

# Informe

## Laboratorio 2 de PAR

Fall 2022-2023



**Paula Barrachina Cáceres (*par4109*)**  
**Marc Castro Chavez (*par4111*)**

# Índice

<b>A very practical introduction to OpenMP (I)</b>	<b>2</b>
Procedimiento:	2
1.3 Test your understanding	2
1.4 Observing overheads	2
<b>A very practical introduction to OpenMP (Part II)</b>	<b>5</b>
Procedimiento:	5
2.2 Test your understanding	5
2.3 Observing overheads	5
2.3.1 Thread creation and termination	5
2.3.2 Task creation and synchronisation	6
<b>Deliverable</b>	<b>7</b>
Day 1: Parallel regions and implicit tasks	7
Day 2: explicit tasks	10

# A very practical introduction to OpenMP (I)

**Fecha: 27/09/2022**

En esta segunda sesión el objetivo principal era introducir los componentes principales del modelo de programación de OpenMP y hacer un estudio de los overheads de diferentes versiones del programa pi.

## **Procedimiento:**

En la primera parte nos piden ejecutar la versión inicial del código pi, con los dos scripts introducidos en el apartado anterior.

### 1.3 Test your understanding

El test está resuelto en el siguiente apartado: **Day 1: Parallel regions and implicit tasks**

### 1.4 Observing overheads

En esta parte de la práctica se nos pedía estudiar cuatro programas de diferentes versiones de Pi, cada una usando un mecanismo de sincronización diferente.

Código	Operaciones de sincronización (critical or atomic)
<i>pi_omp_critical.c</i>	critical
<i>pi_omp_atomic.c</i>	atomic
<i>pi.omp_sumlocal.c</i>	critical
<i>pi_omp_reduction.c</i>	ninguna

Tabla 1: Tabla del tipo de operaciones de los códigos.

Lo primero que se nos pedía era compilar con un único thread y 100.000.000 iteraciones. Al compilar todos los códigos obtenemos los siguientes resultados:

Código	Tiempo
<i>pi_omp_critical.c</i>	1862585.0000 $\mu$ s
<i>pi_omp_atomic.c</i>	-5343.0000 $\mu$ s
<i>pi.omp_sumlocal.c</i>	-3837.0000 $\mu$ s
<i>pi_omp_reduction.c</i>	-694.0000 $\mu$ s

Tabla 2: Tabla de la compilación con 1 thread y 100.000.000 iteraciones

Como se puede apreciar en los resultados hay tres códigos que dan un tiempo negativo. Al ejecutar esperábamos un valor cercano a cero, no el obtenido.

Al analizar el código y buscar el motivo vemos que el tiempo negativo es debido a que al hacer la suma total el tiempo de ejecución la variable stamp1 es mucho mayor que la variable stamp2, es decir el primer *for* del código tiene un tiempo mucho más elevado que el segundo dando así un tiempo negativo.

script	<b>submit-omp.sh</b>
Resultado	3'1415926536
Tiempo ejecución	0.06960 s

Tabla 3: Obtenida de la ejecución de *pi-v0.c*



Figura 1: Línea temporal de la ejecución de *pi-v0.c*

Como se puede observar en las tablas 2 y 3 la diferencia de tiempo respecto a la ejecución en secuencial es notable en el caso de *pi\_omp\_critical*. En los otros casos no hemos creído que la comparación fuera adecuada dado a los tiempos negativos obtenidos.

Seguidamente se nos pedía compilar con 4 y 8 con el mismo número de iteraciones. Al compilar todos los códigos obtenemos los siguientes resultados:

Código	Tiempo con 4 threads	Tiempo con 8 threads
<i>pi_omp_critical.c</i>	51099664.5000 $\mu$ s	42589540.3750 $\mu$ s
<i>pi_omp_atomic.c</i>	9137319.7500 $\mu$ s	9896861.3750 $\mu$ s
<i>pi.omp_sumlocal.c</i>	5216.0000 $\mu$ s	17498.0000 $\mu$ s
<i>pi_omp_reduction.c</i>	11028.7500 $\mu$ s	16042.6250 $\mu$ s

Tabla 4: Obtenida de la ejecución de *pi-v0.c*

Como se puede deducir de los resultados de la tabla 4, tanto en el caso de *critical* cómo *atomic* a medida que se añaden threads el tiempo de ejecución va aumentando por los overheads. Esto es debido a que ambos ejecutan un thread tras otro haciendo que el tiempo de espera sea más elevado en resumen más ineficiente. Aún así conseguimos un tiempo mejor con *atomic* ya que este se aprovecha del hardware y se protegen las lecturas y escrituras en vez de una región entera del programa.

En cambio con *reduction* y *sumlocal* obtenemos mejores resultados ya que los threads se dividen el trabajo al ejecutar las iteraciones, de manera que la espera se produce al acabar de computar antes de comparar el resultado de todos los threads y obtener el resultado final.

En conclusión si queremos evitar overhead deberíamos usar en este caso *sumlocal* ya que es el que ha obtenido mejores resultados, aunque el overhead no es evitable puede reducirse considerablemente aplicando el código adecuado, como hemos podido comprobar con estas cuatro versiones del código.

# A very practical introduction to OpenMP (Part II)

Fecha: 04/10/2022

## Procedimiento:

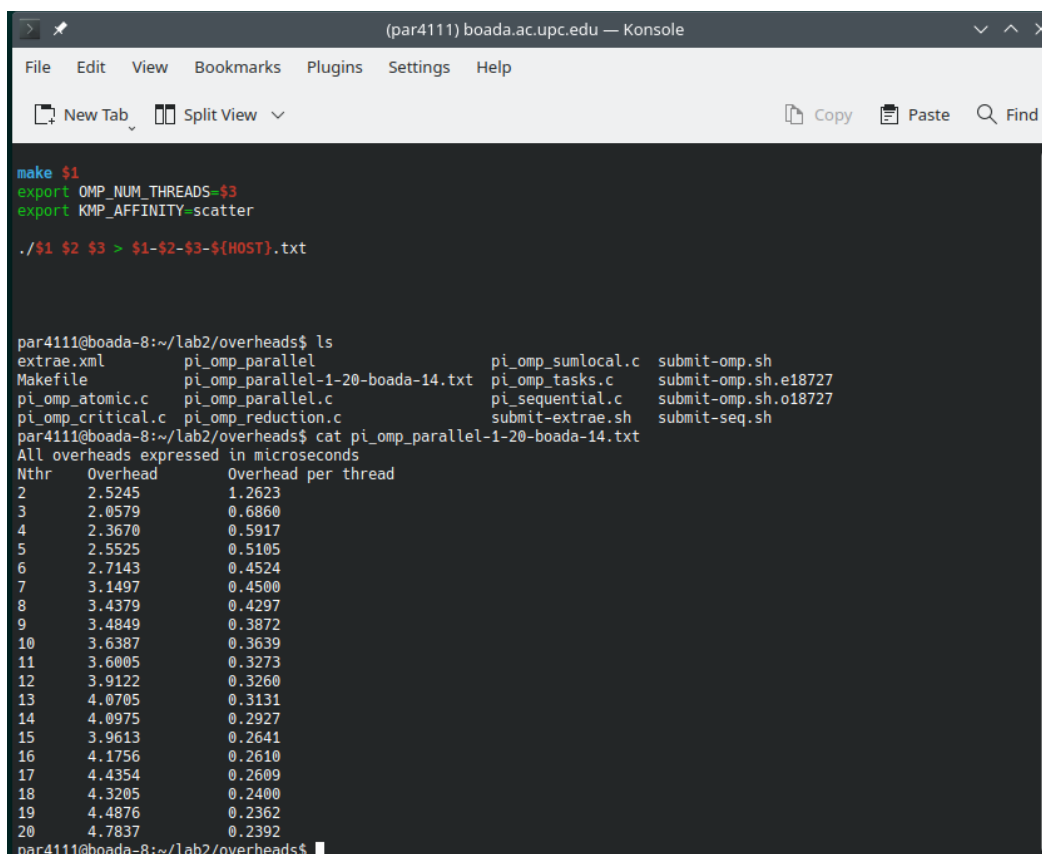
### 2.2 Test your understanding

El test está resuelto en el siguiente apartado: **Day 2: explicit tasks**

### 2.3 Observing overheads

#### 2.3.1 Thread creation and termination

En esta parte de la sesión nos piden ejecutar el binario generado por **pi\_omp\_parallel** con una sola iteración y un máximo de 20 threads.



```
(par4111) boada.ac.upc.edu — Konsole
File Edit View Bookmarks Plugins Settings Help
New Tab Split View Copy Paste Find

make $1
export OMP_NUM_THREADS=$3
export KMP_AFFINITY=scatter
./$1 $2 $3 > $1-$2-$3-{$HOST}.txt

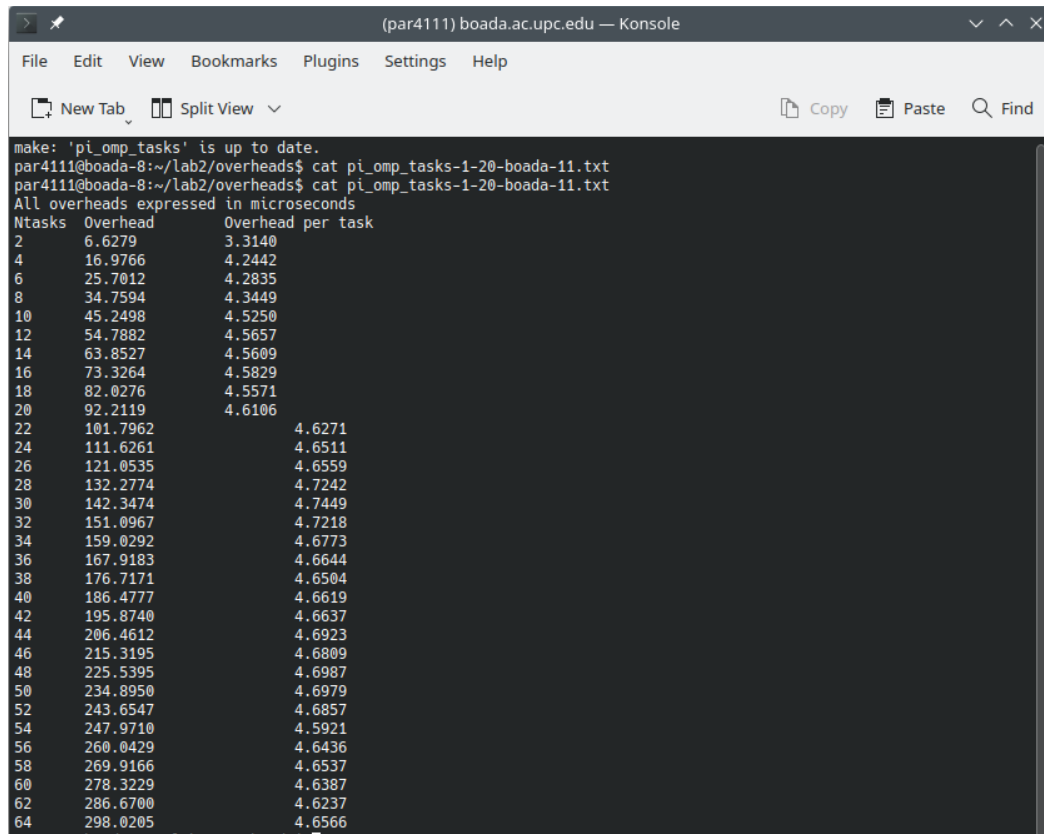
par4111@boada-8:~/lab2/overheads$ ls
extrae.xml      pi_omp_parallel      pi_omp_sumlocal.c  submit-omp.sh
Makefile        pi_omp_parallel-1-20-boada-14.txt pi_omp_tasks.c     submit-omp.sh.e18727
pi_omp_atomic.c pi_omp_parallel.c    pi_sequential.c    submit-omp.sh.o18727
pi_omp_critical.c pi_omp_reduction.c  submit-extrae.sh    submit-seq.sh
par4111@boada-8:~/lab2/overheads$ cat pi_omp_parallel-1-20-boada-14.txt
All overheads expressed in microseconds
Nthr   Overhead      Overhead per thread
2       2.5245         1.2623
3       2.0579         0.6860
4       2.3670         0.5917
5       2.5525         0.5105
6       2.7143         0.4524
7       3.1497         0.4500
8       3.4379         0.4297
9       3.4849         0.3872
10      3.6387         0.3639
11      3.6005         0.3273
12      3.9122         0.3260
13      4.0705         0.3131
14      4.0975         0.2927
15      3.9613         0.2641
16      4.1756         0.2610
17      4.4354         0.2609
18      4.3205         0.2400
19      4.4876         0.2362
20      4.7837         0.2392
par4111@boada-8:~/lab2/overheads$
```

Figura 2: Figura de la ejecución de **sbatch ./submit-omp.sh pi\_omp\_parallel 1 20**

Observando la ejecución podemos ver que el tiempo de overhead va subiendo y no es constante. Además vemos que sube de manera proporcional al número de threads usados. Podemos asumir por estos resultados que al paralelizar el programa se crea un overhead grande pero al crear threads se añade un pequeño overhead adicional por cada uno. Esto explica porque al añadir threads el overhead por thread decremента.

### 2.3.2 Task creation and synchronisation

En esta parte de la sesión nos piden ejecutar el binario generado por *pi\_omp\_parallel* con una sola iteración y un máximo de 10 threads.



```
(par4111) boada.ac.upc.edu — Konsole
File Edit View Bookmarks Plugins Settings Help
New Tab Split View Copy Paste Find
make: 'pi_omp_tasks' is up to date.
par4111@boada-8:~/lab2/overheads$ cat pi_omp_tasks-1-20-boada-11.txt
par4111@boada-8:~/lab2/overheads$ cat pi_omp_tasks-1-20-boada-11.txt
All overheads expressed in microseconds
Ntasks  Overhead      Overhead per task
2       6.6279        3.3140
4       16.9766        4.2442
6       25.7012        4.2835
8       34.7594        4.3449
10      45.2498        4.5250
12      54.7882        4.5657
14      63.8527        4.5609
16      73.3264        4.5829
18      82.0276        4.5571
20      92.2119        4.6106
22      101.7962       4.6271
24      111.6261       4.6511
26      121.0535       4.6559
28      132.2774       4.7242
30      142.3474       4.7449
32      151.0967       4.7218
34      159.0292       4.6773
36      167.9183       4.6644
38      176.7171       4.6504
40      186.4777       4.6619
42      195.8740       4.6637
44      206.4612       4.6923
46      215.3195       4.6809
48      225.5395       4.6987
50      234.8950       4.6979
52      243.6547       4.6857
54      247.9710       4.5921
56      260.0429       4.6436
58      269.9166       4.6537
60      278.3229       4.6387
62      286.6700       4.6237
64      298.0205       4.6566
```

Figura 3: Figura de la ejecución de **sbatch ./submit-omp.sh pi\_omp\_task 10 1**

Si observamos los resultados de esta ejecución podemos ver que el tiempo de overhead se mantiene constante independientemente del número de tareas.

Aún así la poca variación que hay de tiempo es debida a que probablemente la máquina que ejecuta el código tuviera más tareas que afectarán al tiempo, peor no viene dada por los overheads.

# Deliverable

## Day 1: Parallel regions and implicit tasks

### **hello.c**

1. *How many times will you see the "Hello world!" message if the program is executed with `./1.hello`?* El mensaje aparece dos veces en pantalla.
2. *Without changing the program, how to make it to print 4 times the "Hello World!" message?* Se debe incluir el siguiente comando a la hora de ejecutar:

**`OMP_NUM_THREADS=$ ./1.hello`**

### **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?*

No, como se puede ver en la imagen el orden no es el adecuado. Esto es debido a que no hay una sincronización al imprimir y esto hace que los threads impriman el valor desordenado.

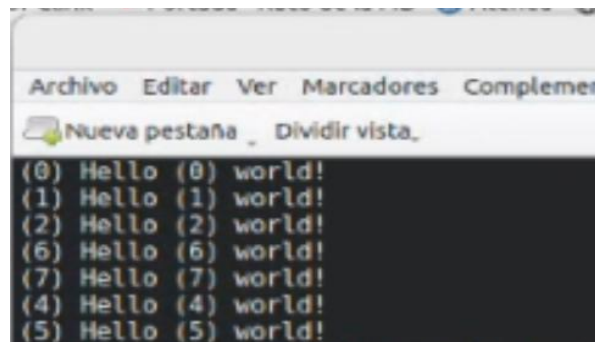


Figura: Resultado de la ejecución de **2.hello.c**

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

No, la lista va cambiando según la ejecución. Debido a que los ocho threads se ejecutan en paralelo y finalizan la tarea casi al mismo tiempo, variando la manera en que aparecen por pantalla.

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

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

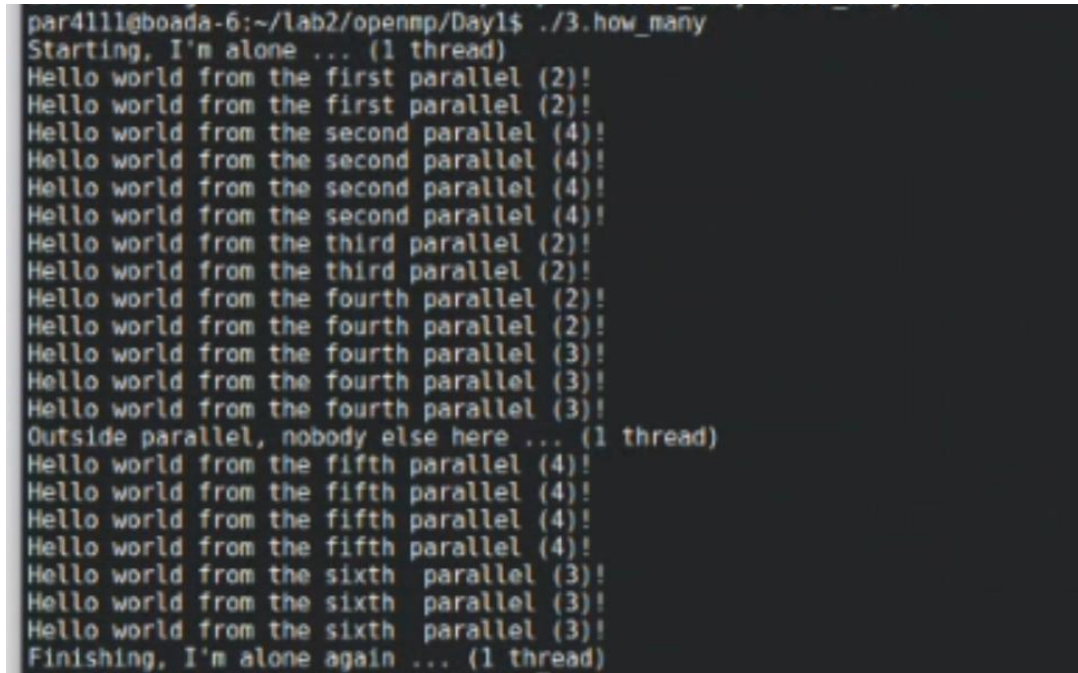
Retorna el número de threads que están en ejecución dentro de la región paralela, retornando 1 si la región del código es secuencial.



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

La primera opción consiste en especificar en las zonas paralelizadas los threads que ejecutarán con **#pragma omp parallel num\_threads(x)**, (siendo x el número de thread a ejecutar).

La segunda opción usa la función **omp\_set\_num\_threads(z)**, que determina cuántos threads ejecutarán las tareas implícitas mientras dure la ejecución.



```
par4lll@boada-6:~/lab2/openmp/Day1$ ./3.how_many
Starting, I'm alone ... (1 thread)
Hello world from the first parallel (2)!
Hello world from the first parallel (2)!
Hello world from the second parallel (4)!
Hello world from the second parallel (4)!
Hello world from the second parallel (4)!
Hello world from the second parallel (4)!
Hello world from the third parallel (2)!
Hello world from the third parallel (2)!
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)
```

Figura: Resultado de la ejecución de **3.how\_many.c**

3. Which is the lifespan for each way of defining the number of threads to be used?
  - **omp\_set\_num\_threads(num\_threads)** dura toda la ejecución o hasta que se vuelva a modificar el número de threads.
  - **num\_threads(num\_threads)** solo durante la línea de código.

#### 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)?

**Output:**

**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**

2. *Is that the value you would expect? (Execute several times if necessary)*

El valor que obtenemos en el segundo y el tercer parallel es el esperado. El segundo es correcto dado que la variable privada x es inicializada a 5, luego cada thread generado crea su variable local x pero no la inicializa. Al acabar la operación las variables locales se destruyen y se imprime el resultado de x. El tercer caso es lo mismo que el 2 pero con la diferencia de que la x ya viene inicializada.

El resultado del primero y cuarto no es lo esperable, ya que el valor va variando dependiendo del thread.

### 5.datarace.c

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

No siempre genera un resultado correcto, esto lo podemos comprobar ejecutando el comando `for i in {1..100}; do ./5.datarace; .`

Esto se debe a que es posible que quede guardado como *maxvalue* un valor que no sea el correcto ya que dejamos atrás el máximo de la secuencia dado que como varios threads se ejecutan simultáneamente no sabemos si hacen todas las comprobaciones necesarias entre los números.

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.*

**#pragma omp critical o #pragma omp parallel private(i) reduction (max:maxvalue)**

Estas dos opciones hacen que la ejecución sea correcta dado que evitan que el valor máximo pueda perderse en una versión paralela de la variable.

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; i < (i+(N/howmany)); ++i)**

### 6.datarace.c

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

No siempre genera un resultado correcto, esto lo podemos comprobar ejecutando el comando **for i in {1..100}; do ./5.datarace; .** Esto se debe a que en el *for* donde se incrementa la variable *countmax* varía el número de iteraciones en función del número id del threads.

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.*

***#pragma omp atomic o #pragma omp barrier.***

Con `atomic` el valor de la variable `countmax` no puede ser modificada simultáneamente por diferentes threads evitando que se pierdan valores. Con `barrier` conseguimos que los threads no podrán ejecutar el `if` simultáneamente y así no se pierdan los valores.

## Day 2: explicit tasks

### ***1.single.c***

1. What is the `nowait` clause doing when associated to `single`?  
Ejecuta cada thread sin esperar al anterior. Mientras el thread 0 está procesando la primera iteración, el segundo ejecuta la tercera y así sucesivamente.
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?  
Nowait reparte las iteraciones entre los 4 threads.  
La instrucción `sleep(1)` que contiene el bucle, pausa la ejecución del thread especificado y paralelamente la CPU sigue enviando threads a hacer el resto de tareas implícitas haciendo que el resultado final se imprima por los cuatro threads casi al mismo tiempo.

### ***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?  
Porque solamente hay un thread ejecutándose, dado que no hay ninguna instrucción en el código que haga que se ejecute en paralelo. Así que lo ejecuta de la forma predeterminada.

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

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

    omp_set_num_threads(6);
    printf("Starting computation of Fibonacci for numbers in linked list \n");

    p = init_list(N);
    head = p;
    #pragma omp parallel
    #pragma omp single nowait

    while (p != NULL) {
        printf("Thread %d creating task that will compute %d\n", omp_get_thread_num(), p->data);
        #pragma omp task firstprivate(p)
        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;
}
```

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

Firstprivate(p) hace que cada thread tenga una instancia privada de la variable p y la inicializa con el valor de la variable original.

Comentándolo hace que el almacenamiento de la variable p quedé compartida y hay que asegurarse de que las tareas se sincronizan adecuadamente.

### 3.taskloop.c

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

**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: (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 4  
 Loop 1: (1) gets iteration 5  
 Loop 1: (1) gets iteration 6  
 Loop 1: (1) gets iteration 7  
 Thread 0 distributing 12 iterations with num\_tasks(4) ...  
 Loop 2: (0) gets iteration 9  
 Loop 2: (0) gets iteration 10  
 Loop 2: (0) gets iteration 11  
 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

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?

Thread 0 distributing 12 iterations with grainsize(5) ...  
 Loop 1: (0) gets iteration 6  
 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 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: (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**

3. Can grainsize and num tasks be used at the same time in the same loop?  
 En primer lugar el compilador no lo permite, pero además son mutuamente excluyentes, si ejecutas ambas en un mismo loop la última en ejecutarse sobrescribe la configuración de la primera.
4. What is happening with the execution of tasks if the nogroup clause is uncommented in the first loop? Why?  
 No ejecuta el loop, solo escribe:  
 Thread 0 distributing 12 iterations with grainsize(5) ...  
 Thread 0 distributing 12 iterations with num\_tasks(5) ...

Esto se debe a que al descomentar la instrucción, los dos loops se intercalan en la ejecución ya que no se genera la región implícita del task loop.

#### **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.

```

#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
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);

// Part III
sum = 0;
#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];

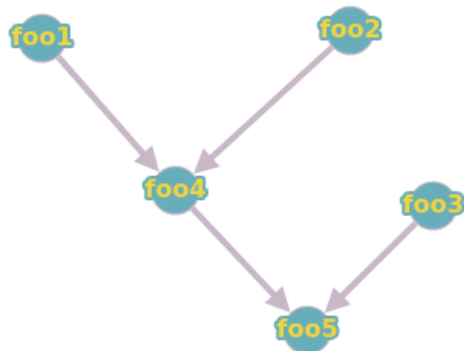
    printf("Value of sum after reduction in combined task and taskloop =
%d\n", sum);
}

return 0;
}

```

### 5.synchtasks.c

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



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.

```
#pragma omp parallel
#pragma omp single
{
    printf("Creating task foo1\n");
    #pragma omp task //depend(out:a)
    foo1();
    printf("Creating task foo2\n");
    #pragma omp task //depend(out:b)
    foo2();
    printf("Creating task foo3\n");
    #pragma omp task //depend(out:c)
    foo3();
    printf("Creating task foo4\n");
    #pragma omp taskwait //depend(in: a, b) depend(out:d)
    foo4();
    printf("Creating task foo5\n");
    #pragma omp taskwait //depend(in: c, d)
    foo5();
}
```



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.

```
#pragma omp parallel  
#pragma omp single  
{  
    printf("Creating task foo1\n");  
    #pragma omp task //depend(out:a)  
    foo1();  
    printf("Creating task foo2\n");  
    #pragma omp task //depend(out:b)  
    foo2();  
    printf("Creating task foo3\n");  
    #pragma omp task //depend(out:c)  
    foo3();  
    printf("Creating task foo4\n");  
    #pragma omp taskgroup //depend(in: a, b) depend(out:d)  
    foo4();  
    printf("Creating task foo5\n");  
    #pragma omp taskgroup //depend(in: c, d)  
    foo5();  
}
```