió inicial on **init_complex_grid** no es paral·lelitza, per tant, el rendiment real és menor i l'ideal (infinits processadors) és molt inferior a la resta.

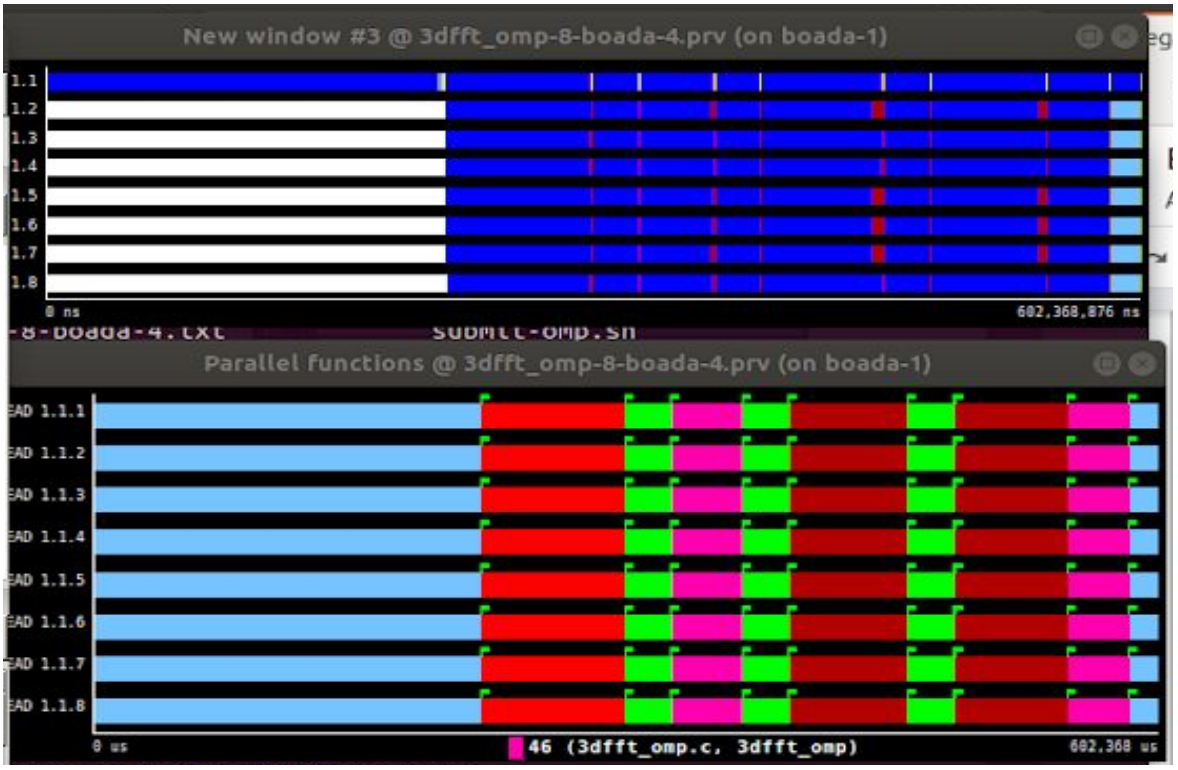
La segona fila correspon a una versió millorada del codi anterior on ara sí que pararel·litzem (hem introduït les comandes “pragma”) la funció **init_complex_grid** i com podem observar hi ha una millora, tot i què, no és tan gran com podríem esperar degut als overheads generats pel paral·lelisme.



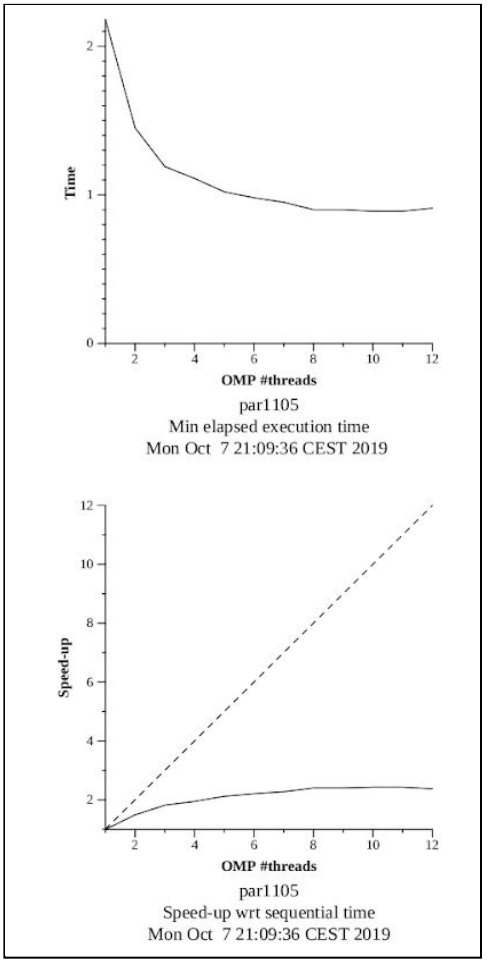
Imatge 9: *Timeline* de la segona versió de **3dfft_omp.c** executat amb 8 threads via *Paraver.*

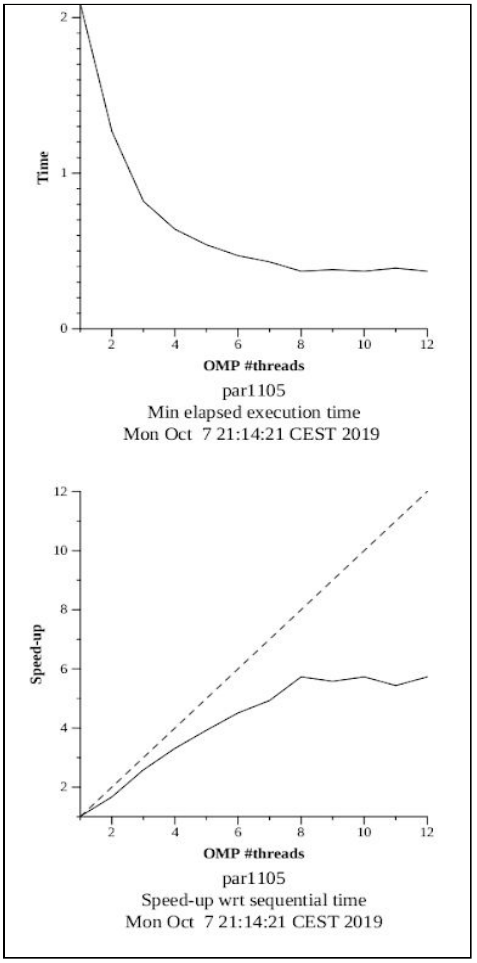
A l’última versió del codi passem de paral·lelitzar dins del bucle-k a paral·lelitzar fora dels bucle, això redueix els overheads a l’hora de paral·lelitzar i, per tant, obtenir una millora destacable tant temporal com en rendiment. Però com podem veure tant a la taula 3, com a la imatge 10, encara que el seu temps d’execució millora bastant, el temps que el programa s’executa en seqüencial (línia de *timeline* en blau) disminueix el que implica que tant φ com S_∞ siguin pitjors que a la versió 2.

Per acabar la sessió 3, hem obtingut les gràfiques (imatges 11 i 12) que ens mostren la escalabilitat *strong* (temps i *speed-up*) d’aquestes dues versions. La versió amb la φ millorada, s’allunya molt ràpid del cas ideal i comença a ser quasi constant. En canvi, la versió amb *overheads* reduïts prolonga més temps l’efecte del cas ideal i la seva desviació és menor respecte la versió anterior, tot i què, com és normal, creix al llarg del temps.



Imatge 10: *Timeline* de la tercera versió de **3dfft_omp.c** executat amb 8 threads via *Paraver.*





Imatge 11: Escalabilitat *strong*, versió φ millorada. Imatge 12: Escalabilitat *strong*, versió amb overheads reduïts.

Conclusions finals

Aquesta pràctica ha estat el primer *hands-on* amb l'entorn que usarem a partir d'ara a totes les sessions per dur a terme l'estudi i anàlisi del paral·lelisme als nostres programes.

La primera sessió ens ha servit per adonar-nos que no només importa el programa en si, sinó també el rerefons. És a dir, tota mesura de rendiment o temporal variarà en funció de l'arquitectura (de nodes i memòria) en la que treballem. A la segona part d'aquesta sessió, hem après que hi ha diferents formes d'enfocar la tasca de mesurar el rendiment.

A la segona hem vist com afecta la col·locació del codi per paral·lelitzar a l'hora de crear tasques (en funció de la mida del gränul). Tant és així, que ens adonem que és molt importar escollir una bona granularització ja que a vegades la part paral·lela del programa no és massa gran i prendre bones decisions marca la diferència.

Finalment, a l'última hem treballat amb l'entorn Paraver per fer l'anàlisi de les mesures, on hem treballat en petits canvis al codi que fan millorar l'eficiència destacablement i això s'ha de tenir en consideració a l'hora de programar.