Brief tutorial on OpenMP programming model

Ismael Douha Prieto, Eloi Cruz Harillo - Grup 1305

Q1 2019/20

1 Introduction

Durant aquesta pràctica hem estat aprenent com fer servir l'eina de paral·lelització OpenMP, fent servir diverses eines de les que disposa. També hem estat analitzant els overheads que presenta la paral·lelització d'un programa.

2 OpenMP questionnaire

2.1 Parallel region

2.1.1 1.hello.c

1.- En aquest programa, es veu com s'imprimeix per pantalla 24 cops el missatge "Hello world!", que es el mateix número que threads disposa el hardware emprat.

2.- Fent servir l'opció de l'imatge aconseguim que només s'imprimeixi per pantalla 4 cops el missatge

```
par1305@boada-1:~/lab2/openmp/basics$ ↩
    OMP_NUM_THREADS=4 ./1.hello

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

2.1.2 2.hello.c

En aquest cas trobem que hi ha una condició de cursa (datarace) sobre la variable id, provocant que l'execució no sigui correcta, no essent de forma equitativa la distribució, amb threads executant més codi que altres.

```
par1305@boada-1:~/lab2/openmp/basics$ ./2.hello

(0) Hello (0) world!
(4) Hello (1) Hello (1) world!
(1) world!
(3) Hello (1) world!
(2) Hello (2) world!
(5) Hello (5) world!
(6) Hello (6) world!
(7) Hello (7) world!
```

Això es pot solucionar declarant com a privada la variable id, quedant de la següent forma

```c
#include <stdio.h>
#include <omp.h>

/* Q1: Is the execution of the program correct? Add a   */
/*     data sharing clause to make it correct           */
/* Q2: Are the lines always printed in the same order?  */
/*     Why the messages sometimes appear intermixed?    */

int main ()
{
    int id;
    #pragma omp parallel num_threads(8) private(id)
    {
        id =omp_get_thread_num();
        printf("(%d) Hello ",id);
        printf("(%d) world!\n",id);
    }
    return 0;
}
```

Figure 1: Codi modificat del fitxer "2.hello.c".

Amb la modificació feta al codi, la execució sortiria de la següent forma:

```
par1305@boada-1:~/lab2/openmp/basics$ ./2.hello

(0) Hello (0) world!
(3) Hello (3) world!
(2) Hello (2) world!
(1) Hello (1) world!
(4) Hello (4) world!
(7) Hello (7) world!
(6) Hello (6) world!
(5) Hello (5) world!
```

2.1.3 how_many.c

1.- Tal i com es pot veure a la sortida de l'execució del codi, s'imprimeixen 20 "Hello world ...". Això ho hem pogut esbrinar facilment gracies a la combinació de comandes "./3.how_many — grep Hello — wc -l" com es pot apreciar.

```
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 second parallel (2)!
Hello world from the second parallel (2)!
Hello world from the second parallel (3)!
Hello world from the second parallel (3)!
Hello world from the second parallel (3)!
Outside parallel, nobody else here ... (1 thread)
Hello world from the third parallel (4)!
Hello world from the third parallel (4)!
Hello world from the third parallel (4)!
Hello world from the third parallel (4)!
Hello world from the fourth  parallel (3)!
Hello world from the fourth  parallel (3)!
Hello world from the fourth  parallel (3)!
Finishing, I'm alone again ... (1 thread)

par1305@boada-1:~/lab2/openmp/basics$ ./3.how_many↩
    | grep Hello
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 first parallel (8)!
Hello world from the second parallel (2)!
Hello world from the second parallel (2)!
Hello world from the second parallel (3)!
Hello world from the second parallel (3)!
Hello world from the second parallel (3)!
```

```
Hello world from the third parallel (4)!
Hello world from the third parallel (4)!
Hello world from the third parallel (4)!
Hello world from the third parallel (4)!
Hello world from the fourth  parallel (3)!
Hello world from the fourth  parallel (3)!
Hello world from the fourth  parallel (3)!

par1305@boada-1:~/lab2/openmp/basics$ ./3.how_many
    | grep Hello | wc -l
20
```

2.- Sempre que estem fora d'una regió paral·lela trobem que només hi ha 1 thread. En canvi, quan estem dins de una regió paral·lela hi han tants threads m cos'especifiquin mitjançant la sentencia "#pragma omp parallel num_threads(n)", on "n" és el número de threads que volem fer servir a la regió paral·lela.

2.1.4 4.data sharing.c

1.- Executant un parell de cops el programa obtenim la següent sortida:

```
par1305@boada-1:~/lab2/openmp/basics$ ./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
par1305@boada-1:~/lab2/openmp/basics$ ./4.
   data_sharing
After first parallel (shared) x is: 115
After second parallel (private) x is: 5
After third  parallel (firstprivate) x is: 5
After fourth parallel (reduction) x is: 125
```

A partir d'aquí, amb la pertinent revisió del codi obtenim les següents conclusions:

Quan es fa servir la shared sense fer servir cap tipus de sincronització, poden aparèixer errors degut a que dos threads poden llegir el mateix valors i per tant sobreescriure un valor erroni.

Les següents clàusules fetes servit són private i firstprivate. Les dues clàusules tenen un comportament semblant. Totes dues converteixen la variable x en privada per a cada thread. Això comporta que el valor de la x al acabar la zona paral·lea sugi el mateix que al començar la zona paral·lela perque mai s'arriva a modifica r.S'ha de putuar que la diferència entre private i firstprivate no es pot

veure reflectida en la execució d'aquest codi ja que no tenim en compte el valor que prenen les variables en l'ambit local dels threads. 3- Finalment la clàusula reduction fa que obtinguem sempre el valor esperat 125 que es el resultat de la suma de 5 (el valor inicial de la variable x i el num de thread de cada thread (de 0 a 15). Reduction executa tots els threads que llegeixen el valor inicial de la variable x i al acavar la zona paral·lela es sumen els resultats de totes les execucions.

2.2 Loop Parallelism

2.2.1 1.schedule.c

1.- A partir de la sortida del codi que es pot veure a continuació i l'anàlisi pertinent del codi arribem a les següents conclusions:

Per al schedule amb "schedule(static)", al dividir-se en parts de mida $N/num\_threads$ aproximadament, essent asignades mitjançant una política Round Robin. Tenint N = 12 amb 4 theards, tenim que cada thread fa 3 iteracions.

La següent opció que tenim és "schedule(static, 2)". Amb aquesta opció lo que fem es dividir l'espai de iteracions en parts de mida 2, per tant tindrem 6 parts, provocant que alguns els threads executin 1 troç i d'altres 2, ja que si tots executessin només un troç, ens quedarien 2 troços sense executar.

Per al tercer loop tenim la clausula "schedule(dynamic,2)", la qual fa que es divideixi l'espai d'iteracion i s'assignin als threads de forma dinamica. Per tant, no hi haurà una distribució equitativa del treball, tal i com es pot observar a la sortida, si no que el primer que acabi executarà un altre troç fins que s'acabin.

Per últim, es fa servir la clàusula "schedule(guided, 2)", que disminueix el "chunk" de les parts a mesura que va disminuïnt el nombre de iteracions restants. Com comencem amb mida 2, de seguida passen a ser troços de la mida mínima, repartint-se als threads en funció de quan aquests terminin les seves execucions prèvies.

```
1 par1305@boada-1:~/lab2/openmp/worksharing$ ./1.↩
    schedule
2 Going to distribute 12 iterations with schedule(↩
    static) ...
3 Loop 1: (0) gets iteration 0
4 Loop 1: (0) gets iteration 1
5 Loop 1: (0) gets iteration 2
6 Loop 1: (1) gets iteration 3
7 Loop 1: (1) gets iteration 4
8 Loop 1: (1) gets iteration 5
9 Loop 1: (2) gets iteration 6
```

```
Loop 1: (2) gets iteration 7
Loop 1: (2) gets iteration 8
Loop 1: (3) gets iteration 9
Loop 1: (3) gets iteration 10
Loop 1: (3) gets iteration 11
Going to distribute 12 iterations with schedule(static, 2) ...
Loop 2: (3) gets iteration 6
Loop 2: (3) gets iteration 7
Loop 2: (1) gets iteration 2
Loop 2: (1) gets iteration 3
Loop 2: (1) gets iteration 10
Loop 2: (1) gets iteration 11
Loop 2: (0) gets iteration 0
Loop 2: (0) gets iteration 1
Loop 2: (0) gets iteration 8
Loop 2: (0) gets iteration 9
Loop 2: (2) gets iteration 4
Loop 2: (2) gets iteration 5
Going to distribute 12 iterations with schedule(dynamic, 2) ...
Loop 3: (0) gets iteration 2
Loop 3: (0) gets iteration 3
Loop 3: (0) gets iteration 8
Loop 3: (0) gets iteration 9
Loop 3: (0) gets iteration 10
Loop 3: (0) gets iteration 11
Loop 3: (1) gets iteration 4
Loop 3: (1) gets iteration 5
Loop 3: (3) gets iteration 0
Loop 3: (3) gets iteration 1
Loop 3: (2) gets iteration 6
Loop 3: (2) gets iteration 7
Going to distribute 12 iterations with schedule(guided, 2) ...
Loop 4: (0) gets iteration 0
Loop 4: (3) gets iteration 6
Loop 4: (3) gets iteration 7
Loop 4: (3) gets iteration 8
Loop 4: (3) gets iteration 9
Loop 4: (3) gets iteration 10
Loop 4: (3) gets iteration 11
Loop 4: (0) gets iteration 1
Loop 4: (1) gets iteration 2
Loop 4: (1) gets iteration 3
Loop 4: (2) gets iteration 4
```

```
53 Loop 4: (2) gets iteration 5
```

2.2.2 2.nowait.c

1.- Per al següent codi, trobem que qualsevol thread pot executar qualsevol iteració de les que hi han, lo que faria que tinguéssim $num\_threads^{num\_iteracions}$ possibles sortides.

```
1 par1305@boada-1:~/lab2/openmp/worksharing$ ./2.↩
    nowait
2 Loop 1: thread (0) gets iteration 0
3 Loop 1: thread (1) gets iteration 1
4 Loop 2: thread (3) gets iteration 2
5 Loop 2: thread (2) gets iteration 3
```

2.- Si treïem la clausula "nowait" del primer for, trobem que tots els threads s'hauran de sincronitzar un cop acabi la regió paral·lela, provocant que els threads que han executat codi durant el primer for també el puguin executar durant els segon.

```
1 par1305@boada-1:~/lab2/openmp/worksharing$ ./2.↩
    nowait
2 Loop 1: thread (0) gets iteration 0
3 Loop 1: thread (2) gets iteration 1
4 Loop 2: thread (2) gets iteration 2
5 Loop 2: thread (0) gets iteration 3
```

3.- Si enlloc de fer servir la clausula dynamic en els dos loops, fem servir la static, sempre executaran els mateixos threads cada bucle i en el mateix ordre. Aixo es deu per a que em passat de fer servir una política de planificació dinàmica a una Round Robin.

```
1 par1305@boada-1:~/lab2/openmp/worksharing$ ./2.↩
    nowait
2 Loop 1: thread (0) gets iteration 0
3 Loop 1: thread (1) gets iteration 1
4 Loop 2: thread (0) gets iteration 2
5 Loop 2: thread (1) gets iteration 3
```

2.2.3 3.collapse.c

1.- En aquest codi, trobem que cada thread executara una iteració del bucle j, independentment de quin valor trobem al bucle i.

```
par1305@boada-1:~/lab2/openmp/worksharing$ ./3.collapse
(0) Iter (0 0)
(0) Iter (0 1)
(0) Iter (0 2)
(0) Iter (0 3)
(2) Iter (1 2)
(2) Iter (1 3)
(3) Iter (2 0)
(3) Iter (2 1)
(3) Iter (2 2)
(7) Iter (4 2)
(7) Iter (4 3)
(7) Iter (4 4)
(5) Iter (3 1)
(5) Iter (3 2)
(5) Iter (3 3)
(2) Iter (1 4)
(4) Iter (2 3)
(4) Iter (2 4)
(4) Iter (3 0)
(6) Iter (3 4)
(6) Iter (4 0)
(6) Iter (4 1)
(1) Iter (0 4)
(1) Iter (1 0)
(1) Iter (1 1)
```

2.- Si traiem la clàusula collapse, es necessari afegir la clàusula ordered per a que s'executin totes les iteracions del bucle.

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>        /* OpenMP */
#define N 5

/* Q1: Which iterations of the loops are executed by each thread     */
/*     when the collapse clause is used?                             */
/* Q2: Is the execution correct if we remove the collapse clause?    */
/*     Add the appropriate clause to make it correct.                */

int main()
{
   int i,j;

   omp_set_num_threads(8);
   #pragma omp parallel for ordered
   for (i=0; i < N; i++) {
#pragma omp ordered
      for (j=0; j < N; j++) {
         int id=omp_get_thread_num();
         printf("(%d) Iter (%d %d)\n",id,i,j);
       }
   }

   return 0;
}
```

Figure 2: Codi modificat del fitxer "3.collapse.c".

Un cop afegida, el codi ja s'executaria de forma correcta, com es pot apreciar a la següent imatge.

```
par1305@boada-1:~/lab2/openmp/worksharing$ ./3.collapse
(0) Iter (0 0)
(0) Iter (0 1)
(0) Iter (0 2)
(0) Iter (0 3)
(0) Iter (0 4)
(1) Iter (1 0)
(1) Iter (1 1)
(1) Iter (1 2)
(1) Iter (1 3)
(1) Iter (1 4)
(2) Iter (2 0)
(2) Iter (2 1)
(2) Iter (2 2)
(2) Iter (2 3)
(2) Iter (2 4)
(3) Iter (3 0)
(3) Iter (3 1)
(3) Iter (3 2)
(3) Iter (3 3)
(3) Iter (3 4)
```

```
(4) Iter (4 0)
(4) Iter (4 1)
(4) Iter (4 2)
(4) Iter (4 3)
(4) Iter (4 4)
```

2.3 Synchronization

2.3.1 1.datarace.c

1.- El programa no sexecuta correctament..

```
par1305@boada-1:~/lab2/openmp/synchronization$ ./1.datarace
Sorry, something went wrong, value of x = 954639
par1305@boada-1:~/lab2/openmp/synchronization$ ./1.datarace
Sorry, something went wrong, value of x = 754183
par1305@boada-1:~/lab2/openmp/synchronization$ ./1.datarace
Sorry, something went wrong, value of x = 945717
```

2.- Les dues modificacion que hem fet han sigut afegir les clausules atomic i critical respectivament. La clàusula critical fa que la instrucció x++ s'executi de forma atòmica per part del procesador i aixi s'evita el data racing. La clausula critical qfa que tota una zona de codi s'executi de forma atomica però es fa per software i es menys eficient.

```
#pragma omp parallel for schedule(dynamic,1) private(i) shared(x)
for (i=0; i < N; i++) {
    //#pragma omp atomic
  #pragma omp critical(x)
  x++;
}
```

2.3.2 2.barrier.c

1.- EL programa crea quatre threads. Cada thread escriu per pantalla tres missatges. En primer lloc "going to sleep...". Aquest missatge no es pot predir l'ordre en que sortira ja que cada thread entrarà a la cpu de forma "aleatòria". Seguidament escriuen el missatge "wakes up and...". Aquests missatges si que es

pot predir que primer l'escriurà el thread amb la id 0, el següent el thread amb la id 1, el següent el thread amb la id 2 i finalment el thread amb la id 4. Això es perquè el temps d'espera de cada thread ve definit per la seva id. Així quan més gran sigui la id més temps s'esperarà. Finalment l'últim missatge "We are all awake!" l'escriuen els processos de forma aleatòria un altre cop perquè tots esperen al barrier per esperar a que tots el processos acabin el codi a executar abans del barrier.

```
1  par1305@boada-1:~/lab2/openmp/synchronization$ ↩
       ./2.barrier
2  (0) going to sleep for 2000 milliseconds ...
3  (1) going to sleep for 5000 milliseconds ...
4  (2) going to sleep for 8000 milliseconds ...
5  (3) going to sleep for 11000 milliseconds ...
6  (0) wakes up and enters barrier ...
7  (1) wakes up and enters barrier ...
8  (2) wakes up and enters barrier ...
9  (3) wakes up and enters barrier ...
10 (3) We are all awake!
11 (0) We are all awake!
12 (1) We are all awake!
13 (2) We are all awake!
14 par1305@boada-1:~/lab2/openmp/synchronization$ ↩
       ./2.barrier
15 (0) going to sleep for 2000 milliseconds ...
16 (2) going to sleep for 8000 milliseconds ...
17 (1) going to sleep for 5000 milliseconds ...
18 (3) going to sleep for 11000 milliseconds ...
19 (0) wakes up and enters barrier ...
20 (1) wakes up and enters barrier ...
21 (2) wakes up and enters barrier ...
22 (3) wakes up and enters barrier ...
23 (3) We are all awake!
24 (2) We are all awake!
25 (0) We are all awake!
26 (1) We are all awake!
```

2.3.3 3.ordered.c

1.- La clàusula ordered fa que les iteracions del bucle s'executin per ordre (simulant un funcionament seqüencial). Els missatges "Inside" pinter per pantalla el número de iteració que estan executant. La clàusula "Inside" provoca que es pintin les iteracions per ordre. En canvi els missatges "Before" es pinten sense cap clàusula de control i això fa que pintin els missatges de forma desordenada. Per a cada iteració sempre es pintarà primer el before que l'inisde.

```
Before ordered - (0) gets iteration 0
Inside ordered - (0) gets iteration 0
Before ordered - (0) gets iteration 1
Inside ordered - (0) gets iteration 1
Before ordered - (0) gets iteration 2
Inside ordered - (0) gets iteration 2
Before ordered - (0) gets iteration 3
Inside ordered - (0) gets iteration 3
Before ordered - (1) gets iteration 5
Before ordered - (7) gets iteration 9
Before ordered - (5) gets iteration 10
Before ordered - (6) gets iteration 11
Before ordered - (0) gets iteration 7
Before ordered - (4) gets iteration 8
Before ordered - (3) gets iteration 4
Inside ordered - (3) gets iteration 4
Before ordered - (3) gets iteration 12
Inside ordered - (1) gets iteration 5
Before ordered - (1) gets iteration 13
Before ordered - (2) gets iteration 6
Inside ordered - (2) gets iteration 6
Inside ordered - (0) gets iteration 7
Inside ordered - (4) gets iteration 8
Before ordered - (2) gets iteration 14
Before ordered - (0) gets iteration 15
Inside ordered - (7) gets iteration 9
Inside ordered - (5) gets iteration 10
Inside ordered - (6) gets iteration 11
Inside ordered - (3) gets iteration 12
Inside ordered - (1) gets iteration 13
Inside ordered - (2) gets iteration 14
Inside ordered - (0) gets iteration 15
```

2.- Fent que la plaificació enlloc de fer-se de forma dinàmica es faci de forma estàtica.

```
omp_set_num_threads(8);
#pragma omp parallel
{
    #pragma omp for schedule(static, 2) ordered
    for (i=0; i < N; i++) {
            int id=omp_get_thread_num();
            printf("Before ordered - (%d) gets ↩
```

```
                iteration %d\n",id,i);
8           #pragma omp ordered
9           printf("Inside ordered - (%d) gets ↩
                iteration %d\n",id,i);
10      }
11  }
```

2.4 Tasks

2.4.1 1.single.c

Per al següent codi, trobem aquesta sortida:

```
1  par1305@boada-1:~/lab2/openmp/tasks$ ./1.single
2  Thread 0 executing instance 0 of single
3  Thread 1 executing instance 1 of single
4  Thread 3 executing instance 2 of single
5  Thread 2 executing instance 3 of single
6  Thread 0 executing instance 4 of single
7  Thread 1 executing instance 5 of single
8  Thread 3 executing instance 6 of single
9  Thread 2 executing instance 7 of single
10 Thread 0 executing instance 8 of single
11 Thread 1 executing instance 9 of single
12 Thread 3 executing instance 10 of single
13 Thread 2 executing instance 11 of single
14 Thread 0 executing instance 12 of single
15 Thread 1 executing instance 13 of single
16 Thread 3 executing instance 14 of single
17 Thread 2 executing instance 15 of single
18 Thread 0 executing instance 16 of single
19 Thread 1 executing instance 17 of single
20 Thread 3 executing instance 18 of single
21 Thread 2 executing instance 19 of single
```

Com es pot apreciar, tots els threads contribueixen a l'execució del bucle. Això es degut a que amb la clausula single forcem a que cada cop s'accedeixi a la regió paral·lela només executi cada iteració un únic thread. Al afegir el nowait, evitant que els demes threads hagi d'esperar-se a que el thread que esta a la regió paral·lela acabi. Mentre s'esta executant el codi observem dona l'aparença de que s'estigui executant en ràfegues. Això pasa degut al sleep(1) que trobem després del printf, que força al thread a estar un segon quiet.

2.4.2 2.fibtasks.c

Executant el codi pertinent trobem que el thread 0 crea totes les tasques i les executa, tal i com es pot observar a continuació.

```
par1305@boada-1:~/lab2/openmp/tasks$ ./2.fibtasks
Staring computation of Fibonacci for numbers in linked list
Thread 0 creating task that will compute 1
Thread 0 creating task that will compute 2
Thread 0 creating task that will compute 3
Thread 0 creating task that will compute 4
Thread 0 creating task that will compute 5
Thread 0 creating task that will compute 6
Thread 0 creating task that will compute 7
Thread 0 creating task that will compute 8
Thread 0 creating task that will compute 9
Thread 0 creating task that will compute 10
Thread 0 creating task that will compute 11
Thread 0 creating task that will compute 12
Thread 0 creating task that will compute 13
Thread 0 creating task that will compute 14
Thread 0 creating task that will compute 15
Thread 0 creating task that will compute 16
Thread 0 creating task that will compute 17
Thread 0 creating task that will compute 18
Thread 0 creating task that will compute 19
Thread 0 creating task that will compute 20
Thread 0 creating task that will compute 21
Thread 0 creating task that will compute 22
Thread 0 creating task that will compute 23
Thread 0 creating task that will compute 24
Thread 0 creating task that will compute 25
Finished creation of tasks to compute the Fibonacci for numbers in linked list
Finished computation of Fibonacci for numbers in linked list
1: 1 computed by thread 0
2: 1 computed by thread 0
3: 2 computed by thread 0
4: 3 computed by thread 0
5: 5 computed by thread 0
6: 8 computed by thread 0
7: 13 computed by thread 0
8: 21 computed by thread 0
```

```
9: 34 computed by thread 0
10: 55 computed by thread 0
11: 89 computed by thread 0
12: 144 computed by thread 0
13: 233 computed by thread 0
14: 377 computed by thread 0
15: 610 computed by thread 0
16: 987 computed by thread 0
17: 1597 computed by thread 0
18: 2584 computed by thread 0
19: 4181 computed by thread 0
20: 6765 computed by thread 0
21: 10946 computed by thread 0
22: 17711 computed by thread 0
23: 28657 computed by thread 0
24: 46368 computed by thread 0
25: 75025 computed by thread 0
```

Això es degut a que no hi ha definida cap regió paral·lela i per tant només hi ha un thread.
2.- Si volem fer que el programa passi a ser paral·lel cal modificar el codi, tal i com es pot apreciar a la imatge.

```
int main(int argc, char *argv[]) {
     struct node *temp, *head;

     printf("Staring computation of Fibonacci for numbers in linked list \n");

     p = init_list(N);
     head = p;

     while (p != NULL) {
           printf("Thread %d creating task that will compute %d\n", omp_get_thread_num(), p->data);
           #pragma omp parallel
           #pragma omp single
           #pragma omp task
               processwork(p);
           p = p->next;
     }
     printf("Finished creation of tasks to compute the Fibonacci for numbers in linked list \n");

     printf("Finished computation of Fibonacci for numbers in linked list \n");
     p = head;
     while (p != NULL) {
        printf("%d: %d computed by thread %d \n", p->data, p->fibdata, p->threadnum);
        temp = p->next;
        free (p);
        p = temp;
     }
     free (p);

     return 0;
}
```

Figure 3: Codi modificat del fitxer "2.fibtasks.c".

Un cop el codi ja sigui paral·lelitzat, la sortida seria la següent:

```
par1305@boada-1:~/lab2/openmp/tasks$ ./2.fibtasks
```

```
Staring computation of Fibonacci for numbers in linked list
Thread 0 creating task that will compute 1
Thread 0 creating task that will compute 2
Thread 0 creating task that will compute 3
Thread 0 creating task that will compute 4
Thread 0 creating task that will compute 5
Thread 0 creating task that will compute 6
Thread 0 creating task that will compute 7
Thread 0 creating task that will compute 8
Thread 0 creating task that will compute 9
Thread 0 creating task that will compute 10
Thread 0 creating task that will compute 11
Thread 0 creating task that will compute 12
Thread 0 creating task that will compute 13
Thread 0 creating task that will compute 14
Thread 0 creating task that will compute 15
Thread 0 creating task that will compute 16
Thread 0 creating task that will compute 17
Thread 0 creating task that will compute 18
Thread 0 creating task that will compute 19
Thread 0 creating task that will compute 20
Thread 0 creating task that will compute 21
Thread 0 creating task that will compute 22
Thread 0 creating task that will compute 23
Thread 0 creating task that will compute 24
Thread 0 creating task that will compute 25
Finished creation of tasks to compute the Fibonacci for numbers in linked list
Finished computation of Fibonacci for numbers in linked list
1: 1 computed by thread 3
2: 1 computed by thread 2
3: 2 computed by thread 17
4: 3 computed by thread 0
5: 5 computed by thread 17
6: 8 computed by thread 20
7: 13 computed by thread 19
8: 21 computed by thread 15
9: 34 computed by thread 4
10: 55 computed by thread 5
11: 89 computed by thread 23
12: 144 computed by thread 15
13: 233 computed by thread 5
14: 377 computed by thread 15
15: 610 computed by thread 3
```

```
16: 987 computed by thread 5
17: 1597 computed by thread 19
18: 2584 computed by thread 15
19: 4181 computed by thread 20
20: 6765 computed by thread 15
21: 10946 computed by thread 7
22: 17711 computed by thread 15
23: 28657 computed by thread 15
24: 46368 computed by thread 11
25: 75025 computed by thread 15
```

On la computació es reparteix entre els diferents threads.

2.4.3 3.synchtasks.c

1.- Si analitzem el codi pertinent, podem observar que amb les clausules "in" i "out" es defineixin unes clares dependecies que queden reflectides al TDG que es pot veure a continuació.

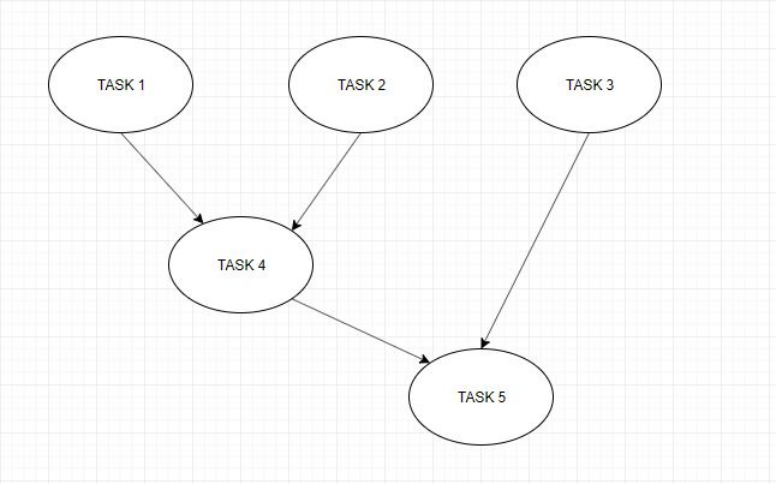


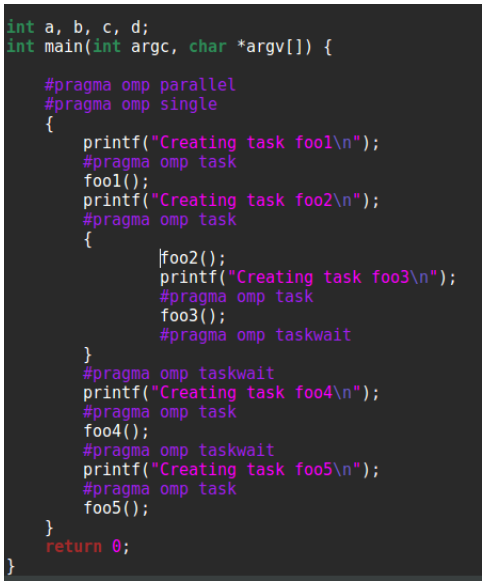
Figure 4: Task Dependency Graph per al codi "3.synchtasks.c".

2.-

```
par1305@boada-1:~/lab2/openmp/tasks$ ./3.synchtasks
Creating task foo1
Creating task foo2
Creating task foo3
Starting function foo2
```

```
Starting function foo1
Creating task foo4
Creating task foo5
Starting function foo3
Terminating function foo2
Terminating function foo1
Starting function foo4
Terminating function foo4
Terminating function foo3
Starting function foo5
Terminating function foo5
```

Aquest mateix TDG es pot generar només fent servir clàusules "taskwait" com es pot apreciar al codi de la imatge, tenint la sortida que es pot apreciar més endavant.



```c
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 foo3\n");
              #pragma omp task
              foo3();
              #pragma omp taskwait
       }
       #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 5: Codi modificat del fitxer "3.synchtasks.c".

```
par1305@boada-1:~/lab2/openmp/tasks$ ./3.synchtasks
Creating task foo1
Creating task foo2
Starting function foo2
Starting function foo1
```

```
Terminating function foo2
Creating task foo3
Starting function foo3
Terminating function foo1
Terminating function foo3
Creating task foo4
Starting function foo4
Terminating function foo4
Creating task foo5
Starting function foo5
Terminating function foo5
```

2.4.4 4.taskloop.c

1.- En aquest codi, trobem que si usem la clausula "grainsize(5)", es generaran tascas de mida com a mínim 5, per tant com només hi han 12 iteracions, trobem que es generen 2 tascas de 6 iteracions, les quals es repartiran entre 1 o 2 threads, segons com accedeixin al task pool.
Si fem servir la clausula "num_tasks(5)", enlloc de decidir la mida decidirem el nombre de tasques a crear. En aquest cas es generaran 5 tasques, essent algunes de 2 iteracions i altres de 3, tal i com es pot observar a la sortida de l'execució del programa. L'asignació d'aquestes tasques als threads serian mitjançant el mateix metode que amb la clàusula "gransize(5)", amb el task pool.

```
par1305@boada-1:~/lab2/openmp/tasks$ ./4.taskloop
Going to distribute 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: (0) gets iteration 10
Loop 1: (0) gets iteration 11
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
Going to distribute 12 iterations with num_tasks(5) ...
Loop 2: (1) gets iteration 0
Loop 2: (1) gets iteration 1
Loop 2: (0) gets iteration 10
```

```
Loop 2: (2) gets iteration 3
Loop 2: (1) gets iteration 2
Loop 2: (1) gets iteration 8
Loop 2: (1) gets iteration 9
Loop 2: (3) gets iteration 6
Loop 2: (0) gets iteration 11
Loop 2: (3) gets iteration 7
Loop 2: (2) gets iteration 4
Loop 2: (2) gets iteration 5
```

2.- Si descomentem la clàusula "nogroup", evitem que es crei una regió de "taskgroup", fent que les iteracions del primer loop es facin al segon, encara que la generació de tasques no variara.

```
par1305@boada-1:~/lab2/openmp/tasks$ ./4.taskloop
Going to distribute 12 iterations with grainsize(5) ...
Going to distribute 12 iterations with num_tasks(5) ...
Loop 1: (2) gets iteration 6
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 1: (3) gets iteration 0
Loop 1: (3) gets iteration 1
Loop 1: (3) gets iteration 2
Loop 1: (3) gets iteration 3
Loop 1: (3) gets iteration 4
Loop 1: (3) gets iteration 5
Loop 2: (1) gets iteration 9
Loop 2: (0) gets iteration 10
Loop 2: (0) gets iteration 11
Loop 1: (2) gets iteration 7
Loop 1: (2) gets iteration 8
Loop 1: (2) gets iteration 9
Loop 1: (2) gets iteration 10
Loop 1: (2) gets iteration 11
```

3 Observing overheads

L'overhead és el temps d'execució extra que necessiten els programes per funcionar correctament a més del seu temps d'execució real. Aquest temps s'afegeix per diversos motius. Per exemple quan en un programa paral·lel s'han d'afegir mecanismes de sincronització per evitar el datarace. Començarem explicant els diversos mecanismes de sincronització que hem pogut observar.

En primer lloc hem vist dues clàusules aparentment amb un funcionament molt semblant (atomic i critical). Aquestes dues les hem utilitzat per solucionar problemes de datarace on diversos threads llegien de la mateixa variable al mateix moment i per tant en el moment d'escriure un thread no tenia en compte el valor del altre thread. La clàusula atomic té la capacitat de fer que una única instrucció d'alt nivell s'executi de de forma atòmica. Això significa que el processador executi totes les operacions de baix nivell corresponents a la instrucció d'alt nivell sense executar cap altre operació pel mig. La instrucció critical té la capacitat de definir una regió de codi d'alt nivell perquè s'executi sense que s'executi res per mig. Utilitzar aquestes clausules afegeix overhead per tal de poder sincronitzar el accessos a memòria. S'ha de dir que la clàusula critical afegeix més overhead perque és una sincronització per software en canvi la sincronització quan s'utilitza atomic la duu a terme el processador.

En segon lloc hem vist la clàusula barrier. Aquesta actua de la següent forma. En un moment determinat existeixen diversos threads que estàn executant el mateix codi. Quan un thread arriba a un barrier ha d'esperar fins que tots els threads que estan executant el mateix tros de codi arribin també al barrier. Aquest temps d'espera s'afegeix al temps total d'execució.

En tercer lloc hem descobert la clàusula ordered que te la capacitat que donades diferents tasques que executen iteracions d'un bucle els diferents fragments de codi desitjats s'executin de forma ordenada (la primera iteració s'executi primer, la segons iteració s'executi en segon lloc i aixi successivament) independentment del thread al qual hagin sigut assignades.