PAR Laboratory Deliverable
Lab 2: Brief tutorial on OpenMP programming model

Víctor Asenjo Carvajal - par1302
Ferran De La Varga Antoja - par1305

March 2022
Semester of Spring 2021-2022

Índex

Session 1

1 Introduction: connecting and unzipping files

Comencem copiant i descomprimint els arxius necessaris per aquestes sessions del Laboratori 2. Per això ens connectarem com vam fer al Laboratori 1:

```
ssh -X par1302@boada.ac.upc.edu
cp /scratch/nas/1/par0/sessions/lab2.tar.gz .
tar -zxvf lab2.tar.gz
```

Figure 1: Comandes de connexió, copiat d'arxius i descompressió d'aquests respectivament

Recordem també que les execucions interactives es fan amb:

```
./run-xxxx.sh
```

Figure 2: Ordre execució interactiva

Mentres que les encuades:

```
sbatch [-p partition] ./submit-xxxx.sh
```

Figure 3: Ordre execució encuada

En aquestes dues sessions de Laboratori 2 coneixerem les moltes opcions que ofereix la biblioteca *OpenMP*.

En primer lloc, el **dia 1**, coneixerem els 4 funcionaments principals de la biblioteca, que ens ajudarà a paral·lelitzar els nostres programes. Aquestes operacions paral·leles d'*OpenMP* són **crítiques**, **atòmica**, **sumalocal** i de **reducció**.

Cadascun d'elles té el mateix objectiu, que és evitar els *data race*. El que fan és protegir una variable a la qual s'està accedint per un nombre diferent de fils perquè el seu valor, de no estar protegit, variaria donant un resultat erroni. La diferència entre aquestes operacions és la forma en què cadascuna d'elles protegeix la variable.

Al **dia 2** seguirem aprenent sobre les operacions *OpenMP* que són útils per als nostres programes paral·lels. Aquí, ens centrarem en la sincronització de les tasques executades per diferents fils. Aquestes operacions són **single**, **taskgroup**, **taskwait**, **task with dependence** i **taskloop**.

Finalment, també estudiarem un dels problemes més difícils del paral·lelisme: *overheads*.

2 OpenMP questionnaire

Day 1: Parallel regions and implicit tasks

`1.hello.c`

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

Ens apareix dues vegades ja que per defecte està definit que s'executi amb dos fils (threads):

```
Hello world!
Hello world!
```

Figure 4: Sortida compilació per defecte de `1.hello`

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

Ara necessitem executar el 1.hello.c amb la comanda `OMP_NUM_THREADS=4 ./1.hello`. Aquesta ordre ens permet definir una variable que es fa servir dins del codi. Per defecte, utilitza dos fils, però amb aquesta bandera, es modifica el valor de la variable en temps de compilació. Així, al lloc de 2 fils, ara hem establert el nombre de fils a 4, per la qual cosa hi haurà 4 fils executant el mateix codi. Cada fil imprimirà un "Hello World!".

```
par1302@boada-1:~/lab2/openmp/Day1$ OMP_NUM_THREADS=4 ./1.hello
Hello world!
Hello world!
Hello world!
Hello world!
```

Figure 5: Sortida de `1.hello` amb 4 threads

`2.hello.c`

1. 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?

Executem i mirem què surt:

```
par1302@boada-1:~/lab2/openmp/Day1$ ./2.hello
(0) Hello (3) Hello (3) world!
(3) world!
(2) Hello (2) world!
(6) Hello (6) world!
(1) Hello (1) world!
(5) Hello (5) world!
(4) Hello (5) world!
(7) Hello (7) world!
par1302@boada-1:~/lab2/openmp/Day1$ ./2.hello
(4) Hello (0) world!
(0) Hello (0) world!
(2) Hello (2) world!
(3) Hello (2) world!
(6) Hello (6) world!
(1) Hello (1) world!
(5) Hello (5) world!
(7) Hello (7) world!
par1302@boada-1:~/lab2/openmp/Day1$ ./2.hello
(0) Hello (3) Hello (3) world!
(3) world!
(5) Hello (5) world!
(2) Hello (5) world!
(4) Hello (1) world!
(1) Hello (1) world!
(7) Hello (7) world!
(6) Hello (6) world!
par1302@boada-1:~/lab2/openmp/Day1$ ./2.hello
(4) Hello (0) world!
(0) Hello (0) world!
(2) Hello (2) world!
(3) Hello (2) world!
(1) Hello (1) world!
(5) Hello (1) world!
(7) Hello (7) world!
(6) Hello (6) world!
```

Figure 6: Sortida `2.Hello` sense modificar

La sortida no és correcta. A algunes iteracions es repeteix una part de la frase, altres no és el mateix thread id... Aquest error es produeix perquè tenim la variable `id` compartida (recordem que les variables a regions paral·lelitzades amb OpenMP per defecte són compartides). Tots els threads estan accedint a la mateixa variable, fent que aquest valor canviï encara que el fil anterior no hagi acabat completament la seva execució. Aquest problema es coneix com a *data race*.

Per solucionar-ho podem posar-la com a privada (igual que es feia a la versió 2 de pi).

```
1  int main ()
2  {
3      int id;
4      #pragma omp parallel private(id) num_threads(8)
5      {
6          id =omp_get_thread_num();
7          printf("(%d) Hello ",id);
8          printf("(%d) world!\n",id);
9      }
10     return 0;
11 }
```

Figure 7: Codi amb private(id)

Amb aquesta modificació aconseguim que la sortida del binari quedi de la següent manera:

```
par1302@boada-1:~/lab2/openmp/Day1$ make 2.hello
icc -Wall -g -O3 -fno-inline -fopenmp -o 2.hello 2.hello.c
par1302@boada-1:~/lab2/openmp/Day1$ ./2.hello
(4) Hello (4) world!
(0) Hello (0) world!
(2) Hello (2) world!
(3) Hello (3) world!
(6) Hello (6) world!
(1) Hello (1) world!
(5) Hello (5) world!
(7) Hello (7) world!
par1302@boada-1:~/lab2/openmp/Day1$ ./2.hello
(3) Hello (3) world!
(0) Hello (0) world!
(1) Hello (1) world!
(2) Hello (2) world!
(6) Hello (6) world!
(4) Hello (4) world!
(5) Hello (5) world!
(7) Hello (7) world!
```

Figure 8: Compilació + Sortida `2.hello` amb variable `id` *private*

Com es fa amb 8 threads es printa vuit vegades. Ara amb la variable private podem veure com ja no hi ha errors de repetició.

2. Are the lines always printed in the same order? Why the messages sometimes appear intermixed? (Execute several times in order to see this).

Les línies no sempre surten en el mateix ordre; ja ho hem vist abans. Els missatges apareixen barrejats ja que cada processador està executant una tasca i potser un acaba abans que un altre. El resultat hauria estat diferent si s'hagués executat només per un *thread*, però com s'està executat per molts processadors, no podem controlar l'ordre d'acabament dels mateixos (de moment).

Això passa perquè tots els *threads* comencen junts i s'executen en paral·lel: un podria haver acabat abans que un altre però tots dos estan esperant l'anterior. Així, doncs, no podem controlar la durada dels mateixos ni quan comencen o acaben a causa del paral·lelisme.

`3.how_many.c`

1. What does `omp_get_num_threads` return when invoked outside and inside a parallel region?

Fora de la regió paral·lelitzada `omp_get_num_threads` retorna `"1 thread"` mentres que dins la regió retorna el `thread_id`.

```
par1302@boada-1:~/lab2/openmp/Day1$ OMP_NUM_THREADS=8 ./3.how_many
Starting, I'm alone ... (1 thread)
Hello world from the first parallel (8)!
Hello world from the first parallel (8)!
Hello world from the first parallel (8)!
Hello world from the first parallel (8)!
Hello world from the first parallel (8)!
Hello world from the first parallel (8)!
Hello world from the first parallel (8)!
Hello world from the first parallel (8)!
Hello world from the szecond parallel (4)!
Hello world from the szecond parallel (4)!
Hello world from the szecond parallel (4)!
Hello world from the szecond parallel (4)!
Hello world from the third parallel (8)!
Hello world from the third parallel (8)!
Hello world from the third parallel (8)!
Hello world from the third parallel (8)!
Hello world from the third parallel (8)!
Hello world from the third parallel (8)!
Hello world from the third parallel (8)!
Hello world from the third parallel (8)!
Hello world from the fourth parallel (2)!
Hello world from the fourth parallel (2)!
Hello world from the fourth parallel (3)!
Hello world from the fourth parallel (3)!
Hello world from the fourth parallel (3)!
Outside parallel, nobody else here ... (1 thread)
Hello world from the fifth parallel (4)!
Hello world from the fifth parallel (4)!
Hello world from the fifth parallel (4)!
Hello world from the fifth parallel (4)!
Hello world from the sixth  parallel (3)!
Hello world from the sixth  parallel (3)!
Hello world from the sixth  parallel (3)!
Finishing, I'm alone again ... (1 thread)
```

Figure 9: Sortida `3.how_many`

Un cop obtenim la sortida (Figura 9) podem veure que, depenent de la regió paral·lela, el nombre de sortides de "Hello world..." és diferent.

En el primer paral·lel podem observar que hi ha 8 sortides. Això passa perquè `#pragma omp parallel` s'escriu abans de `printf`, establint el nombre de *threads* a 8, que és el nombre predeterminat.

Això no passa en el segon, quart, cinquè o sisè paral·lel perquè el nombre de *threads* s'ha canviat a un nombre específic. Amb 4, 2, 4 i 3 *threads* respectivament.

Dins de la zona paral·lelitzada `omp_get_num_threads` retorna el número de threads que estan executant el codi. Fora de la zona paral·lelitzada, retorna 1 thread.

Podem observar que `omp_set_num_threads(i)` modifica la variable de 8 *threads* a 2 i després a 3. Llavors l'ultim `#pragma_omp_parallel` només crea 3 *threads*.

2. Indicate the two alternatives to supersede the number of threads that is specified by the OMP NUM THREADS environment variable.

- Canviant temporalment el nombre de *threads*:

```
#pragma omp parallel num_threads(4);
```

Figure 10: Canvi temporal nombre de threads

- Canviant definitivament el nombre de *threads*:

```
omp_set_num_threads(i); // i is the number of threads
#pragma omp parallel;
```

Figure 11: Canvi definitiu nombre de threads

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

La diferència entre aquestes dues formes per canviar el nombre de threads és que la primera només canvia el seu valor temporalment (per a la regió paral·lela). En canvi, a la segona manera, estem canviant el valor per a la resta del programa.

Per tant, a la versió temporal, si executem el programa amb la comanda `OMP_NUM_THREADS=8`, aquesta variable global es pot modificar durant el programa amb el `omp_set_num_threads(i)`. D'altra banda `num_threads(i)` només tindrà afecte en la zona del `parallel`.

`4.data_sharing.c`

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

Si executem `4.data_sharing` vàries vegades la sortida sempre és la mateixa:

La primera sortida és 120 perquè és la suma dels ids de cada thread. Si els 16 threads comparteixen una mateixa variable, afegiran el seu identificador a la mateixa variable. Mentre que, si privatitzem la variable, cada thread serà l'únic a la seva regió paral·lela amb accés, de manera que tots tindran el mateix id que és 0, sumant-se al valor inicial de x.

```
par1302@boada-1:~/lab2/openmp/Day1$ ./4.data_sharing
After first parallel (shared) x is: 120
After second parallel (private) x is: 5
After third  parallel (firstprivate) x is: 5
After fourth parallel (reduction) x is: 125
```

Figure 12: Sortida `4.data_sharing`

La part compartida seria:

$$\sum_{n=0}^{15} n = 120$$

Aleshores, el valor x es canvia a 5. Com que la variable és privada, la variable no té valor, el que significa que obtindrà un nombre aleatori. El valor només es canvia dins de la regió paral·lela, de manera que tan aviat com surt, el valor x roman intacte, sent 5. Per al firstprivate, la principal diferència entre la clàusula privada, és que aconsegueix que la variable x s'inicialitzi el valor, aquest és 5, a més, aquest valor només es canvia a la regió paral·lela, després torna a ser 5. Finalment, a la clàusula de reducció, el valor de x encara és 5. Quan entra a la regió paral·lela, crea una còpia per a cada thread i, a mesura que acaba, els afegeix l'un a l'altre. La sortida matemàtica seria aquesta:

$$\sum_{n=0}^{15} n = 120 + xvalue(5)$$

`5.datarace.c`

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

```
par1302@boada-1:~/lab2/openmp/Day1$ ./5.datarace
Program executed correctly - maxvalue=15 found
par1302@boada-1:~/lab2/openmp/Day1$ ./5.datarace
Program executed correctly - maxvalue=15 found
par1302@boada-1:~/lab2/openmp/Day1$ ./5.datarace
Program executed correctly - maxvalue=15 found
par1302@boada-1:~/lab2/openmp/Day1$ ./5.datarace
Program executed correctly - maxvalue=15 found
par1302@boada-1:~/lab2/openmp/Day1$ ./5.datarace
Program executed correctly - maxvalue=15 found
par1302@boada-1:~/lab2/openmp/Day1$ ./5.datarace
Program executed correctly - maxvalue=15 found
par1302@boada-1:~/lab2/openmp/Day1$ ./5.datarace
Program executed correctly - maxvalue=15 found
par1302@boada-1:~/lab2/openmp/Day1$ ./5.datarace
Sorry, something went wrong - incorrect maxvalue=12 found
par1302@boada-1:~/lab2/openmp/Day1$ ./5.datarace
Program executed correctly - maxvalue=15 found
par1302@boada-1:~/lab2/openmp/Day1$ ./5.datarace
^[[AProgram executed correctly - maxvalue=15 found
par1302@boada-1:~/lab2/openmp/Day1$ ./5.datarace
Program executed correctly - maxvalue=15 found
```

Figure 13: Sortides `5.datarace`

No retorna la mateixa resposta a causa del bucle. Podem veure que el codi inicia la regió paral·lela al començament del main. Això fa que, un cop entrem al bucle, no tinguem control de l'ordre en el que cada thread executarà el codi. El problema principal és que hi ha un data race, de manera que en el cas que un thread trobi el valor correcte, un altre pot canviar el valor de la variable maxvalue, donant lloc a un error.

2. 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.

- Solució 1:

```
for (i=id; i < N; i+=howmany) {
            #pragma omp critical
            if (vector[i] > maxvalue)
                maxvalue = vector[i];
        }
```

Figure 14: Solució alternativa 1 sense canviar estructura de codi

- Solució 2:

```
omp_set_num_threads(8);
#pragma omp parallel private(i) reduction(max:maxvalue)
{
int id = omp_get_thread_num();
int howmany = omp_get_num_threads();

for (i=id; i < N; i+=howmany) {
       if (vector[i] > maxvalue)
          maxvalue = vector[i];
    }
}
```

Figure 15: Solució alternativa 2 sense canviar estructura de codi

La versió amb el `critical` és molt més lenta ja que la regió de l'`if` s'executa de manera concurrent.

En la versió del `reduction`, el compilador crea una còpia privada del fitxer variable de reducció que s'utilitza per emmagatzemar el resultat parcial calculat per cada fil; aquesta variable s'inicialitza amb el valor neutre de l'operador especificat. Al final de la regió, el compilador assegura que la variable compartida s'actualitza correctament combinant els resultats parcials calculats per cada thread. En aquest cas, com no hi ha cap operació, especifiquem a la crida que volem el valor màxim dels calculats.

3. 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.

```
for (i=id*(N/howmany+1); i < (id+1)*(N/howmany+1) && i < N; i++) {
        if (vector[i] > maxvalue)
           maxvalue = vector[i];
    }
```

Figure 16: Alternative distribution of iterations to implicit tasks (threads) so that each of them executes only one block of consecutive iterations

Hem dividit la mida del vector pel nombre de threads per distribuir el treball per a cada thread. Per fer-ho, hem inicialitzat la variable `i` amb l'identificador del thread i multiplicant-la per la fracció de N entre número de threads. Pel que fa a la comparació del bucle, hem comparat la variable `i` amb N, que ha de ser inferior a aquest nombre i inferior a ((id+1)*N/ número de threads).

6.datarace.c

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

Es pot donar el cas que el resultat sigui erroni degut a que dos o més threads estiguin escrivint a la varaible `countmax`. Aquesta vegada, cada vegada que troba el valor màxim, n'afegeix un a una variable. El problema és la data race, de nou, pot ocasionar que dos threads canvïin el valor al mateix temps i n'està afectant la integritat.

`countmax` al mateix temps.

```
par1305@boada-1:~/lab2/openmp/Day1$ ./6.datarace
Program executed correctly - maxvalue=15 found 3 times
par1305@boada-1:~/lab2/openmp/Day1$ ./6.datarace
Sorry, something went wrong - incorrect maxvalue=15 found 2 times
```

Figure 17: Execució ./6.datarace amb error

2. 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.

Una opció és utilitzar el `reduction`, de tal manera que cada thread tindrà la seva variable local `countmax` i al final de tot es farà una suma d'aquestes variables locals per obtenir el `countmax` definitiu.

```
    omp_set_num_threads(8);
    #pragma omp parallel private(i) reduction(+:countmax)
    {
    int id = omp_get_thread_num();
    int howmany = omp_get_num_threads();

    for (i=id; i < N; i+=howmany) {
        if (vector[i]==maxvalue)
            countmax++;
        }
    }
```

Figure 18: Modificació programa

D'altra banda és pot usar l'`atomic`, que garantitza que no hi hagi conflicte en el moment de l'escriptura i lectura de la variable.

```
1       omp_set_num_threads(8);
2       #pragma omp parallel private(i)
3       {
4       int id = omp_get_thread_num();
5       int howmany = omp_get_num_threads();
6
7       for (i=id; i < N; i+=howmany) {
8           if (vector[i]==maxvalue) {
9               #pragma omp atomic
10              countmax++;
11          }
12      }
```

Figure 19: atomic modification

7.datarace.c

1. Is this program executing correctly? If not, explain why it is not providing the correct result for one or the two variables (`countmax` and `maxvalue`)

El programa no s'executa correctament perquè `countmax` no està protegit i llavors cada thread podria trobar un valor màxim.

```
par1305@boada-1:~/lab2/openmp/Day1$ ./7.datarace
Sorry, something went wrong - maxvalue=15 found 9 times
```

Figure 20: Sortida ./7.datarace

2. Write a correct way to synchronise the execution of implicit tasks (threads) for this program.
Una possible solució seria ajuntant els dos programes que hem fet anteriorment. Així no hi haurà conflicte.

```
    int i, maxvalue=0;
    int countmax = 0;

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

    for (i=id; i < N; i+=howmany) {
           if (vector[i] > maxvalue)
              maxvalue = vector[i];
        }
    }

    #pragma omp parallel private(i) reduction(+:countmax)
    {
    int id = omp_get_thread_num();
    int howmany = omp_get_num_threads();

    for (i=id; i < N; i+=howmany) {
        if (vector[i]==maxvalue)
            countmax++;
        }
    }
```

Figure 21: Correct way to synchronise the execution of implicit tasks

8.barrier.c

1. Can you predict the sequence of `printf` in this program? Do threads exit from the `#pragma omp barrier` construct in any specific order?

No és possible saber l'ordre d'execució de la primera part ni després del barrier, ja que depèn de la velocitat d'execució del codi.

```
(0) going to sleep for 2000 milliseconds ...
(1) going to sleep for 5000 milliseconds ...
(2) going to sleep for 8000 milliseconds ...
(3) going to sleep for 11000 milliseconds ...
(0) wakes up and enters barrier ...
(1) wakes up and enters barrier ...
(2) wakes up and enters barrier ...
(3) wakes up and enters barrier ...
(0) We are all awake!
(1) We are all awake!
(2) We are all awake!
(3) We are all awake!
par1305@boada-1:~/lab2/openmp/Day1$ ./8.barrier
(2) going to sleep for 8000 milliseconds ...
(0) going to sleep for 2000 milliseconds ...
(1) going to sleep for 5000 milliseconds ...
(3) going to sleep for 11000 milliseconds ...
(0) wakes up and enters barrier ...
(1) wakes up and enters barrier ...
(2) wakes up and enters barrier ...
(3) wakes up and enters barrier ...
(0) We are all awake!
(1) We are all awake!
(2) We are all awake!
(3) We are all awake!
```

Figure 22: Sortida barrier

Day 2: explicit tasks

`1.single.c`

1. What is the `nowait` clause doing when associated to `single`?

El que fem declarant un bloc `single` és que per a aquesta regió de codi específica, només en volem un sol `thread` per executar el bloc mentre els altres `threads` lliures estan esperant. Si escrivim la clàusula `nowait`, bàsicament estem dient als altres fils que ometin aquesta "regió d'un sol `thread`" i segueixin executant la resta del codi.

2. 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?

La diferència entre l'execució amb i sense nowait, és que en el primer cas, podem veure al terminal que la sortida s'imprimeix per grups de 4. Això passa com a resultat que un `thread` està executant la primera iteració, les restants esperen fins que s'acabi la primera i després entren a una altra iteració. D'altra banda, sense la clàusula nowait, la sortida sembla ser seqüencial a causa de l'únic `thread` que està fent totes les iteracions i els altres estan esperant que acabi.

```
par1302@boada-1:~/lab2/openmp/Day2$ ./1.single
Thread 1 executing instance 1 of single
Thread 0 executing instance 0 of single
Thread 2 executing instance 2 of single
Thread 3 executing instance 3 of single
Thread 0 executing instance 5 of single
Thread 1 executing instance 4 of single
Thread 2 executing instance 6 of single
Thread 3 executing instance 7 of single
Thread 0 executing instance 8 of single
Thread 1 executing instance 9 of single
Thread 2 executing instance 10 of single
Thread 3 executing instance 11 of single
Thread 0 executing instance 13 of single
Thread 1 executing instance 12 of single
Thread 2 executing instance 14 of single
Thread 3 executing instance 15 of single
Thread 0 executing instance 16 of single
Thread 1 executing instance 17 of single
Thread 2 executing instance 18 of single
Thread 3 executing instance 19 of single
```

Figure 23: Sortida ./1.single

`2.fibtasks.c`

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

El codi no s'executa en paral·lel perquè no hi ha cap clàusula que indiqui cap regió. Li falta el `#pragma omp parallel`. Això fa que no s'executi en paral·lel, ja que no hi ha cap tasca a fer.

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

```
1       #pragma omp parallel
2       #pragma omp single
3       #pragma omp task
4       while (p != NULL) {
5             printf("Thread %d creating task that will compute %d\n",
                  omp_get_thread_num(), p->data);
6             #pragma omp task firstprivate(p)
7                 processwork(p);
8             p = p->next;
9       }
```

Figure 24: Error de compilació amb ambdues clàusules

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

La clàusula `firstprivate(p)` indica que cada `thread` tindrà la seva instància de la variable `p`, inicialitzada anteriorment abans del `parallel`. Quan ho comentem, passa que tota l'execució la fa només un `thread`. Podem veure que si executem diverses vegades, l'identificador del `thread` canvia, cosa que ens indica que s'estan creant diversos fils, però només un està executant el programa.

`3.taskloop.c`

1. Which iterations of the loops are executed by each thread for each task `grainsize` or `num_tasks` specified?

A `grainsize` s'indica quantes iteracions es realitzaran per tasca. Cada `thread`, que té una mida de gra establerta com a 4, farà 4 iteracions del bucle, ho podem veure clarament en aquesta sortida.

```
Thread 0 distributing 12 iterations with grainsize(4) ...
Loop 1: (0) gets iteration 8
Loop 1: (1) gets iteration 4
Loop 1: (1) gets iteration 5
Loop 1: (1) gets iteration 6
Loop 1: (1) gets iteration 7
Loop 1: (0) gets iteration 9
Loop 1: (0) gets iteration 10
Loop 1: (0) gets iteration 11
Loop 1: (2) gets iteration 0
Loop 1: (2) gets iteration 1
Loop 1: (2) gets iteration 2
Loop 1: (2) gets iteration 3
```

Figure 25: Sortida de `3.taskloop` *grainsize(4)*

El num_tasks, especifica el nombre mínim de tasques que es crearan, de manera que, per exemple, en aquest cas, que el valor és 4, es crearan almenys 4 tasques. Com el bucle són 12 iteracions, les 4 tasques queden repartides en 3 iteracions entre els diferents threads, però no podem dir quin executarà quina tasca en concret. De fet un thread podria arribar a fer-ne 3 de les 4 tasques o no fer-ne cap...

```
Thread 0 distributing 12 iterations with num_tasks(4) ...
Loop 2: (1) gets iteration 0
Loop 2: (1) gets iteration 1
Loop 2: (1) gets iteration 2
Loop 2: (2) gets iteration 3
Loop 2: (2) gets iteration 4
Loop 2: (2) gets iteration 5
Loop 2: (1) gets iteration 6
Loop 2: (1) gets iteration 7
Loop 2: (1) gets iteration 8
Loop 2: (0) gets iteration 9
Loop 2: (0) gets iteration 10
Loop 2: (0) gets iteration 11
```

Figure 26: Sortida de `3.taskloop` *num_tasks(4)*

2. 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?

El nombre de tasques que executarà cada thread pot variar per execució. Ara, en la primera part, es crearan 2 tasques de 6 iteracions per tasca. Són 6 en lloc de 5 perquè es fa una divisió entera: 12/5 = 2 tasques i per tant 6*2 = 12 iteracions. En canvi, en la segona part es crearan 5 tasques: 2 tasques que executaran 3 iteracions cada una i 3 tasques que executaran 2 iteracions cada una.

```
Thread 0 distributing 12 iterations with grainsize(5) ...
Loop 1: (0) gets iteration 6
Loop 1: (0) gets iteration 7
Loop 1: (0) gets iteration 8
Loop 1: (0) gets iteration 9
Loop 1: (1) gets iteration 0
Loop 1: (1) gets iteration 1
Loop 1: (1) gets iteration 2
Loop 1: (1) gets iteration 3
Loop 1: (1) gets iteration 4
Loop 1: (1) gets iteration 5
Loop 1: (0) gets iteration 10
Loop 1: (0) gets iteration 11
```

Figure 27: Sortida de `3.taskloop` *grainsize(5)*

```
Thread 0 distributing 12 iterations with num_tasks(5) ...
Loop 2: (1) gets iteration 0
Loop 2: (1) gets iteration 1
Loop 2: (1) gets iteration 2
Loop 2: (1) gets iteration 3
Loop 2: (1) gets iteration 4
Loop 2: (1) gets iteration 5
Loop 2: (1) gets iteration 6
Loop 2: (1) gets iteration 7
Loop 2: (1) gets iteration 8
Loop 2: (1) gets iteration 9
Loop 2: (0) gets iteration 10
Loop 2: (0) gets iteration 11
```

Figure 28: Sortida de `3.taskloop` *num_tasks(5)*

3. Can `grainsize` and `num_tasks` be used at the same time in the same loop?

No, no es poden utilitzar en el mateix bucle perquè els seus objectius són oposats. A més, la compilació ens mostra error.

```
par1302@boada-1:~/lab2/openmp/Day2$ make 3.taskloop
icc 3.taskloop.c -Wall -g -O3 -fno-inline -fopenmp -fopenmp -o 3.taskloop
3.taskloop.c(28): error: directive cannot contain both grainsize and num_tasks clauses
        #pragma omp taskloop grainsize(VALUE) num_tasks(VALUE) // nogroup
                                              ^

compilation aborted for 3.taskloop.c (code 2)
Makefile:17: recipe for target '3.taskloop' failed
make: *** [3.taskloop] Error 2par1302@boada-1:~/lab2/openmp/Day2$ make 3.taskloop
icc 3.taskloop.c -Wall -g -O3 -fno-inline -fopenmp -fopenmp -o 3.taskloop
3.taskloop.c(28): error: directive cannot contain both grainsize and num_tasks clauses
        #pragma omp taskloop grainsize(VALUE) num_tasks(VALUE) // nogroup
                                              ^

compilation aborted for 3.taskloop.c (code 2)
Makefile:17: recipe for target '3.taskloop' failed
make: *** [3.taskloop] Error 2
```

Figure 29: Error de compilació amb ambdues clàusules

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

El que està passant és que el primer i el segon bucles s'executen simultàniament, les dues parts de la sortida es fusionen. Això passa perquè quan la clàusula s'especifica *nogroup*, no es crea cap regió de bucle de tasques i totes les tasques s'assignen al mateix buffer.

`4.reduction.c`

1. 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.

Primer de tot hem inicialitzat la variable *sum* a 0 abans de cada part. Hem afegit un `reduction` a la segona part per assegurar que el resultat de la soma fos correcte. I en la 3a part, en el primer loop hem afegit un `taskgroup` per esperar a tots els threads i per assegurar la variable *sum* hem afegit un `task_reduction` amb un `in_reduction` i un `reduction` també al segon loop.

```
int main()
{
    int i;

    for (i=0; i<SIZE; i++)
        X[i] = i;

    omp_set_num_threads(4);
    #pragma omp parallel
    #pragma omp single
    {
        sum = 0;
        // 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
        sum = 0;
        #pragma omp taskloop grainsize(BS) reduction(+: sum)
        for (i=0; i< SIZE; i++)
            sum += X[i];

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

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

        #pragma omp taskloop grainsize(BS) reduction(+: 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;
}
```

Figure 30: Parallelisation completed

`5.synchtasks.c`

1. Draw the task dependence graph that is specified in this program

Podem obesrvar que foo4 depèn de foo1 i foo2. foo4 i foo3 són requisits de foo5.

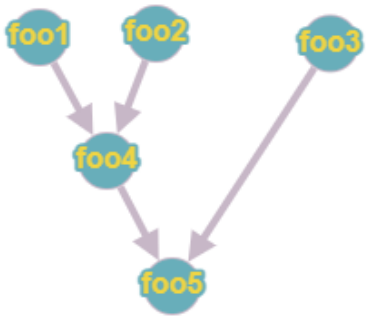

Figure 31: Graf de dependències de synchtaks.c

2. 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`.

Hem afegit dos `taskwait` abans de foo4 i foo5. El codi queda així:

```
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 foo3\n");
        #pragma omp task
        foo3();
    #pragma omp taskwait
        printf("Creating task foo4\n");
        #pragma omp task
        foo4();
    #pragma omp taskwait
        printf("Creating task foo5\n");
        #pragma omp task
        foo5();
    }
    return 0;
}
```

Figure 32: Program using only `taskwait` as task synchronisation mechanism I

3. 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`.

Hem separat l'ordre de l'execució de les foox() i hem utilitzat el `taskgroup` per garantir el graf de precedències:

```
int main(int argc, char *argv[]) {

    #pragma omp parallel
    #pragma omp single
    {
        printf("Creating task foo3\n");
        #pragma omp task
        foo3();
        #pragma omp taskgroup
        {
            #pragma omp taskgroup
            {
                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 task
            foo4();
        }
        printf("Creating task foo5\n");
        #pragma omp task
        foo5();
    }
    return 0;
}
```

Figure 33: Program using only `taskgroup` as task synchronisation mechanism II

3 Observing overheads

En aquest apartat ens basarem en els diferents codis de pi correctes, per treure conclusions sobre les seves millores de paral·lelització.

3.1 Day 1:

Number of threads	Critical (pi-v4.c)	Atomic (pi-v5.c)	Sum-local (pi-v6.c)	Reduction (pi-v7.c)
1	2.659s	0.001s	0.010s	0.008s
4	36.126s	5.546s	0.011s	0.010s
8	34.432s	5.892s	0.019s	0.020s

Figure 34: Taula dels temps d'execució de les diferents versions dels programes

Com podem veure, el temps d'execució de la versió amb el `critical` és terriblement lent. Dins del bucle diem que la instrucció (sum...) es faci d'un en un. La instrucció de la divisió és lenta. En canvi, si posem un `atomic`, aquesta instrucció es farà en paral·lel i finalment una suma al final. Tot i que es millora molt el temps, encara hi ha temps de sincronitzacions que no són desitjables.

D'altra banda, en el bucle de la versió 6 no hi ha cap `critical` ni cap `atomic`; s'utilitza la variable privada `double sumlocal` inicialitzada a 0, dins del bucle cada thread utilitza la seva variable i quan s'acaba el bucle s'actualitza `sum` de manera `atomic`. Això com es pot observar, ja és més eficient.

Per acabar, en la versió amb el `reduction(+:sum)` el compilador generarà un codi com el de la versió 6 i és per això que els temps de la versió 6 i 7 són semblants.

Pel que fa a la variació del número de threads, en el `critical` i una mica a l'`atomic`, com més threads, més costós serà. D'altra banda, en la resta de versions no serà tan impactant.

Per a cada versió s'han fet 100000000 iteracions, ja que era el més adequat.

3.2 Day 2:

A continuació podem veure l'overhead de pi_omp_parallel i pi_omp_tasks segons el número de threads:

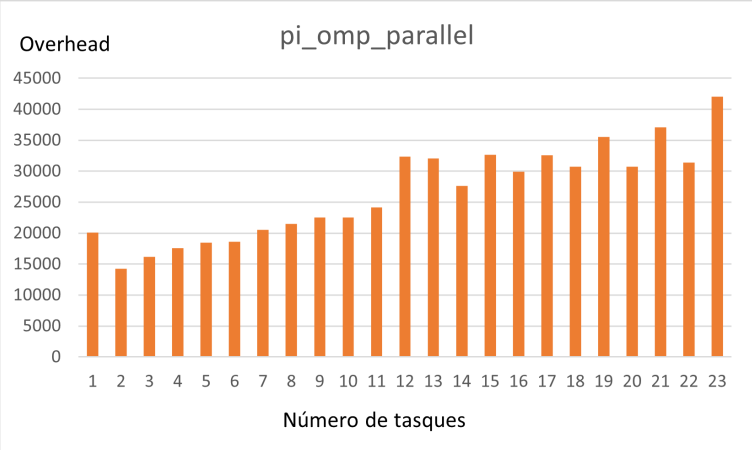

Figure 35: Overhead pi_omp_parallel

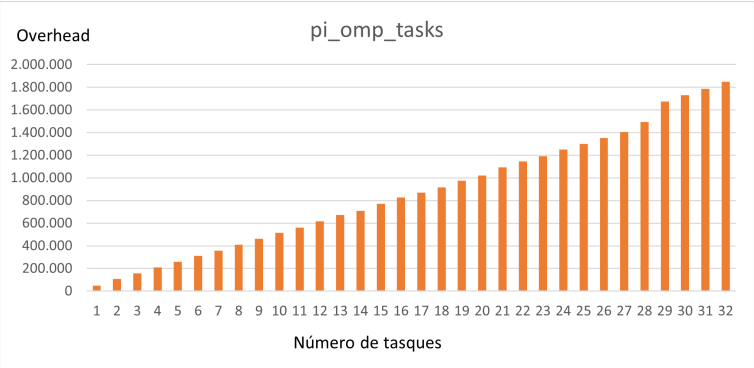

Figure 36: Overhead pi_omp_tasks

Com podem observar en les dues gràfiques, és que a mesura que es va augmentant el número de threads, el temps d'overhead augmenta. En el pi_omp_tasks augmenta de manera clarament lineal mentre que en el pi_omp_parallel casi lineal excepte amb alguna alteració.

Pel que fa al overhead per thread, en el pi_omp_parallel ens hem trobat que amb 2 threads era alt, però anava disminuint a mesura que afegies threads. En el pi_omp_tasks, ens hem trobat que l'overhead per thread es mantenia constant.

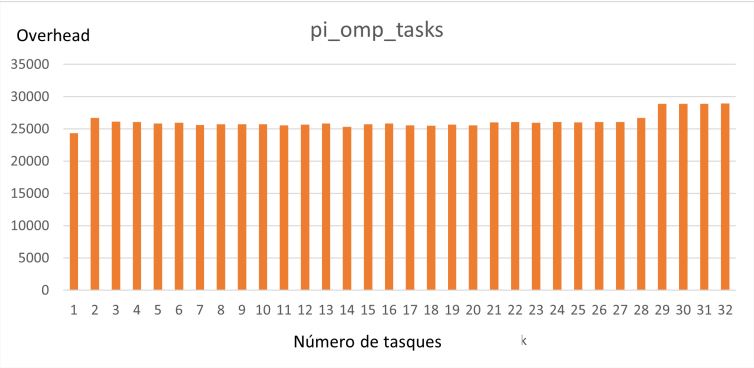

Figure 37: Overhead per thread pi_omp_tasks

En conclusió, el paral·lelisme ens ajuda a executar els programes més ràpidament, però s'ha de tenir en compte que el codi estigui ben programat i de manera eficient i no abusar del número de threads ja que pot augmentar el temps d'overhead degut el temps de sincronització.

Els sobrecostos de temps en general són provocats per la creació de threads i totes les coses que es relacionen amb l'intercanvi de dades o l'accés a les variables, escriure o llegir-hi. La creació de diverses tasques, no difereix en el temps d'execució o els sobrecostos. Augmentarà òbviament el temps, però podem esperar un resultat lineal, no com quan es creen més threads.