Lliurament
Lab 1

Jacobo Moral
Carles Pàmies

Grup 41 – Par4101

04/10/17
Curs Tardor 2017-2018

Node architecture and memory

1. Draw and briefly describe the architecture of the computer in which you are doing this lab session (number of sockets, cores per socket, threads per core, cache hierarchy size and sharing, and amount of main memory).

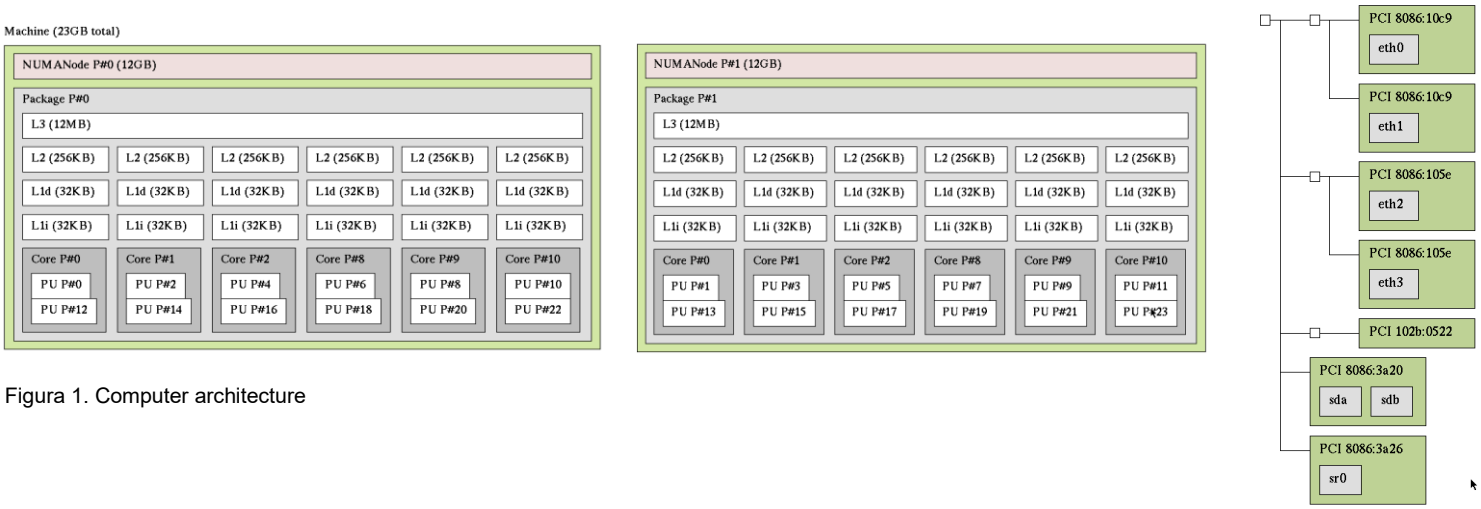


Figura 1. Computer architecture

Per tal de crear les imatges anteriors sobre l'arquitectura de la màquina, hem executat les següents instruccions:

> *lstopo –of fig map.fig*
#crea un arxiu map.fig amb la informació sobre la màquina.

> *xfig map.fig*
#obre l'arxiu creat anteriorment amb el programa Xfig, per poder veure la informació de la màquina gràficament.

Number of sockets	Cores per socket	Threads per core	Main Memory
2	6	2	24GB (2x12GB)

Cache	Size
L1i	32KB
L1d	32KB
L2	256KB
L3	12MB (12288KB)

Pel que fa a la jerarquia de cache, cadascun dels cores té una L1i, una L1d i una L2 pròpia i, a més, una L3 que comparteix només amb els altres cinc cores del seu socket o node.

Per tant, tenim en total dos L3, i dotze de cadascuna de les L1i, L1d i L2.

Timing sequential and parallel executions

2. Describe what do you need to add to your program to measure the elapsed execution time between a pair of points in the program, clearly indicating the library header file that needs to be included, the library functions that need to be invoked, the data structure and its fields.

Codi extret de l'arxiu *pi_seq.c.* Tot el següent és necessari per poder mesurar l'*elapsed time* i treure'l per pantalla.

```
#include <sys/time.h> [1]

...

double getusec_() {
        struct timeval time;
        gettimeofday(&time, NULL); [2]
        return ((double)time.tv_sec * (double)1e6 + (double)time.tv_usec);
}

#define START_COUNT_TIME stamp = getusec_();
#define STOP_COUNT_TIME(_m) stamp = getusec_() - stamp;\
                            stamp = stamp/1e6;\
                            printf ("%s%0.6f\n",(_m), stamp);[3]

…

START_COUNT_TIME;

… [4]

STOP_COUNT_TIME("");
```

[1] Llibreria necessària per poder mesurar l'*elapsed time* (temps real de rellotge).

[2] Funció per la qual necessitem la llibreria.

[3] Segons el man, sec es posa en segons i usec en microsegons.

[4] Codi temps del qual volem mesurar.

3. Plot the speed–up obtained when varying the number of threads (strong scalability) and problem size (weak scalability) for pi omp.c. Reason about how the scalability of the program.

Tant al *strong scalability* com al *weak scalability*, els gràfics s'han obtingut mitjançant la següent seqüència de comandes:

> *qsub -l execution submit-type-omp.sh*
type = weak/strong; aquesta instrucció manda executar de una manera isolada el programa pi_omp en weak o strong scalability. A més, crea

> *ps2pdf nom_del_arxiu.ps*
converteix l'arxiu que conté els gràfics(.ps) a pdf

>*display nom_del_arxiu.pdf*
obre l'arxiu pdf

Strong scalability

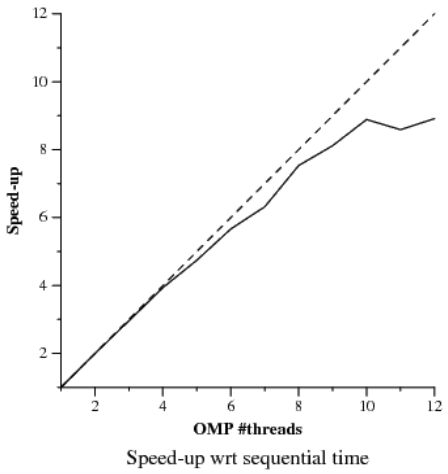


Figura 2. Sequential speed-up

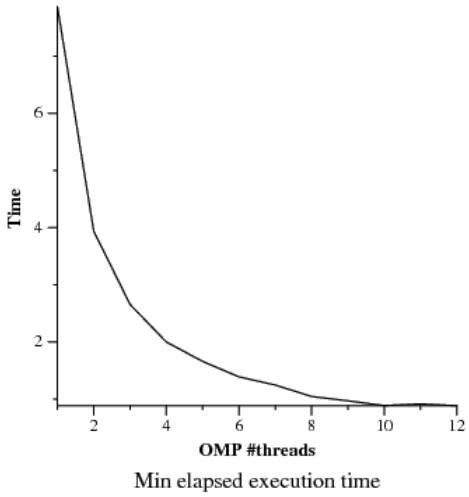


Figura 3. Sequential exec. time

Podem observar com en aquest cas de *strong scalability* (augmenta el nombre de threads a cada execució del programa), el temps de cada execució disminueix (Figura 3). Això és degut a que reparteix el mateix programa en diferents threads i, per tant, el temps que trigarà en executar-se serà aproximadament $t_n = t_o/n$, amb n = nombre de threads i t_o el temps amb només un thread. Veiem també que a partir de 10 threads ja gairebé no millora el temps.

Al gràfic del speed-up (Figura 2), podem veure això mateix: el speed-up augmenta proporcionalment al nombre de threads fins a 10. A partir d'aquest nombre de threads, el speed-up segueix constant. Speed-up = t_o/t_n. Per tant, té sentit que augmenti proporcionalment al nombre de threads. Substituint «$t_n = t_o/n$» a la fòrmula del speed-up:
Speed-up = $t_o/(t_o/n) = n$. Això indica que la gràfica hauria de ser lineal i proporcional a n (nombre de threads), com hem dit abans.

Weak scalability

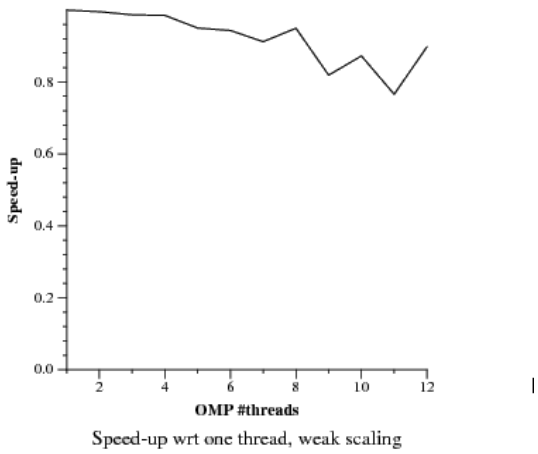


Figura 4. OMP speed-up

Pel cas del *weak scalability (*a cada execució augmenta el nombre de threads proporcionalment a la mida del programa), el speed-up es més o menys constant.
Això és degut a que quan augmentem el nombre de threads, també augmentem la mida del programa i, per tant, el temps que triga és més o menys constant a cada iteració.
Per tant, $t_o \approx t_n$. Alhesores, speed-up = $t_o/t_o = 1$ (gràfic constant a y = 1).

Visualizing the task graph and data dependences

4. Include the source code for function dot product in which you show the Tareador instrumentation that has been added to study the potential parallelism in the code. This instrumentation has to appropriately define tasks and filter the analysis of variable(s) that cause the dependence(s).

```
void dot_product (long N, double A[N], double B[N], double *acc){

        double prod;
        int i;
        *acc=0.0;

        for (i=0; i<N; i++) {

                tareador_start_task("dot_product_inner");[1]
                prod = my_func(A[i], B[i]); [2]
                tareador_disable_object(acc); [3]
                *acc += prod; [4]
                tareador_enable_object(acc); [3]
                tareador_end_task("dot_product_inner");[1]

        }
}
```

[1] Tot el codi entre aquestes dos instruccions será considerat tasca i, per tant, es repartirà entre els diferents threads. El paràmetre és només un àlies i podem ficar un que ens agradi, però que al mateix temps representi la tasca.

[2] Codi propi de la tasca.

[3] Aquestes instruccions eliminen dependències corresponents al codi que hi ha entre la instrucció de disable i la d'enable.

[4] Codi que accedeix a memoria.

5. Capture the task dependence graph for that task decomposition and the execution timelines (for 8 processors) that allow you to understand the potential parallelism attainable. Briefly comment the relevant information that is reported by the tools.

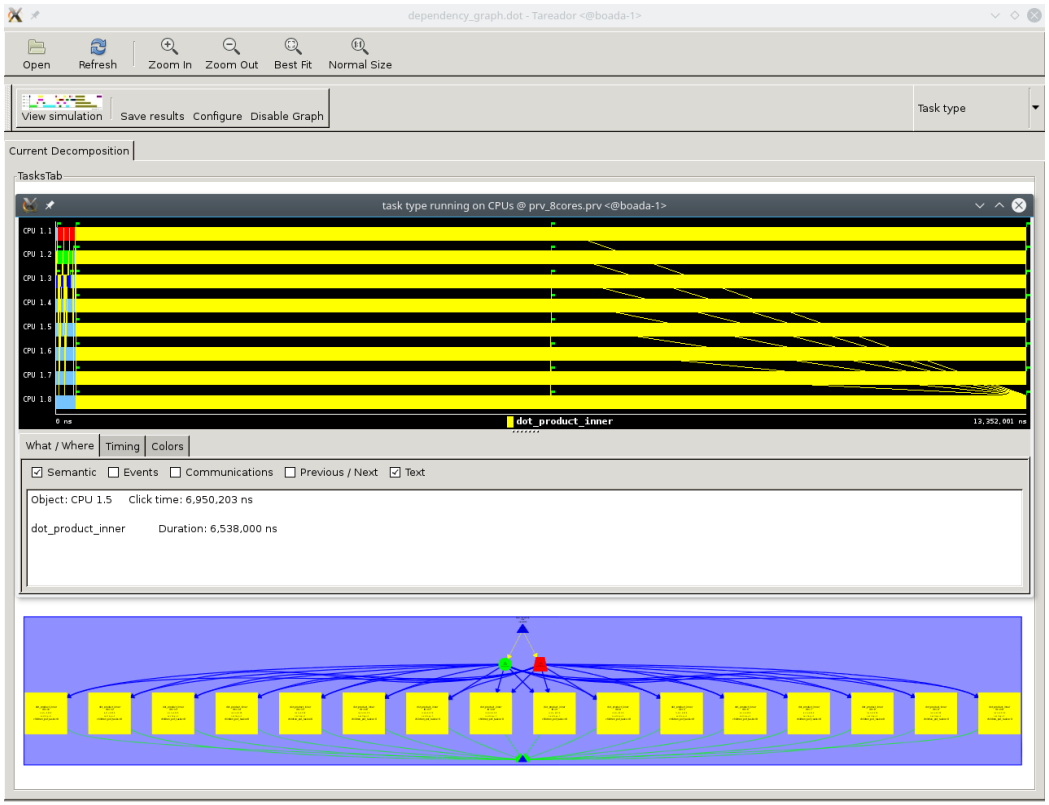


Figura 5. Dot_product task decomposition and timeline

En aquest graf, els tres polígons per entendre'l són el cercle verd, el trapezi vermell i els cuadrats grocs.

El cercle verd i el trapezi Vermell són les tasques *init A* i *init B* respectivament. Són unes tasques que inicialitzen els vectors A i B.

Els quadrats grocs representen la tasca *dot_product_inner*. Aquesta tasca es troba dins de la funció *dot_product* composa de la crida d'una funció i d'un accés a memoria. Però, el més important és veure i entendre les dependències. Per poder executar *dot_product_inner*, s'haurà d'haver acabat l'execució de *init A* i *init B*.

Ell graf també ens mostra el temps d'execució de cada tasca en relació a les altres. Clarament cadascuna de les subtasques (que n'hi ha 16) de *dot_product_inner*, triga molt més temps en executar-se que la resta. Per tant ens podem fer una idea del speed-up que podem aconseguir utilitzant només 8 threads.

El Timeline ens dóna informació molt semblant. Podem veure una simulació de l'execució del programa amb 8 processadors diferents. Podem veure com dos processadors executen *init A* i *init B* mentre els altres no fan res, i que quan aquestos acaben, tots 8 comencen *dot_product_inner*. Cadascun fa 2 execucions (separades per la bandera verda).

Analysis of task decompositions

6. Complete the following table for the initial and different versions generated for 3dfft seq.c, briefly commenting the evolution of the metrics with the different versions.

Version	T_1 (ns)	T_∞ (ns)	Parallelism
seq	**593.772.001**	**593.705.001**	**1.00011285**
v1	**593.772.001**	**593.705.001**	**1.00011285**
v2	**593.772.001**	**315.437.001**	**1.88237905**
v3	**593.772.001**	**108.937.001**	**5.45059985**
v4	**593.772.001**	**60.012.001**	**9.89422101**

Podem observar com a mesura que paral·lelitzem més el programa, el temps amb infinits threads (T_∞) disminueix.
El que passa amb v1, però, és que encara que tingui una part del codi paral·lelitzada (cadascuna de les funcions principals del programa és una tasca diferent), aquestes tenen dependències entre sí i per tant s'executen seqüencialment.
Veiem que a mesura que es paral·lelitza més el programa, més gran és el paral·lelisme (T_1/T_∞).

7. With the results from the parallel simulation with 2, 4, 8, 16 and 32 processors, draw the execution time and speedup plots for version v4 with respect to the sequential execution (that you can estimate from the simulation of the initial task decomposition that we provided in 3dfft seq.c, using just 1 processor). Briefly comment the scalability behaviour shown on these two plots.

A la Figura 6, observem com el temps va disminuint a mesura que augmentem el nombre de processadors. Això es degut a que el programa està internament descompost en tasques per tal de afavorir el paral·lelisme.
Veiem també, però, que els temps del programa amb 16 i amb 32 processadors és el mateix (també pasarà amb més de 32). És degut a que cada tasca és una iteració d'un bucle de mida N, on aquesta N està inicialitzada (arxiu *constants.h*) a mida 10. Per tant, podem deduir que a partir de 10 processadors, el temps serà constant.

Respecte a la Figura 7, podem veure gairebé el mateix. El speed-up (T_{seq}/T_n) augmenta a mesura que augmenta el nombre de processadors. Pel que hem dit abans, creiem que el speed-up es farà constant a partir de #processadors = 10.

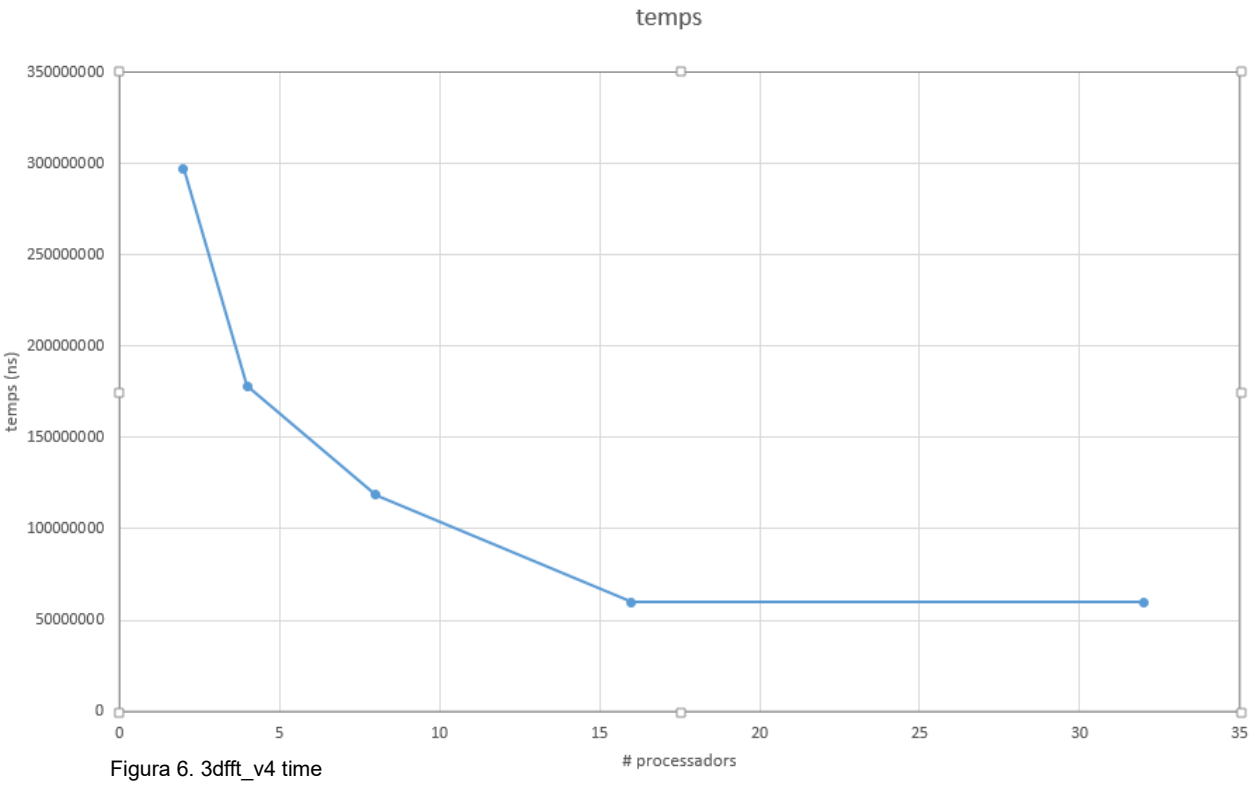


Figura 6. 3dfft_v4 time

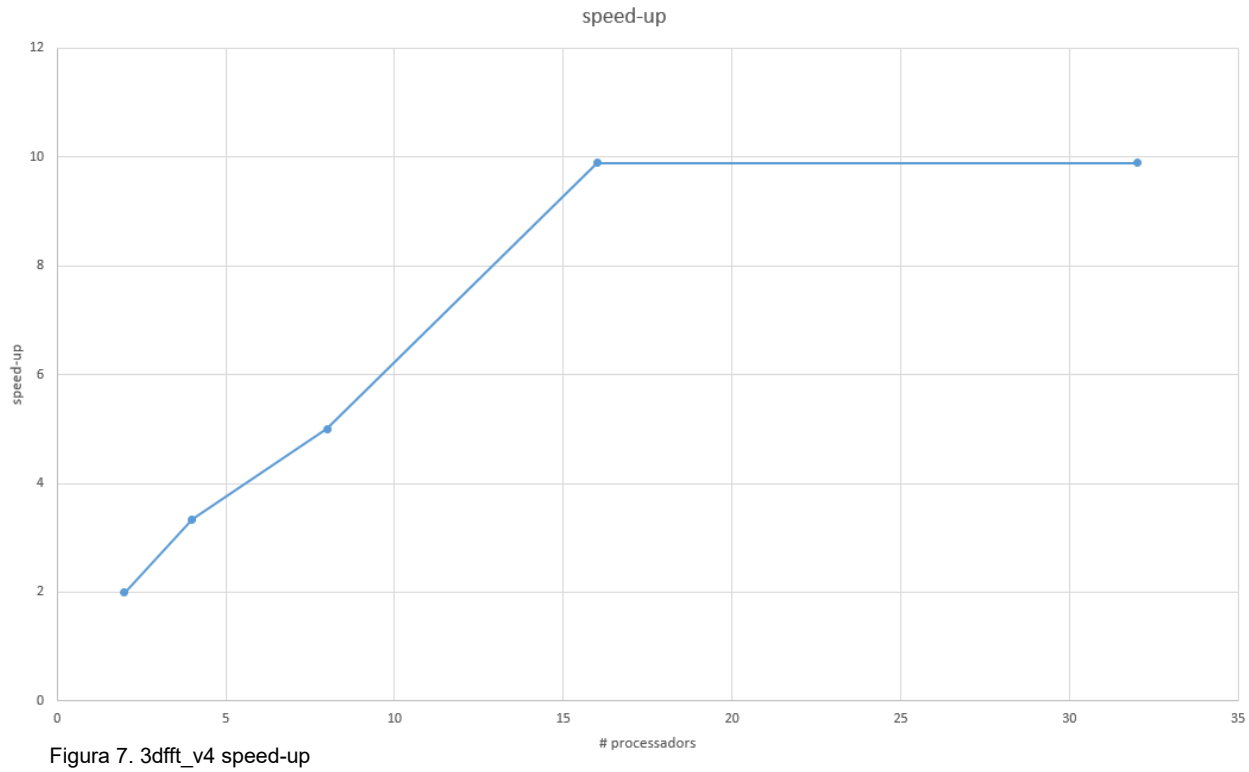


Figura 7. 3dfft_v4 speed-up

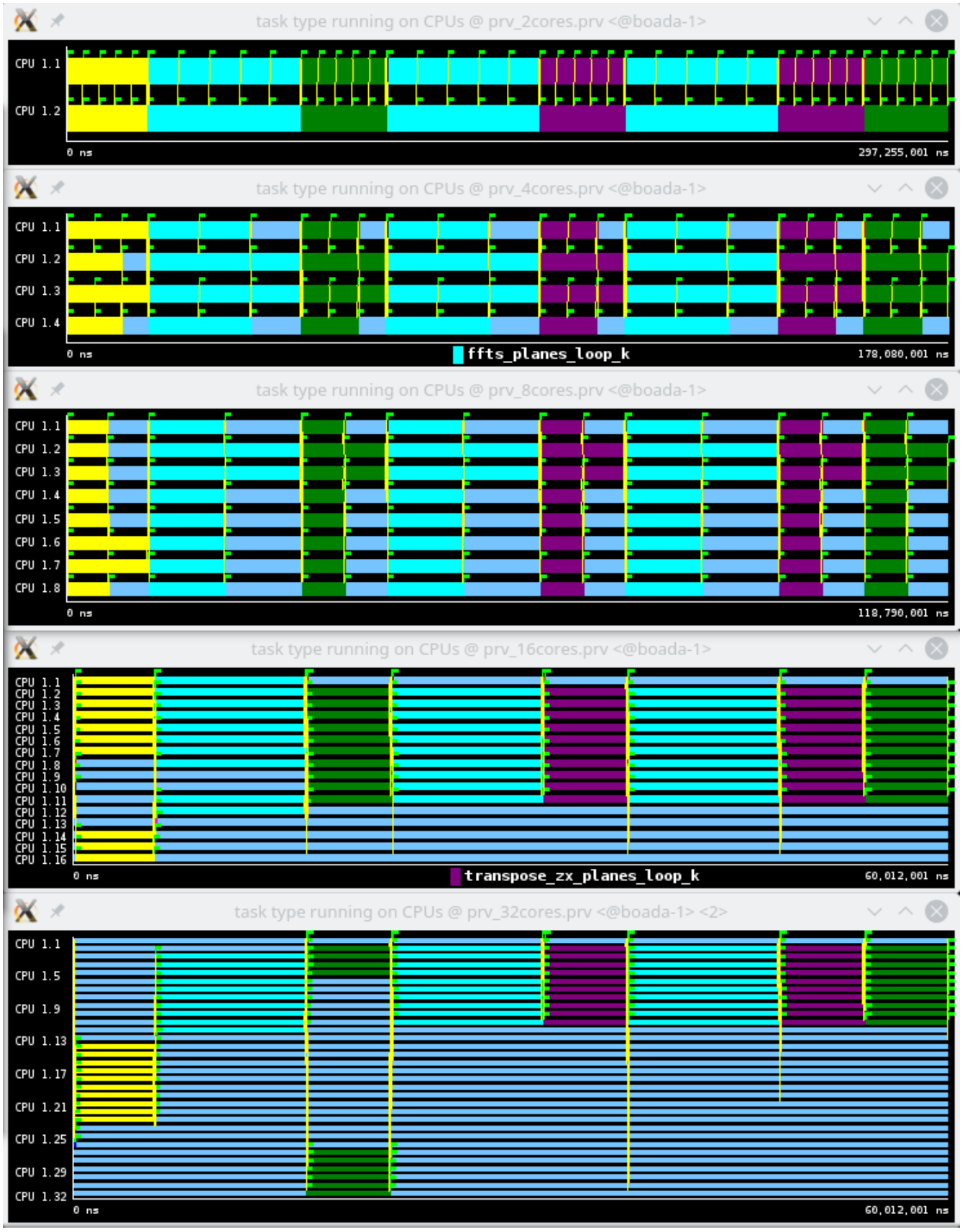


Figura 8. 3dfft_v4 with 2, 4, 8, 16 and 32 cores

Tracing sequential and parallel executions

8. From the instrumented version of pi seq.c, and using the appropriate Paraver configuration file, obtain the value of the parallel fraction φ for this program when executed with 100.000.000 iterations, showing the steps you followed to obtain it. Clearly indicate which Paraver configuration file(s) did you use.

La fracció de paral·lelisme (φ) de *pi_seq_i.c* és al voltant de 75% (74.82%).

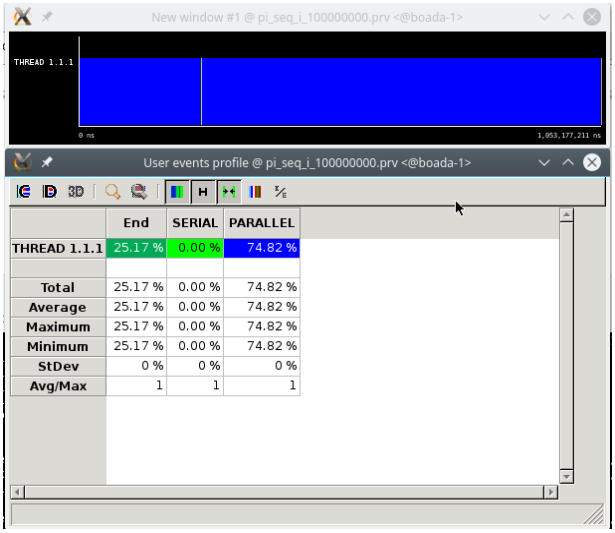
Es pot obtenir fàcilment de dues formes diferents.

Per les dues, necessitem l'API *Extrae* i *Paraver*. El primer que s'ha de fer és afegir a la línea de compilació de l'arxiu, les instruccions necessàries per compilar *Extrae (-I$(EXTRAE HOME)/include* i *-L$(EXTRAE HOME)/lib -lomptrace*).
Segon, executem el script (*submit-seq-i.sh*) que en ve donat a la cua (*qsub -I execution nom_script*), els quals, a més de definer les variables com el nombre d'iteracions, també conté les instruccions per obtenir fitxers amb la información de l'execució (.prv, .pcf i .row).

Una vegada està executat i ens ha donat aquests arxius que ens interessen (a més del .o i el .e), obrim el .prv amb *Paraver* amb la instrucció wxparaver *nom_arxiu.prv.*

Un cop obert, pel primer mètode només necessitem obrir l'arxiu de configuració */cfgs/user/APP_userevents_profile.cfg* prement el botó *Load Configuration File.*
Veurem una pantalla com la Figura 9, on ens dirà el % de programa paral·lelitzable.

L'altre mètode consisteix en obrir un altre arxiu de configuració (*/cfgs/user/APP_userevents.cfg*), mostrat a la Figura 10, on podem veure el temps del programa que és paral·lelitzable fent doble click. També podem veure el temps total avall a la dreta. Així, fent una divisió obtenim també el % de temps.



	End	SERIAL	PARALLEL
THREAD 1.1.1	25.17 %	0.00 %	74.82 %
Total	25.17 %	0.00 %	74.82 %
Average	25.17 %	0.00 %	74.82 %
Maximum	25.17 %	0.00 %	74.82 %
Minimum	25.17 %	0.00 %	74.82 %
StDev	0 %	0 %	0 %
Avg/Max	1	1	1

Figura 9: userevents_profile

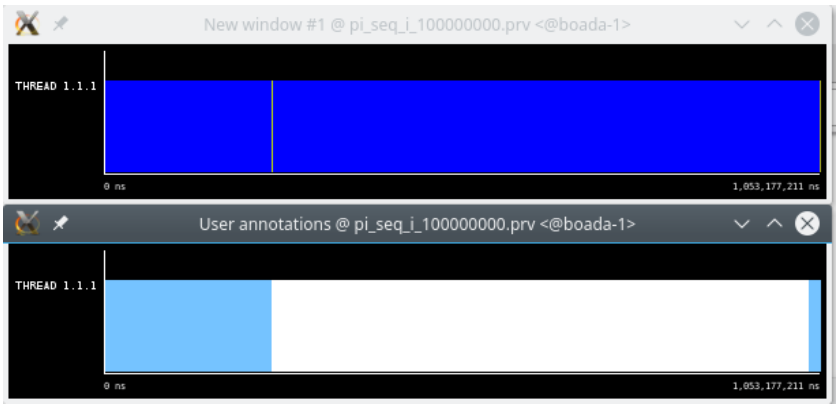


Figura 10: userevents

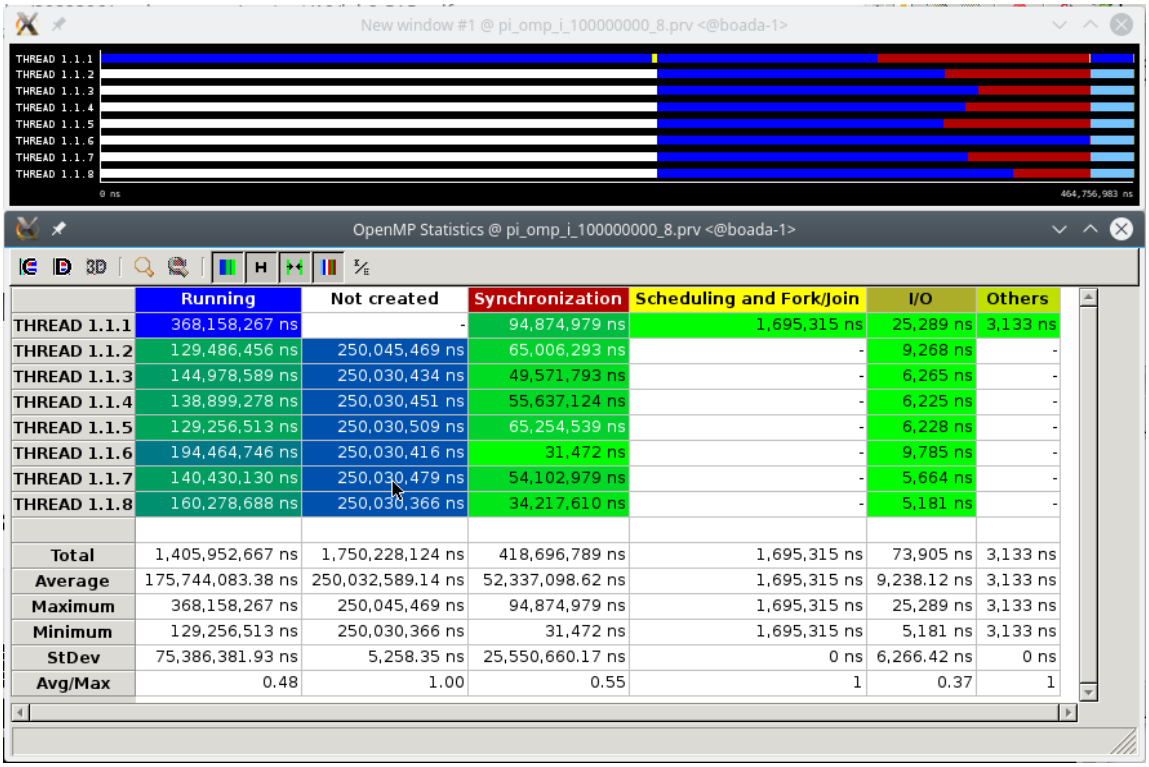
9. From the instrumented version of pi omp.c, and using the appropriate Paraver configuration file, show a profile of the % of time spent in the different OpenMP states when using 8 threads and for 100.000.000 iterations. Clearly indicate which Paraver configuration file(s) did you use and your own conclusions from that profile.

Per obtenir la taula de la Figura 11, s'han fet servir el mateixos passos que abans, però canviant l'arxiu de *pi_seq_i.c* per *pi_omp_i.c*. Aquest cop, l'arxiu de configuració ha estat /cfgs/OpenMP/OMP_state_profile.cfg.
Un cop tenim aquesta taula, si volem veure els percentatges en comptes dels valors absoluts, haurem de fer click en el botó de la Figura 12, a la finestra del Paraver, i després on diu *Statistic*, premer en *Time* i seleccionar *Percentage*. Veurem d'aquesta manera una taula com la de la Figura 13.

Les conclusions que podem extreure d'aquesta taula són diverses. La primera és que el programa no està paral·lelitzat tot el possible, ja que el thread principal està en execució més temps que la resta. En concret, la resta només "existeixen" el 56% del temps. És a dir, que el primer 46% de programa és seqüencial.

La segona conclusió és que el temps que triga el programa en la sincronització és molt gran: entre un 7 i un 20% del temps total de cada thread sense contar un d'ells. Això vol dir que no totes les tasques triguen el mateix i aquest és un temps molt valuós que perdem.

Per tant, les millores possibles són: millorar la paral·lelització de la primera meitat del programa i dividir millor les tasques amb dependències per no haver de fer esperar a altres threads.



	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	368,158,267 ns	-	94,874,979 ns	1,695,315 ns	25,289 ns	3,133 ns
THREAD 1.1.2	129,486,456 ns	250,045,469 ns	65,006,293 ns	-	9,268 ns	-
THREAD 1.1.3	144,978,589 ns	250,030,434 ns	49,571,793 ns	-	6,265 ns	-
THREAD 1.1.4	138,899,278 ns	250,030,451 ns	55,637,124 ns	-	6,225 ns	-
THREAD 1.1.5	129,256,513 ns	250,030,509 ns	65,254,539 ns	-	6,228 ns	-
THREAD 1.1.6	194,464,746 ns	250,030,416 ns	31,472 ns	-	9,785 ns	-
THREAD 1.1.7	140,430,130 ns	250,030,479 ns	54,102,979 ns	-	5,664 ns	-
THREAD 1.1.8	160,278,688 ns	250,030,366 ns	34,217,610 ns	-	5,181 ns	-
Total	1,405,952,667 ns	1,750,228,124 ns	418,696,789 ns	1,695,315 ns	73,905 ns	3,133 ns
Average	175,744,083.38 ns	250,032,589.14 ns	52,337,098.62 ns	1,695,315 ns	9,238.12 ns	3,133 ns
Maximum	368,158,267 ns	250,045,469 ns	94,874,979 ns	1,695,315 ns	25,289 ns	3,133 ns
Minimum	129,256,513 ns	250,030,366 ns	31,472 ns	1,695,315 ns	5,181 ns	3,133 ns
StDev	75,386,381.93 ns	5,258.35 ns	25,550,660.17 ns	0 ns	6,266.42 ns	0 ns
Avg/Max	0.48	1.00	0.55	1	0.37	1

Figura 11. OMP statistics for pi_omp_i (values)

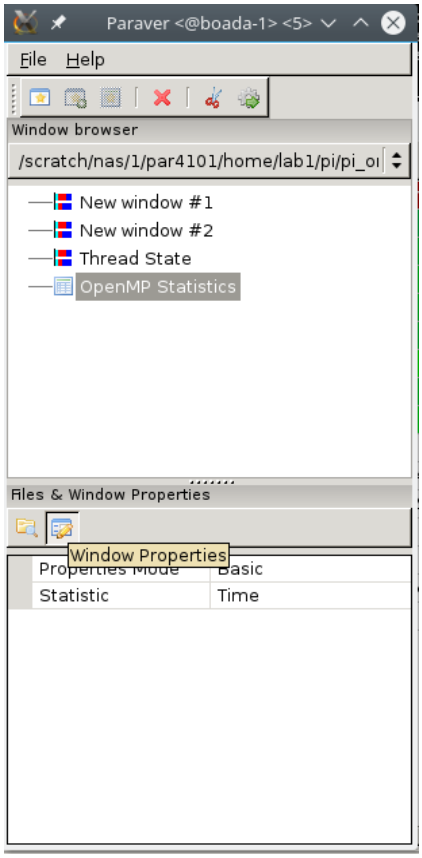


Figura 12. Paraver interface

OpenMP Statistics @ pi_omp_i_100000000_8.prv <@boada-1>

	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	79.22 %	-	20.41 %	0.36 %	0.01 %	0.00 %
THREAD 1.1.2	29.13 %	56.25 %	14.62 %	-	0.00 %	-
THREAD 1.1.3	32.61 %	56.24 %	11.15 %	-	0.00 %	-
THREAD 1.1.4	31.24 %	56.24 %	12.51 %	-	0.00 %	-
THREAD 1.1.5	29.08 %	56.24 %	14.68 %	-	0.00 %	-
THREAD 1.1.6	43.75 %	56.25 %	0.01 %	-	0.00 %	-
THREAD 1.1.7	31.59 %	56.24 %	12.17 %	-	0.00 %	-
THREAD 1.1.8	36.06 %	56.25 %	7.70 %	-	0.00 %	-
Total	312.66 %	393.70 %	93.25 %	0.36 %	0.02 %	0.00 %
Average	39.08 %	56.24 %	11.66 %	0.36 %	0.00 %	0.00 %
Maximum	79.22 %	56.25 %	20.41 %	0.36 %	0.01 %	0.00 %
Minimum	29.08 %	56.24 %	0.01 %	0.36 %	0.00 %	0.00 %
StDev	15.81 %	0.00 %	5.56 %	0 %	0.00 %	0 %
Avg/Max	0.49	1.00	0.57	1	0.38	1

Figura 13. OMP statistics for pi_omp_i (percentage)