**Lliurament**

**Lab 1**

Jacobo Moral

Carles Pàmies

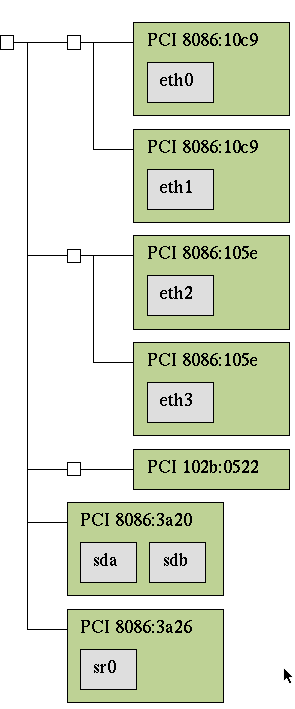
Grup 41 – Par4101

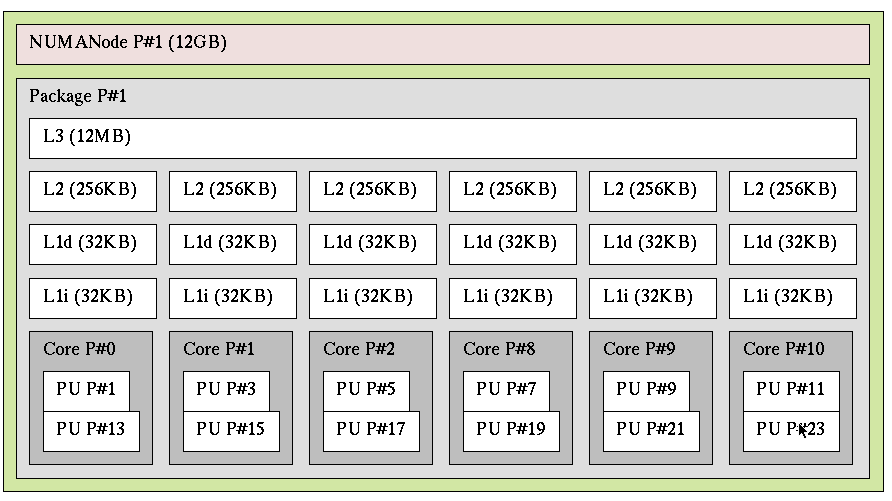
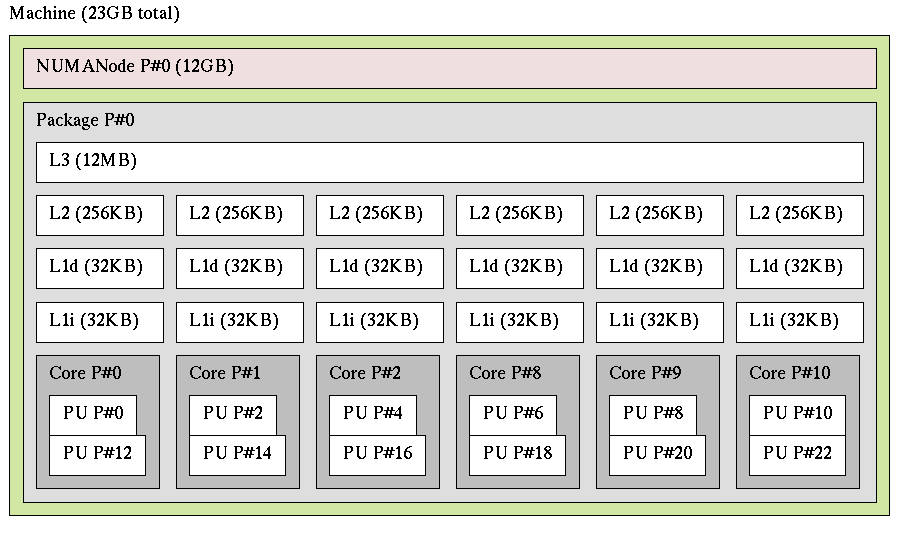
04/10/17

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).**





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

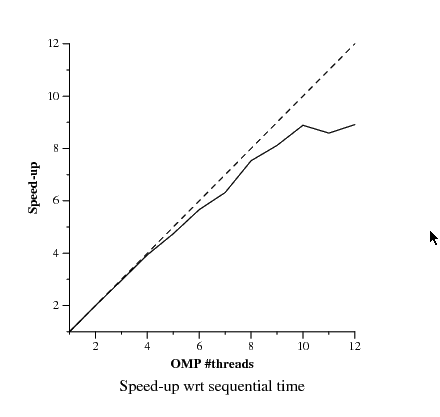
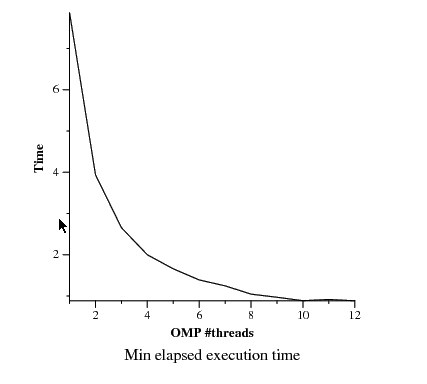
*> ps2pdf nom\_del\_arxiu.ps*

# converteix l’arxiu que conté els gràfics(.ps) a pdf

*>display nom\_del\_arxiu.pdf*

*#* obre l’arxiu pdf

*Strong scalability*

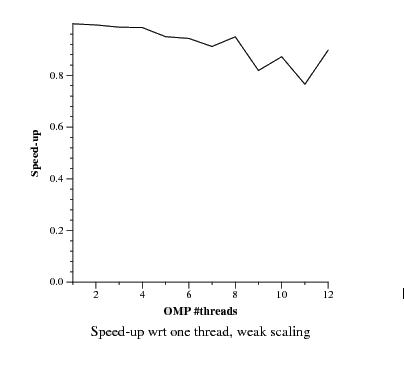
**

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 (segona figura). Això és degut a que reparteix el mateix programa en diferents threads i, per tant, el temps que trigarà en executar-se serà aproximadament tn = to/n, amb n = nombre de threads i to 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 (primera figura), 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 = to/tn. Per tant, té sentit que augmenti proporcionalment al nombre de threads. Substituint «tn = to/n» a la fòrmula del speed-up:

Speed-up = to/(to/n) = n. Això indica que la gràfica hauria de ser lineal i proporcional a n (nombre de threads), com hem dit abans.

*Weak scalability*

**

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, to ≈ tn. Alhesores, speed-up = to/to = 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*

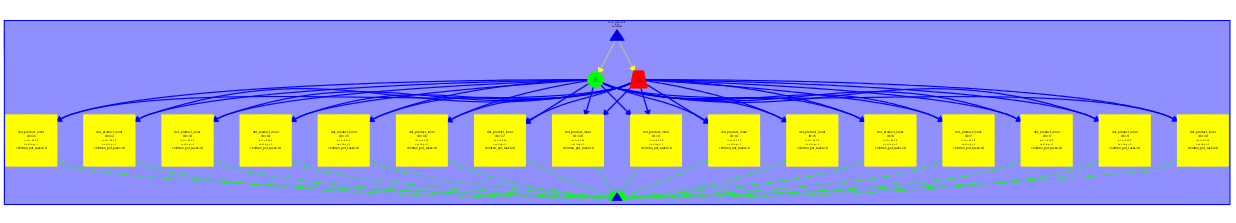
*}   
 }*

1Tot 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.**

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.

FALTA EXECUTION TIMELINE

**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** | **T1** | **T∞** | **Parallelism** |
| seq |  |  |  |
| **v1** |  |  |  |
| **v2** |  |  |  |
| **v3** |  |  |  |
| **v4** |  |  |  |

**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.**

**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*).

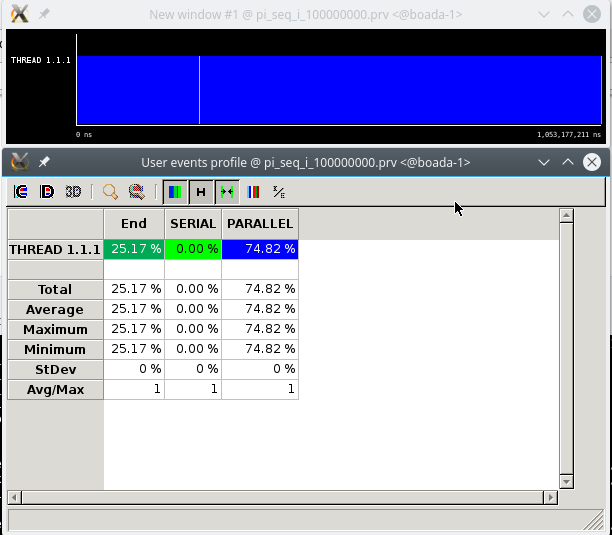
Segon, executem el script (*submit-seq-i.sh*) que en ve donat a la cua (*qsub -l execution nom\_script*), els quals, a més de definer les variables com el nombre d’iteracions, també conté les instruccions per obtenir fitxers amb la información de l’execució (.prv, .pcf i .row).

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 1, 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 2, 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.



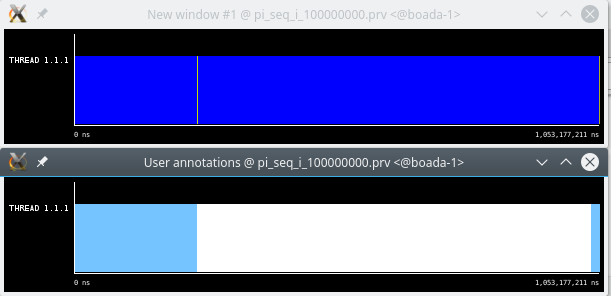


Figura 1: userevents\_profile

Figura 1: userevents\_profile

Figura 2: 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 aquesta pantalla (Figura 3), 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.

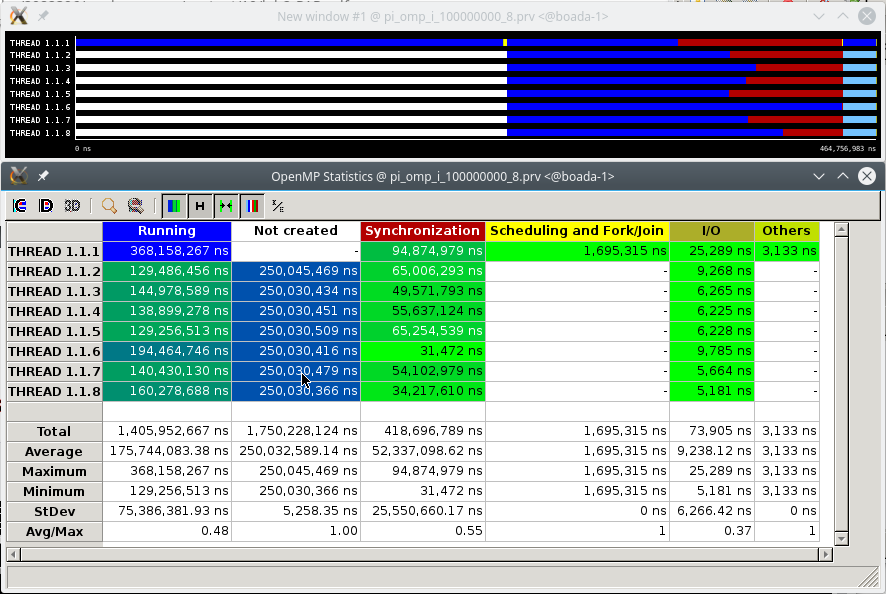
****Les conclusions que podem extreure són

Figura 3