

# PACO Lab2 - Informe

Ixent Cornella, Arnau Roca (PACO1201)

## SESSIÓ 1

### 1. 1.hello.c

- a. How many times will you see the "Hello world!" message if the program is executed with `./1.hello`?

Si executem d'aquesta manera la comanda, podem veure el missatge dues vegades.

- b. Without changing the program, how to make it to print 4 times the "Hello World!" message?

Si posem davant de la comanda `OMP_NUM_THREADS=4` podem veure 4 vegades el missatge, i en general el missatge es imprimirà  $n$  vegades.

### 2. 2.hello.c:

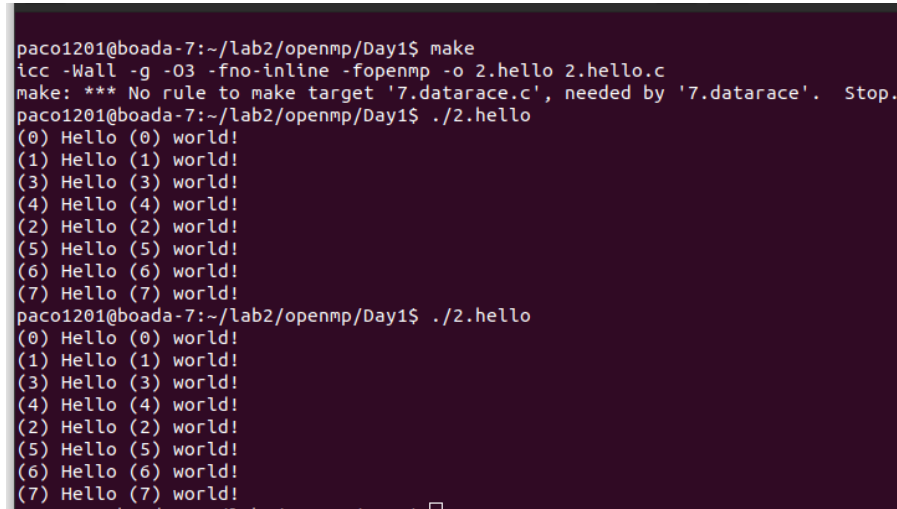
- a. Is the execution of the program correct? (i.e., prints a sequence of "(Thid) Hello (Thid) world!" being Thid the thread identifier). If not, add a data sharing clause to make it correct?

No, no és correcte, surt d'aquesta manera :

A terminal window with a dark background and light-colored text. The prompt is 'paco1201@boada-7:~/lab2/openmp/Day1\$'. The command executed is './2.hello'. The output consists of several lines of text: '(0) Hello (1) Hello (1) world!', '(1) world!', '(3) Hello (1) world!', '(4) Hello (4) world!', '(5) Hello (5) world!', '(6) Hello (6) world!', '(2) Hello (2) world!', and '(7) Hello (7) world!'. The numbers in parentheses represent thread IDs, and the output is not in a consistent order, indicating a race condition or lack of synchronization.

Figura 1: Resultat d'executar 2.hello al principi.

Per arreglar-ho hauriem de fer la variable "id" privada, i després afegir un "critical" per evitar que no s'imprimeixin en desordre (#pragma omp critical). Quedaria així:



```
paco1201@boada-7:~/lab2/openmp/Day1$ make
icc -Wall -g -O3 -fno-inline -fopenmp -o 2.hello 2.hello.c
make: *** No rule to make target '7.datarace.c', needed by '7.datarace'. Stop.
paco1201@boada-7:~/lab2/openmp/Day1$ ./2.hello
(0) Hello (0) world!
(1) Hello (1) world!
(3) Hello (3) world!
(4) Hello (4) world!
(2) Hello (2) world!
(5) Hello (5) world!
(6) Hello (6) world!
(7) Hello (7) world!
paco1201@boada-7:~/lab2/openmp/Day1$ ./2.hello
(0) Hello (0) world!
(1) Hello (1) world!
(3) Hello (3) world!
(4) Hello (4) world!
(2) Hello (2) world!
(5) Hello (5) world!
(6) Hello (6) world!
(7) Hello (7) world!
```

*Figura 2: Resultat d'executar 2.hello un cop modificati.*

- b. Are the lines always printed in the same order? Why the messages sometimes appear inter-mixed? (Execute several times in order to see this).**

No, com podem veure a la figura 2, surt el Hello World corresponent a cada thread, però el ordre d'execució dels threads no és sempre el mateix, cosa que fa que no estigui ben ordenat. Això és degut a que tot i que afegim un critical i fem la variable privada, no tenim manera de controlar (ara per ara) quin thread s'executarà primer.

### **3. 3.how many.c: Assuming the OMP NUM THREADS variable is set to 8 with "OMP NUM THREADS=8 ./3.how many"**

- a. What does omp\_get\_num\_threads return when invoked outside and inside a parallel region?**

Quan està fora de la regió paralela és 1, cosa que té sentit ja que no hi ha paral·lelisme, només hi ha un thread. Al estar a dins, la cosa canvia, i depèn de varies coses: Si executem la funció `get_num_threads()` al principi i sense cap variable que li afecti, imprimirà el número de threads

que li hem assignat, que és 8 (hem d'haver activat `pragma omp parallel`). D'altra banda, si modifiquem el número de threads dins d'un programa, (amb la comanda `omp_set_num_threads(x)`, o afegint `num_threads(x)` després de la comanda `parallel`), tindrem aquest número de threads modificat imprimit.

**b. Indicate the two alternatives to supersede the number of threads that is specified by the `OMP_NUM_THREADS` environment variable.**

Amb la comanda `omp_set_num_threads(x)`, o afegint `num_threads(x)` després de la comanda `#pragma omp parallel`. Aquestes dos comandes serveixen per canviar el número de threads a usar.

**c. Which is the lifespan for each way of defining the number of threads to be used?**

L'esperança de vida de quan definim el número de threads, és la següent línia si no afegim corxets (`{ }`), o si afegim corxets, fins que els tanquin (`}`).

#### 4. 4.data sharing.c

**a. Which is the value of variable `x` after the execution of each parallel region with different data-sharing attribute (`shared`, `private`, `firstprivate` and `reduction`)? Is that the value you would expect? (Execute several times if necessary)**

El valor de `shared` cada cop no és privat i per tant no pots controlar el valor (ja que demana el número del thread, no el total i li suma 0, per tant sempre serà diferent), el `private` i `firstprivate` sempre és 5 (ja que és privat i el podem controlar) i el `reduction` és 501 (ja que és la suma de tots els números de threads:  $1+2+3+4+\dots$  que és resumeix amb la fórmula  $(32*(32-1))/2$  i finalment li suma 5, la qual cosa ens dona 501).

#### 5.data race.c

**a. Should this program always return a correct result? Reason either your positive or negative answer.**

No, no s'està executant correctament, ja que tenim un error de que maxvalue no és correcte, ja que ha de tenir un valor de 15 però no sempre el té, ja que hi ha el paral·lisme que pot afectar el valor final. Si veiem el contingut del programa, podem observar on és el problema:

```
#define N 1 << 20
int vector[N]={0, 0, 0, 1, 2, 3, 4, 5, 6, 7, 15, 14, 13, 12, 11, 10, 9, 8, 15, 15};

int main()
{
    int i, maxvalue=3;

    omp_set_num_threads(8);
    #pragma omp parallel private(i)
    {
        int id = omp_get_thread_num();
        int howmany = omp_get_num_threads();

        for (i=id; i < N; i+=howmany) {
            if (vector[i] > maxvalue)
            {
                sleep(1); // this is just to force problems
                maxvalue = vector[i];
            }
        }
    }

    if (maxvalue==15)
        printf("Program executed correctly - maxvalue=%d found\n", maxvalue);
    else printf("Sorry, something went wrong - incorrect maxvalue=%d found\n", maxvalue);

    return 0;
}
```

Figura 3: Codi de 5.datarace.c

maxvalue és canviat constantment per cada thread, i al final, l'últim en modificar-ho és el que editarà la variable amb el valor final.

**b. Propose two alternative solutions to make it correct, without changing the structure of the code (just add directives or clauses). Explain why they make the execution correct.**

Una manera de fer que funcioni es forçant els threads a esperar-se abans del for, i ho podem fer amb un critical. Això forçarà a que s'esperin abans d'executar el codi del bucle i el valor final de maxvalue sempre sigui 15. L'altra manera és crear una variable auxiliar no privada, i fer maxvalue privada. Abans de sortir de la directiva parallel, assignem el resultat de maxvalue a aux i ja tenim el valor maxmim a maxvalue.

c. Write an alternative distribution of iterations to implicit tasks (threads) so that each of them executes only one block of consecutive iterations (i.e.  $N$  divided by the number of threads).

```

/* Q2: Propose two alternative solutions to make it correct, */
/* without changing the structure of the code (just add */
/* directives or clauses). Explain why they make the */
/* execution correct. */
/* Q3: Write an alternative distribution of iterations to */
/* implicit tasks (threads) so that each of them executes */
/* only one block of consecutive iterations (i.e. N */
/* divided by the number of threads. */

#define N 1 << 20
int vector[N]={0, 0, 0, 1, 2, 3, 4, 5, 6, 7, 15, 14, 13, 12, 11, 10, 9, 8, 15, 15};

int main()
{
    int i, maxvalue=3;

    omp_set_num_threads(8);
    #pragma omp parallel private(i)
    {
        int id = omp_get_thread_num();
        int howmany = omp_get_num_threads();

        for (i=id*(N/howmany); i < (id+1)*(N/howmany); i++) {
            if (vector[i] > maxvalue)
            {
                sleep(1); // this is just to force problems
                maxvalue = vector[i];
            }
        }
    }

    if (maxvalue==15)

```

Figura 4: Codi de amb iteracions consecutives

## 6.datarace.c

a. Should this program always return a correct result? Reason either your positive or negative answer.

No, no s'està executant correctament, ens dona un error alguns cops, i de tant en quan ens dona el valor correcte. El motiu és el mateix de l'apartat anterior, una datarace per accedir a countmax.

b. Propose two alternative solutions to make it correct, without changing the structure of the program (just using directives or clauses) and never making use of `critical`. Explain why they make the execution correct.

El reduction és una opció, només cal ficar “reduction (+:countmax)” i ja és sumarà countmax de forma coherent, evitant dataraces a la variable.

Per altra banda també podem usar el atomic, degut a que l'operació és de només una variable (countmax++), fa el mateix que el critical i s'hauria de posar així: “pragma omp atomic” just a la línia damunt del countmax++.

## SESSIÓ 2

### 1.single.c0

**a. What is the nowait clause doing when associated to single?**

La clàusula `nowait`, com pot dir el seu nom “no esperar”, fa que els altres threads continuïn executant la seva instrucció fins al final, sense esperar als altres. Els threads executen el bucle, però les iteracions es fan de un en un, un sol cop quan els threads estan disponibles.

**b. Then, can you explain why all threads contribute to the execution of the multiple instances of single? Why those instances appear to be executed in bursts?**

Ja que la clàusula `#pragma omp parallel` es crida just abans del bucle, tots els threads disponibles l'executaran, però `#pragma omp single nowait` fa que totes les instruccions de dins del bucle només s'executin un cop, per cada thread disponible, llavors tindrem com a resultat que cada iteració serà executada per un thread únicament.

D'altra banda, els 4 threads han d'esperar 1 segon, ja que hi ha la comanda `sleep(1)`, explicant així el retràs (cada iteració és executada pels threads disponibles i després tots esperen un segon per tornar a executar iteracions posteriors).

### 2.fibtasks.c

**a. Why all tasks are created and executed by the same thread? In other words, why the program is not executing in parallel?**

Això és degut a que no hi ha paral·lelisme al codi! No hi ha cap clàusula de l'estil `#pragma omp parallel`, per tant, no hi haurà paral·lelisme si no afegim aquesta instrucció.

**b. Modify the code so that tasks are executed in parallel and each iteration of the while loop is executed only once.**

Hem afegit l'instrucció `#pragma omp parallel` al primer `while`, i després `#pragma omp single`. Amb això, assignem cada tasca a un thread, però només la fa un thread l'assignació. El resultat és que ara cada thread pot executar alguna tasca computacional.

**c. What is the firstprivate(p) clause doing? Comment it and execute again. What is happening with the execution? Why?**

El que fa és especificar que cada thread ha de tenir la seva instància de la variable ja inicialitzada (anteriorment al codi) , alhora, el que aconseguim és que cada thread agafi el valor correcte i l'utilitzi de forma privada, per evitar errors amb els punters. Si traiem la clàusula, el que passa és que al compartir la variable entre els threads, el punter pot arribar a ser null, ja que està sent constantment canviat, i per tant, resultat: core dumped (accés a dades d'un punter null).

### **3.taskloop.c**

**a. Which iterations of the loops are executed by each thread for each task grain size or num tasks specified?**

Al tenir 12 iteracions, grainsize(4) ha de repartir, com a mínim, 4 tasques a un thread aleatori, per tant, s'assignen 4 iteracions, i després 4 més (sense tenir en compte si un thread ja les ha fet), i les últimes 4 al final. El resultat és que les iteracions s'executen en ràfegues de 4 iteracions per thread.

En el num tasks, podem veure que el comportament és l'oposat, és a dir, divideix les 12 iteracions, entre els 4 threads (en general, amb el nombre que li hem indicat). Així ens quedaria un repartiment de 4 tasques i 3 iteracions per tasca. Cada 3 iteracions les executa el mateix thread i després canvia.

En definitiva, grainsize(n) divideix les iteracions, i numtasks(n) divideix les tasques primer.

**b. Change the value for grainsize and num tasks to 5. How many iterations is now each thread executing? How is the number of iterations decided in each case?**

En aquest cas el grainsize ens farà 6 iteracions per cada thread, això és degut a que el grainsize té com a número mínim el número que li fiquem. El número de iteracions serà més gran en cas que el nombre que haguem posat no doni residu 0, llavors buscarà un nombre que sí que doni residu 0 (aquest augment pot ser fins no més del doble del nombre proposat).

Pel que fa a num\_tasks, aquesta clàusula intentarà repartir les 12 iteracions entre 5, però al quedar un número fraccional, llavors repartirà les iteracions



de forma desigual. En nostre cas,  $12/5 = 2.4$ , llavors les primeres 6 iteracions seran executades per 2 threads, i la resta es dividiran en 2 iteracions per thread.

**c. Can grainsize and num tasks be used at the same time in the same loop?**

No. No es pot tenir les dos clàusules a la mateixa línia, ja que confondria al procesador a l'hora de repartir les tasques, no es pot fer de les dues maneres alhora.

**d. What is happening with the execution of tasks if the nogroup clause is uncommented in the first loop? Why?**

Si fiquem nogroup, estem fent que no hi hagi cap regió de tasques, el que vol dir és que cada thread anirà fent les tasques que trobi, sense necessàriament fer les ràfegues que fa amb group.

#### 4.reduction.c

a) Complete the parallelisation of the program so that the correct value for variable sum is returned in each printf statement. Note: in each part of the 3 parts of the program, all tasks generated should potentially execute in parallel.

```
omp_set_num_threads(4);
#pragma omp parallel
#pragma omp single
{
    // Part I
    #pragma omp taskgroup task_reduction(+: sum)
    {
        for (i=0; i< SIZE; i++)
            #pragma omp task firstprivate(i) in_reduction(+: sum)
            sum += X[i];
    }

    printf("Value of sum after reduction in tasks = %d\n", sum);

    // Part II
    #pragma omp taskloop grainsize(BS) firstprivate(sum)
    for (i=0; i< SIZE; i++)
        sum += X[i];

    printf("Value of sum after reduction in taskloop = %d\n", sum);
}
```

```

// Part III
#pragma omp taskgroup task_reduction(+:sum)
{
    #pragma omp taskloop grainsize(BS) firstprivate(sum)
    for (i=0; i< SIZE/2; i++)
        sum += X[i];

    #pragma omp taskloop grainsize(BS) firstprivate(sum)
    for (i=SIZE/2; i< SIZE; i++)
        sum += X[i];
}
printf("Value of sum after reduction in combined task and taskloop = %d\n", sum);
}

return 0;

```

Figura 5: Codi resultant per a que funcioni correctament el programa

## 5.synchtasks.c

### a) Draw the task dependence graph that is specified in this program

A, B and C are independent so they run in parallel. Task D depends on task A and B to start executing so it will start when A and B are done. Task E depends on D and C, so it will need to wait until both are completed to execute.

A, B i C són independents, i els executem en paralel. La tasca D depèn de A i B per començar, per tant quan es completin A i B, començarà D. La tasca E depèn de D i C, llavors s'executarà quan acabin aquestes dos.

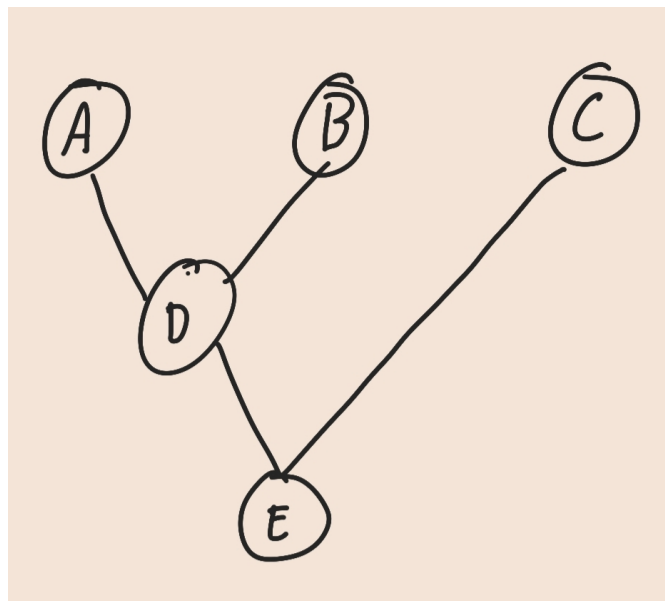


Figura 6: Graf de dependències resultant.

**b) Rewrite the program using only taskwait as task synchronisation mechanism (no depend clauses allowed), trying to achieve the same potential parallelism that was obtained when using depend.**

El que hem de fer és usar les clàusules `#pragma omp task` i `#pragma omp taskwait` per a poder reorganitzar el codi. La clau aquí és estructurar el codi de forma que les dependències es respectin, es a dir, executar les tasques A i B, després esperar a que acabin i executar la tasca D, i després ja es pot executar la tasca C, i finalment esperem a que acabin i executem la tasca final E.

```
int a, b, c, d;
int main(int argc, char *argv[]) {
    #pragma omp parallel
    #pragma omp single
    {
        printf("Creating task foo1\n");
        #pragma omp task
        foo1();
        printf("Creating task foo2\n");
        #pragma omp task
        foo2();
        printf("Creating task foo4\n");
        #pragma omp taskwait
        #pragma omp task
        foo4();
        printf("Creating task foo3\n");
        #pragma omp task
        foo3();
        printf("Creating task foo5\n");
        #pragma omp taskwait
        #pragma omp task
        foo5();
    }
    return 0;
}
```

*Figura 7: Codi per a l'execució correcta amb taskwait.*

**c) Rewrite the program using only taskgroup as task synchronisation mechanism (no depend clauses allowed), again trying to achieve the same potential parallelism that was obtained when using depend.**

Aquí lo ideal es crear dos taskgroups, un per A i B, i l'altre per D i C. D'aquesta manera evitem dependències, i executarem les tasques en l'ordre indicat.

```
int a, b, c, d;
int main(int argc, char *argv[]) {
    #pragma omp parallel
    #pragma omp single
    {
        #pragma omp taskgroup
        {
            printf("Creating task foo1\n");
            #pragma omp task
            foo1();
            printf("Creating task foo2\n");
            #pragma omp task
            foo2();
        }
        #pragma omp taskgroup
        {
            printf("Creating task foo4\n");
            #pragma omp task
            foo4();
            printf("Creating task foo3\n");
            #pragma omp task
            foo3();
        }
        printf("Creating task foo5\n");
        #pragma omp task
        foo5();
    }
    return 0;
}
```

*Figura 8: Codi per a l'execució correcta amb taskwait.*

**Please explain in this section of your deliverable the main results obtained and your conclusions in terms of overheads for parallel, task and the different synchronisation mechanisms. Include any tables/plots that support your conclusions.**

Hem observat que existeixen múltiples maneres d'assignar les tasques a cada thread, que les tasques es poden repartir tenint en compte el número de tasques que volem o com volem el gra, i també que l'overhead juga un rol important a l'hora de millor l'eficiència d'un programa.

En primer lloc, podem assignar tasques a un thread de diverses maneres, ja sigui utilitzant taskgroup, task o dependencies. Això fa que es puguin repartir bé les tasques entre cada thread, i millor per tant el rendiment. També evitem possibles errors amb dependències de dades, o d'altres problemes de sincronització, sempre que el codi estigui ben estructurat.

Seguidament, podem repartir aquestes tasques de dues maneres diferents: tenint en compte el gra o el número de tasques. Al ajustar el gra de les tasques, estem fent que el nombre total d'iteracions que fa un thread s'ajusti al número que li indiquem, decidint així quantes tasques fa un thread, i per tant, modificant el gra. D'altra banda, també es pot repartir les tasques de forma en que cada thread tingui n tasques, cada mètode es pot fer servir segons el context.

Finalment, l'overhead és un tema seriós. Un programa que a simple vista pot sembla molt eficient per l'alt paral·lelisme, pot ser molt lent degut a l'overhead, ja que és important ajustar les tasques de forma en que es reparteixin equitativament entre threads, però sense exagerar per evitar que el temps de crear una tasca sigui més gran que fer-la.