

PAR Laboratory Assignment

Laboratori 2: Brief tutorial on OpenMP

programming model



**UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH**

Professor:

Jordi Tubella

Alumnes:

Eric Hurtado (par2114)

Jordi Pitarch (par2118)

Índice

1. Introducción	3
2. Session 1	
2.1. hello.c	4
2.2. how_many.c	6
2.3. data_sharing.c	8
2.4. datarace.c	9
3. Session 2	
3.1. single.c	13
3.2. fibtasks.c	13
3.3. taskloop.c	15
3.4. reduction.c	18
3.5. synchtasks.c	19
4. Observing overheads	21
5. Conclusiones	24

1. Introducción

Este trabajo de laboratorio ha sido preparado con el propósito de introducir los principales constructos en las extensiones OpenMP al lenguaje de programación C. En cada una de las dos sesiones primero irás a través de un conjunto de diferentes versiones de código (algunas de ellas no correctas) para el cálculo del número Pi en paralela; y luego se le presentará un conjunto de ejemplos simples que serán útiles para practicar. Se están introduciendo los componentes principales del modelo de programación OpenMP. Le pedimos que complete el cuestionario en la parte entregable de este segundo trabajo de laboratorio. La sesión terminará observando los gastos generales introducidos por el uso de diferentes construcciones de sincronización en OpenMP.

2. Session 1

2.1 hello.c

1.hello.c

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

El "Hello world!", se ejecuta dos veces debido a que el “pragma omp parallel” crea por defecto 2 threads en la ejecución

```
par2118@boada-8:~/lab2/openmp/Day1$ make 1.hello
icc -Wall -g -O3 -fno-inline -fopenmp -o 1.hello 1.hello.c
par2118@boada-8:~/lab2/openmp/Day1$ ./1.hello
Hello world!
Hello world!
par2118@boada-8:~/lab2/openmp/Day1$ █
```

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

Para cambiar el número de threads que se ejecutan en un “pragma omp parallel”, usamos el flag OMP_NUM_THREADS.

```
par2118@boada-8:~/lab2/openmp/Day1$ OMP_NUM_THREADS=4 ./1.hello
Hello world!
Hello world!
Hello world!
Hello world!
par2118@boada-8:~/lab2/openmp/Day1$ █
```

2.hello.c

- 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?

```
par2118@boada-8:~/lab2/openmp/Day1$ ./2.hello
(0) Hello (1) Hello (1) world!
(3) Hello (1) world!
(4) Hello (4) world!
(5) Hello (5) world!
(1) world!
(6) Hello (6) world!
(2) Hello (2) world!
(7) Hello (7) world!
par2118@boada-8:~/lab2/openmp/Day1$ █
```

La ejecución del programa es errónea debido a que se pueden ejecutar varios threads a la vez y por tanto los mensajes pueden aparecer intercalados, pero nunca cortados por la mitad.

Para hacer que el código se ejecute de forma correcta hemos añadido la cláusula private a la variable “id”.

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

- 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 tiene por qué debido a que lo único que se garantiza es que no se repitan los mensajes de “hello” y de “world”, Por tanto estos pueden aparecer en cualquier orden.

```
par2118@boada-8:~/lab2/openmp/Day1$ ./2.hello
(0) Hello (0) world!
(1) Hello (1) world!
(2) Hello (2) world!
(4) Hello (4) world!
(5) Hello (5) world!
(3) Hello (3) world!
(6) Hello (6) world!
(7) Hello (7) world!
par2118@boada-8:~/lab2/openmp/Day1$ ./2.hello
(0) Hello (0) world!
(3) Hello (3) world!
(2) Hello (2) world!
(1) Hello (1) world!
(4) Hello (4) world!
(5) Hello (5) world!
(6) Hello (6) world!
(7) Hello (7) world!
```

2.2 how_many.c

3.how_many.c

- **What does `omp_get_num_threads` return when invoked outside and inside a parallel region?**

El “`omp_get_num_threads()`, nos devuelve el número de threads que hay en ejecución en cada momento. Por tanto en nuestro código, este será lo mismo al número de veces que se ejecuta cada instrucción de nuestro código.

Como se aprecia en la imagen posterior, la primera región se ha ejecutado 8 veces, debido a que hay 8 threads. En la segunda región al haber 4 threads, se ejecuta 4 veces y por tanto el “`omp_get_number_threads`” da 4 en este caso. En la tercera región al volver a depender de la variable global definida al principio, vuelve a ser ejecutado 8 veces. En la cuarta región al ser un bucle con dos iteraciones y ser la $i=2$ y $i=3$ y definir estos como el número de threads, se ejecuta respectivamente cada iteración “ i ” veces. La quinta región es igual a la segunda región y por tanto al estar definida como 4, se ejecuta 4 veces. Por último en la sexta región se ejecuta 3 veces debido a que la última iteración del for nos ha dejado con $i=3$, y por tanto coge este valor para definir el número de threads en ejecución.

```
par2118@boada-8:~/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 second parallel (4)!
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 sixth parallel (3)!
Hello world from the sixth parallel (3)!
Hello world from the sixth parallel (3)!
Finishing, I'm alone again ... (1 thread)
par2118@boada-8:~/lab2/openmp/Day1$
```

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

Además de la utilizada en el apartado anterior, también se pueden usar “omp_set_num_threads(#threads) o num_threads(#threads).

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

Tanto con el uso de “OMP_NUM_THREADS”, como para “omp_set_num_threads”, mantienen el número de threads hasta que se indique otro número con “omp_set_num_threads”.

Por otra parte para el “num_threads” solo se mantiene en la región paralela en la cual actúa la cláusula.

2.3 data_sharing.c

4.data_sharing.c

- **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)**

Los valores de la primera región paralela no se pueden asegurar debido al “shared(x)”, pero este nunca será mayor a 31, debido a que hay 32 threads definidos por el “omp_set_num_threads(32)”. Para la segunda región paralela, se toma el valor x=5 y al un valor privado, se pierde el valor al salir de la cláusula. Para la tercera región paralela, pasa lo mismo que en la región anterior, con la diferencia de que ahora cada thread inicializa de forma local la variable x=5, pero se siguen perdiendo al final de la cláusula. Por último, en la cuarta región paralela, se crean variables locales y al final se hace un reduction, que es la suma de los resultados obtenidos en una variable global, en nuestro caso 501.

```
par2118@boada-8:~/lab2/openmp/Day1$ ./4.data_sharing
After first parallel (shared) x is: 31
After second parallel (private) x is: 5
After third parallel (firstprivate) x is: 5
After fourth parallel (reduction) x is: 501
par2118@boada-8:~/lab2/openmp/Day1$ ./4.data_sharing
After first parallel (shared) x is: 31
After second parallel (private) x is: 5
After third parallel (firstprivate) x is: 5
After fourth parallel (reduction) x is: 501
par2118@boada-8:~/lab2/openmp/Day1$ ./4.data_sharing
After first parallel (shared) x is: 30
After second parallel (private) x is: 5
After third parallel (firstprivate) x is: 5
After fourth parallel (reduction) x is: 501
par2118@boada-8:~/lab2/openmp/Day1$ ./4.data_sharing
After first parallel (shared) x is: 30
After second parallel (private) x is: 5
After third parallel (firstprivate) x is: 5
After fourth parallel (reduction) x is: 501
par2118@boada-8:~/lab2/openmp/Day1$
```

2.4 datarace.c

5.datarace.c

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

No tiene por que dar siempre el resultado correcto debido a los datarace que hay en el código. Esta es provocada a que varios threads pueden estar escribiendo la variable "maxvalue" y por tanto esta puede dar resultados distintos para cada caso.

```
par2118@boada-8:~/lab2/openmp/Day1$ ./5.datarace
Sorry, something went wrong - incorrect maxvalue=11 found
par2118@boada-8:~/lab2/openmp/Day1$ ./5.datarace
Program executed correctly - maxvalue=15 found
par2118@boada-8:~/lab2/openmp/Day1$ ./5.datarace
Program executed correctly - maxvalue=15 found
par2118@boada-8:~/lab2/openmp/Day1$ ./5.datarace
Sorry, something went wrong - incorrect maxvalue=11 found
```

- 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.

En primer lugar para evitar estos datarace, hemos decidido hacer critica la región para cada ejecución del "for". Así nos aseguramos que cuando un thread ejecute este código, no haya ningún otro thread ejecutando el mismo código. Sin embargo esta ejecución produce unos

overheads muy grandes, debido al tiempo de sincronización entre threads para poder acceder a la región crítica.

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

Por otra parte, también podríamos combatir este datarace con un reduction de la variable “maxvalue”. Con esta directiva nos aseguramos que al final de la región el operador max se aplica correctamente para cada una de las soluciones parciales y así obtener el resultado deseado.

```
int i, maxvalue=3;

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)
        {
            sleep(1); // this is just to force problems
            maxvalue = vector[i];
        }
    }
}
```

- 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).

Una forma alternativa basandonos en la segunda versión del apartado anterior seria mediante modificando las iteraciones del bucle del “for”, para que este se desplace de bloque en bloque. Esto lo hemos conseguido mediante la inicialización de “i” al principio del bloque y esto se consigue mediante el tamaño del bloque por el número de thread que

ejecuta ese bloque. Por otra parte hay que comprobar que el bloque existe en la condición del medio. Y por último iteramos la “i” con un “i+=1”.

```
int i, maxvalue=3;

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();
    double size = N;
    int size_block = size/howmany;
    for (i=id*size_block; i <size_block * (id+1); i+=1) {

        if (vector[i] > maxvalue)
        {
            sleep(1); // this is just to force problems
            maxvalue = vector[i];
        }
    }
}
```

6.datarace.c

- 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)

Esta versión tampoco devuelve el resultado correcto debido a que hay dataraces tanto en la escritura como en la lectura de los datos. A diferencia de la versión anterior, en este caso la variable que provoca este problema es la “countmax”, debido a que puede ser modificada sin tener en cuenta lo que han hecho otros threads.

```
par2118@boada-8:~/lab2/openmp/Day1$ ./6.datarace
Program executed correctly - maxvalue=15 found 3 times
par2118@boada-8:~/lab2/openmp/Day1$ ./6.datarace
Program executed correctly - maxvalue=15 found 3 times
par2118@boada-8:~/lab2/openmp/Day1$ ./6.datarace
Sorry, something went wrong - incorrect maxvalue=15 found 2 times
par2118@boada-8:~/lab2/openmp/Day1$ ./6.datarace
Sorry, something went wrong - incorrect maxvalue=15 found 2 times
par2118@boada-8:~/lab2/openmp/Day1$ ./6.datarace
Sorry, something went wrong - incorrect maxvalue=15 found 0 times
par2118@boada-8:~/lab2/openmp/Day1$ ./6.datarace
Program executed correctly - maxvalue=15 found 3 times
```

- 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.

Para solucionar los “datarace”, en primer lugar hemos implementado el “pragma omp atomic”, justo antes de la edición de la variable “countmax”, para que se aprovechen las operaciones atomicas que nos proporciona nuestro hardware, para que así el incremento de countmax se haga de forma indivisible.

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

Por otra parte hemos decidido hacer “pragma omp barrier” para que así evitar lectura indeseadas entre threads.

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

3. Session 2

3.1 single.c

1.single.c

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

Normalmente la cláusula “nowait” asociada a un “single, elimina la espera provocada por el segundo, para así permitir al resto de threads poder ejecutar de forma paralela la tarea generada por el single.

- **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?**

Todos los threads contribuyen a la ejecución de múltiples instancias de single, debido a la eliminación de la espera. Como consecuencia aparecerá en la ejecución siempre que se ejecutan en grupos de 4, 1 por thread, pero no tendrán siempre el mismo orden, tal como se puede ver en la captura siguiente.

```
par2118@boada-7:~/lab2/openmp/Day2$ make 1.single
icc 1.single.c -Wall -g -O3 -fno-inline -fopenmp -fopenmp -o 1.single
par2118@boada-7:~/lab2/openmp/Day2$ ./1.single
Thread 0 executing instance 0 of single
Thread 1 executing instance 1 of single
Thread 3 executing instance 2 of single
Thread 2 executing instance 3 of single
Thread 0 executing instance 4 of single
Thread 1 executing instance 5 of single
Thread 3 executing instance 6 of single
Thread 2 executing instance 7 of single
Thread 0 executing instance 8 of single
Thread 1 executing instance 9 of single
Thread 3 executing instance 10 of single
Thread 2 executing instance 11 of single
Thread 1 executing instance 12 of single
Thread 0 executing instance 13 of single
Thread 2 executing instance 15 of single
Thread 3 executing instance 14 of single
Thread 1 executing instance 16 of single
Thread 2 executing instance 18 of single
Thread 0 executing instance 17 of single
Thread 3 executing instance 19 of single
par2118@boada-7:~/lab2/openmp/Day2$ █
```

3.2 fibtasks.c

2.fibtasks.c

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

Como no hay ningún “#pragma omp parallel”, se ejecuta todo de forma secuencial. La única cláusula que hay en el programa es un “#pragma omp task firstprivate(p)”, la cual no influye en el paralelismo del programa.

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

Para conseguir que el bucle se ejecute en paralelo en cada iteración, hemos decidido implementar dos cláusulas:

“#pragma omp parallel”

“#pragma omp single”

En las capturas anteriores se puede observar tanto la modificación del código, como el resultado obtenido.

```
  x = fib(n - 3);
  y = fib(n - 2);
  return (x + y);
}

void processwork(struct node* p)
{
    int i;
    n = p->data;
    p->fibdata = fib(n);
    p->threadnum = omp_get_thread_num();
}

struct nodes init_list(int nvalues) {
    int i;
    struct node *head, *p1, *p2;
    p1 = (struct node*)malloc(sizeof(struct node));
    head = p1;
    p1->data = 1;
    p1->next = NULL;
    p1->threadnum = 0;
    for(i=1; i<nvalues; i++) {
        p2 = (struct node*)malloc(sizeof(struct node));
        p1->next = p2;
        p2->data = i;
        p2->next = NULL;
        p2->threadnum = 0;
        p1 = p2;
    }
    p1->next = NULL;
    return head;
}

struct node *p;

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

    omp_set_num_threads(4);
    printf("Starting computation of Fibonacci for numbers in linked list\n");

    p = init_list(10);
    head = p;
    #pragma omp parallel
    #pragma omp single
    while (p != NULL) {
        #pragma omp task firstprivate(p)
        #printf("Thread %d creating task that will compute %d\n",omp_get_thread_num(), p->data);
        temp = p->next;
        p = p->next;
        #pragma omp task
    }
    printf("Finished creation of tasks to compute the Fibonacci in linked list\n");
}
```

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

La cláusula “firstprivate(p)”, hace que todos los threads de la ejecución contengan su copia de “p”. Si lo eliminamos, se ejecutaría todo en el mismo thread, debido a que solo habría una copia de la variable. En la captura siguiente se puede observar el resultado de la ejecución del programa eliminando la cláusula

```

jordipitarchblasco-7:~/lab2/openmp/Day2$ ./fibTasks
Starting 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
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: 4389 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: 75925 computed by thread 0
par2118@boada-7:~/lab2/openmp/Day2$ 

```

3.3 taskloop.c

3.taskloop.c

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

En la captura siguiente, podemos observar el resultado de la ejecución del programa, en la cual se puede ver, el thread que ejecuta cada tarea en cada caso.

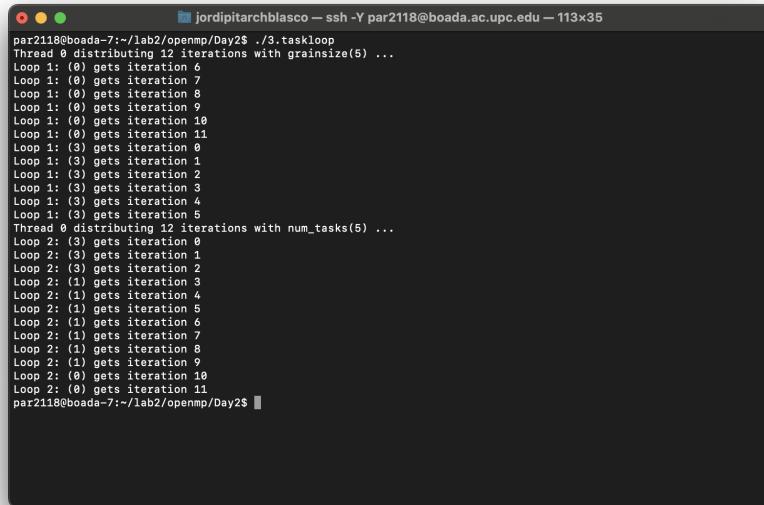
```

par2118@boada-7:~/lab2/openmp/Day2$ ./3.taskloop
Thread 0 distributing 12 iterations with grainsize(4) ...
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: (1) gets iteration 6
Loop 1: (1) 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
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: (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: (0) gets iteration 9
Loop 2: (0) gets iteration 10
Loop 2: (0) gets iteration 11
par2118@boada-7:~/lab2/openmp/Day2$ 

```

- 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?

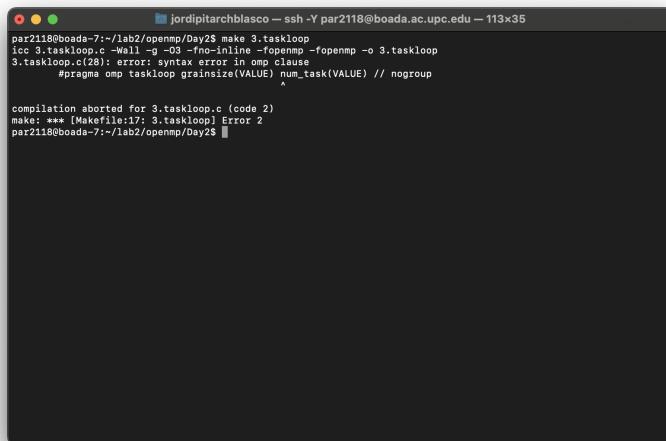
Ahora pasan a ejecutarse 5 iteraciones por cada thread. Por otra parte, podemos observar como ahora las tareas no aparecen de forma ordenada, esto es debido a que al aumentar el número de tareas paralelizables, no se garantiza el orden de finalización de los mismos.



```
jordipitarchblasco - ssh -Y par2118@boada.ac.upc.edu - 113x35
par2118@boada-7:~/lab2/openmp/Day2$ ./3.taskloop
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: (0) gets iteration 10
Loop 1: (0) gets iteration 11
Loop 1: (0) gets iteration 0
Loop 1: (0) gets iteration 1
Loop 1: (0) gets iteration 2
Loop 1: (0) gets iteration 3
Loop 1: (0) gets iteration 4
Loop 1: (0) gets iteration 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
Thread 0 distributing 12 iterations with num_tasks(5) ...
Loop 2: (3) gets iteration 0
Loop 2: (3) gets iteration 1
Loop 2: (3) 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
par2118@boada-7:~/lab2/openmp/Day2$
```

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

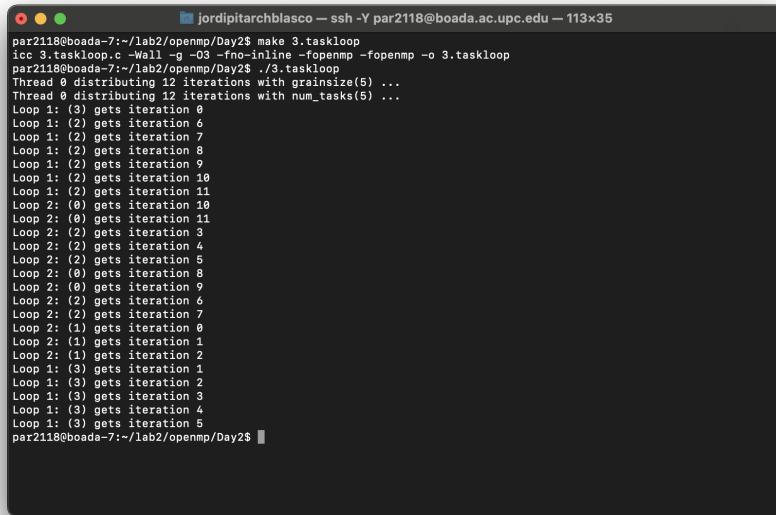
No se podría, debido a que dividen las tareas de forma distinta, por tanto son incompatibles.



```
jordipitarchblasco - ssh -Y par2118@boada.ac.upc.edu - 113x35
par2118@boada-7:~/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: syntax error in omp clause
    #pragma omp taskloop grainsize(VALUE) num_task(VALUE) // nogroup
                                         ^
compilation aborted for 3.taskloop.c (code 2)
make: *** [Makefile:17: 3.taskloop] Error 2
par2118@boada-7:~/lab2/openmp/Day2$
```

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

Si descomentamos la cláusula “nogroup” del primer bucle, se elimina el grupo de tareas implícito durante la construcción del bucle, como consecuencia, cada thread puede ejecutar la primera tarea que encuentre. Cada iteración se hará en una tarea.



A terminal window titled "jordipitarchblasco" showing the output of an OpenMP taskloop program. The window title is "jordipitarchblasco" and the command line is "par2118@boada:~/lab2/openmp/Day2\$". The output shows threads 0 and 1 distributing iterations across three loops (Loop 1, Loop 2, Loop 3) with grain size 5. The iterations are assigned sequentially to threads, demonstrating the lack of implicit grouping.

```
jordipitarchblasco -Y par2118@boada.ac.upc.edu - 113x35
par2118@boada:~/lab2/openmp/Day2$ make 3.taskloop
icc 3.taskloop.c -Wall -g -O3 -fno-inline -fopenmp -fopenmp -o 3.taskloop
par2118@boada:~/lab2/openmp/Day2$ ./3.taskloop
Thread 0 distributing 12 iterations with grainsize(5) ...
Thread 0 distributing 12 iterations with num_tasks(5) ...
Loop 1: (3) gets iteration 0
Loop 1: (2) gets iteration 6
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
Loop 2: (8) gets iteration 10
Loop 2: (8) gets iteration 11
Loop 2: (2) gets iteration 3
Loop 2: (2) gets iteration 4
Loop 2: (2) gets iteration 5
Loop 2: (8) gets iteration 8
Loop 2: (8) gets iteration 9
Loop 2: (2) gets iteration 6
Loop 2: (2) gets iteration 7
Loop 2: (1) gets iteration 0
Loop 2: (1) gets iteration 1
Loop 2: (1) gets iteration 2
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
par2118@boada:~/lab2/openmp/Day2$
```

3.4 reduction.c

4.reduction.c

- 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.

```
#define SIZE 8192
#define BS 16
int X[SIZE], sum;

int main()
{
    int i;

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

    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);
    }

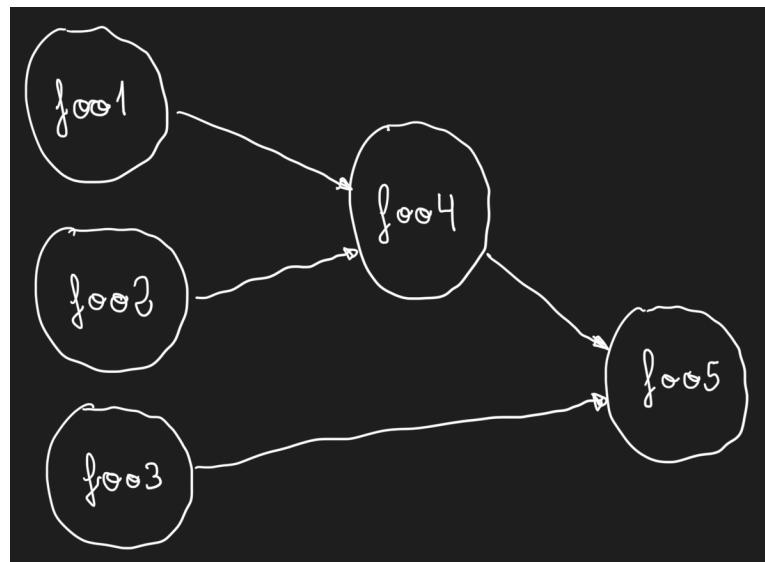
    return 0;
}
```

3.5 synctasks.c

5.synctasks.c

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

Debido a que la tarea “foo4” depende de “foo1” y “foo2”, esta se ejecutará después de estas, mientras que “foo5” al depender de “foo4”, será la última en ejecutarse. Por otra parte “foo1”, “foo2” y “foo3”, se ejecutarán al principio.



- 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.

Para hacer que las tareas “foo4” y “foo5” esperen a las otras tareas, hemos decidido implementar un “taskwait”, para que así esperen a la ejecución de los anteriores, y por tanto conseguimos el mismo resultado que en el apartado anterior.

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

- 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.

En este apartado hemos usado la cláusula “taskgroup”, en primer lugar para ejecutar “foo1”, “foo2” y “foo3”. A continuación hemos creado otro “taskgroup”, para “foo4”, debido a que es la siguiente tarea a ejecutar. Y por último se ejecuta “foo5”, la cual no es necesario crear una cláusula, debido a que se ejecuta al final.

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

4. Observing overheads

Synchronization overheads

- pi omp critical.c: a critical region is used to protect every access to sum, ensuring exclusive access to it. This version is equivalent to pi-v4.
- pi omp atomic.c: it makes use of atomic to guarantee atomic (indivisible) access to the memory location where variable sum is stored. This version is equivalent to pi-v5.
- pi omp sumlocal.c: a “per-thread” private copy sumlocal is used followed by a global update at the end using only one critical region. This version is equivalent to pi-v6.
- pi omp reduction.c: it makes use of the reduction clause applied to the global variable sum. cThis version is equivalent to pi-v7.

Hemos ejecutado cada 1 de las 4 versiones mencionadas anteriormente, y hemos obtenido los valores siguientes para 100.000.000 instrucciones y el número de threads indicados por la columna (todos los tiempos se encuentran en nanosegundos).

Versión	1 Thread	4 Threads	8 Threads	16 Threads
critical	1861902	48584067	42341259.375	46999295.3125
atomic	---	8212694	9785917	11747265.5
sumlocal	---	11101.25	17633.125	13061.6875
reduction	---	10641.5	16154.75	14434.9375

En la tabla superior podemos observar los diferentes tiempos de ejecución de cada una de las versiones del programa, en el caso del atomic, sumlocal y reduction en 1 thread hemos puesto “---” debido a que nos daban tiempos negativos, lo que significa que los tiempos de ejecución eran muy pequeños. Observando los tiempos de ejecución en comparación al 1,23458 segundos del tiempo de ejecución secuencial, podemos observar como los tiempos se reducen en todos los casos cuando ejecutamos el programa con el mismo número de threads, uno.

Por lo que respecta a las versiones con más threads ejecutando el código, podemos observar como en el caso del critical, el tiempo de ejecución aumenta a medida que aumenta el número de threads. A Pesar de que este usa el software para mejorar los tiempos, en este caso acaba provocando más overheads que se producen en el mismo, lo que impide una mejor eficiencia del código. Por otra parte, el atomic también aumenta el tiempo de ejecución debido a los overheads, pero en menor medida que el caso anterior.

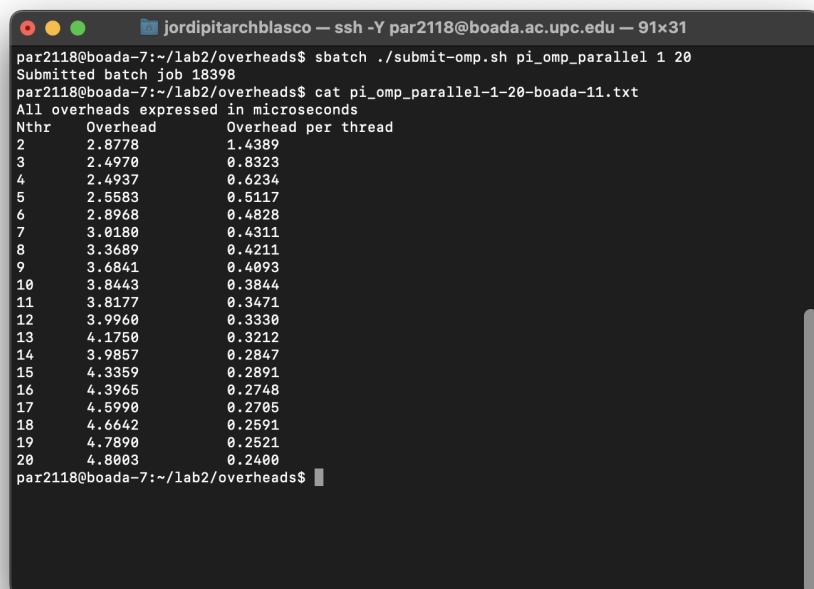
Esto es debido a que el atomic usa el hardware disponible, el cual es capaz de proteger regiones del código más rápido.

Por otra parte el sumlocal y el reduction, consiguen mejores resultados a medida que aumentan el número de threads, la cual significa que los overheads producidos durante la ejecución han sido mucho menores que en los dos primeros ejemplos explicados en el párrafo anterior. Estos dos se basan en que ellos solo esperan cuando tienen que hacer la reducción de los otros threads.

Thread creation and termination

En este apartado usaremos el “pi_omp_parallel.c”. Este código nos permite saber las diferencias temporales entre la ejecución secuencial y la ejecución paralela para cierto número de threads.

Observando los tiempos obtenidos en la imagen posterior, podemos ver que no tenemos un overhead constante en el tiempo, ya que este aumenta a medida que aumentan los threads del programa. Gracias a esto podemos concluir que cuando paralezas un programa, el overhead es inevitable, pero que a mayor número de threads ejecutando el programa, no significa que el overhead por thread también aumente, sino que este disminuye a medida que aumenta el otro.

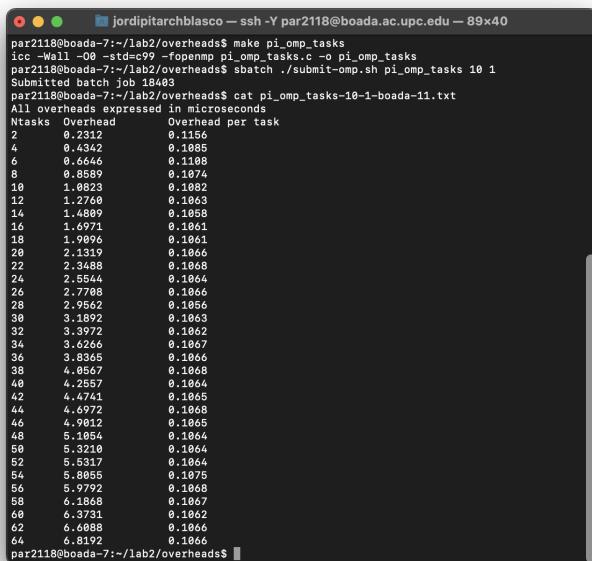


```
jordipitarchblasco - ssh -Y par2118@boada.ac.upc.edu - 91x31
par2118@boada-7:~/lab2/overheads$ sbatch ./submit-omp.sh pi_omp_parallel 1 20
Submitted batch job 18398
par2118@boada-7:~/lab2/overheads$ cat pi_omp_parallel-1-20-boada-11.txt
All overheads expressed in microseconds
Nthr    Overhead      Overhead per thread
 2      2.8778      1.4389
 3      2.4970      0.8323
 4      2.4937      0.6234
 5      2.5583      0.5117
 6      2.8968      0.4828
 7      3.0180      0.4311
 8      3.3689      0.4211
 9      3.6841      0.4093
10     3.8443      0.3844
11     3.8177      0.3471
12     3.9968      0.3330
13     4.1750      0.3212
14     3.9857      0.2847
15     4.3359      0.2891
16     4.3965      0.2748
17     4.5990      0.2705
18     4.6642      0.2591
19     4.7890      0.2521
20     4.8003      0.2400
par2118@boada-7:~/lab2/overheads$
```

Task creation and synchronisation

En este caso hemos usado “pi_omp_tasks.c”, este se basa al igual que en el anterior, para comparar los tiempos de ejecución de la versión secuencial y la versión que crea el mismo tipo de tareas que en el apartado anterior.

Como podemos observar en la imagen posterior, el overhead no depende del número de tareas, ya que este se mantiene constante. Por otra parte, podemos observar que al igual que en el caso anterior, el overhead aumenta a mayor número de threads, pero eso no significa que por cada thread el overhead también aumente. Pero como se ve en la segunda columna el overhead por tarea se mantiene constante independientemente del número de threads, por tanto el overhead de crear y sincronizar por cada tarea no aumenta.



```
jordipitarchblasco - ssh -V par2118@boada.ac.upc.edu - 89x40
par2118@boada:~/Lab2/overheads$ make pi_omp_tasks
icc -Wall -O0 -std=c99 -fopenmp pi_omp_tasks.c -o pi_omp_tasks
par2118@boada:~/Lab2/overheads$ sbatch ./submit-omp.sh pi_omp_tasks 10 1
Submitted batch job 18483
par2118@boada:~/Lab2/overheads$ cat pi_omp_tasks-10-1-boada-11.txt
All overheads expressed in microseconds
Ntasks Overhead Overhead per task
 2     0.2312    0.1156
 4     0.4342    0.1085
 6     0.6446    0.1073
 8     0.8559    0.1071
10     1.0823    0.1082
12     1.2768    0.1063
14     1.4889    0.1058
16     1.6971    0.1061
18     1.9896    0.1061
20     2.1319    0.1066
22     2.3488    0.1068
24     2.5544    0.1064
26     2.7788    0.1066
28     2.9562    0.1056
30     3.1892    0.1063
32     3.3972    0.1062
34     3.6266    0.1067
36     3.8465    0.1065
38     4.0665    0.1066
40     4.2857    0.1064
42     4.4741    0.1065
44     4.6972    0.1068
46     4.9812    0.1065
48     5.1054    0.1064
50     5.3210    0.1064
52     5.6317    0.1064
54     5.8855    0.1075
56     5.9792    0.1068
58     6.1868    0.1067
60     6.3731    0.1062
62     6.6088    0.1066
64     6.8192    0.1066
par2118@boada:~/Lab2/overheads$
```

5. Conclusiones

En este segundo laboratorio, hemos aprendido a usar diferentes cláusulas de OpenMP, para la paralelización de un programa de diferentes formas, tal como hemos podido observar en todas las versiones de “pi” que hemos podido probar a lo largo de la misma.

En la primera sesión nos enfocamos en el funcionamiento de OpenMP y todas las posibilidades que nos ofrecían sus cláusulas para poder observar cual era la más óptima y por que algunas no podían ser usadas debido a su incorrecto resultado y rendimiento.

En la segunda sesión nos enfocamos más en ver los diferentes problemas que nos podían surgir, debido a que los programas pueden ser más complejos de lo que parece a simple, y muchas veces hay que tener en cuenta los overheads y la sincronización de los procesos para tener un rendimiento óptimo.