

**PAR Laboratory Assignment**  
**Lab 2: Brief tutorial on OpenMP**  
**programming model**

Natalia Dai  
Daniel Ruiz Jiménez

**Username:** par2306  
par2317

**Date:** 19/03

**Spring 2021-22**

# ÍNDICE

1 - Day 1 .....	pgs 1 - 8
2 - Day 2 .....	pgs 9 - 14
3 - Overheads .....	pgs 15 - 16

# 1. A very practical introduction to OpenMP

## PREGUNTAS

### 1.hello.c

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

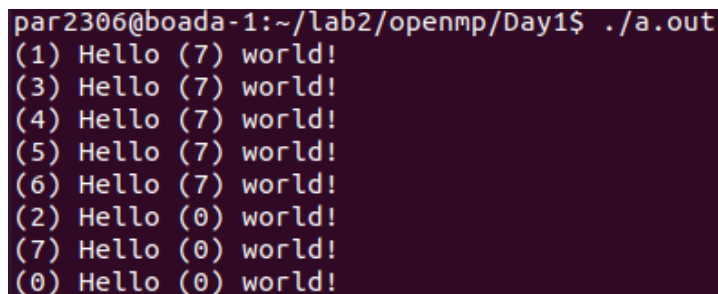
2 veces.

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

Cuando ejecutamos el programa, en la terminal, ponemos la sentencia "OMP\_NUM\_THREADS=4" y luego "./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?**



```
par2306@boada-1:~/lab2/openmp/Day1$ ./a.out
(1) Hello (7) world!
(3) Hello (7) world!
(4) Hello (7) world!
(5) Hello (7) world!
(6) Hello (7) world!
(2) Hello (0) world!
(7) Hello (0) world!
(0) Hello (0) world!
```

La ejecución del programa no es correcta porque hay data races entre los threads. Por eso cada vez que ejecutamos se ejecuta en un orden distinto. Para arreglar esto, hemos encontrado dos posibles opciones para resolver el problema:

1. ponemos la sentencia "#pragma omp critical" justo después del "#pragma omp parallel" para que el id de cada hello se corresponda con el id de cada world.
2. ponemos private(id) después de "#pragma omp parallel". El resultado es el mismo que en la opción anterior.

**2. 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, no siempre se ejecutan en el mismo orden. Esto depende del orden aleatorio en el que los threads ejecutan el programa. A veces empezará uno y a veces otro.

```
par2317@boada-1:~/lab2/openmp/Day1$ ./2.hello
(4) Hello (4) world!
(0) Hello (0) world!
(3) Hello (3) world!
(2) Hello (2) world!
(5) Hello (5) world!
(1) Hello (1) world!
(6) Hello (6) world!
(7) Hello (7) world!
```

**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?**

Cuando estamos dentro de una región paralela, omp\_get\_num\_threads devuelve el número de threads asignados en ese momento a la ejecución. Esto lo podemos ver claramente al principio, ya que tenemos 8 threads e imprime la frase 8 veces, 1 por thread, con (8) al final. Si reducimos el número de threads, se reducen también las impresiones de la frase.

Cuando estamos fuera de la región paralela, sólo se ejecuta una vez. Esto es porque lo imprime el primer thread que empieza la ejecución (ganador de la data race).

```
par2317@boada-1:~/lab2/openmp/Day1$ OMP_NUM_THREADS=8 ./3.how_many
Starting, I'm alone ... (1 thread)
Hello world from the first parallel (8)!
Hello world from the first parallel (8)!
Hello world from the first parallel (8)!
Hello world from the first parallel (8)!
Hello world from the first parallel (8)!
Hello world from the first parallel (8)!
Hello world from the first parallel (8)!
Hello world from the first parallel (8)!
Hello world from the 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 (8)!
Hello world from the third parallel (8)!
Hello world from the third parallel (8)!
Hello world from the third parallel (8)!
Hello world from the third parallel (8)!
Hello world from the third parallel (8)!
Hello world from the third parallel (8)!
Hello world from the third parallel (8)!
Hello world from the fourth parallel (2)!
Hello world from the fourth parallel (2)!
Hello world from the fourth parallel (3)!
Hello world from the fourth parallel (3)!
Hello world from the fourth parallel (3)!
Outside parallel, nobody else here ... (1 thread)
Hello world from the fifth parallel (4)!
Hello world from the fifth parallel (4)!
Hello world from the fifth parallel (4)!
Hello world from the fifth parallel (4)!
Hello world from the sixth parallel (3)!
Hello world from the sixth parallel (3)!
Hello world from the sixth parallel (3)!
Finishing, I'm alone again ... (1 thread)
```

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

La primera opción es abrir una nueva sección paralela cambiando la variable num\_threads por el nuevo número, por ejemplo, 4: num\_threads(4).

La segunda opción es mediante la sentencia omp\_set\_num\_threads, que fija el número de threads a lo que le pases por parámetro, incluso puede ser una variable.

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

Para la primera opción, el límite lo marcan los dos corchetes {} o un cambio de nuevo en el número de threads. Para la segunda opción, cuando se cambian los valores de los threads otra vez.

**4.data sharing.c**

**1. 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)**

En la primera sección (shared), el valor de la x es 120.

En la segunda sección (private), el valor es 5. En la tercera (firstprivate) también. Esto se debe a que, dentro de la sección paralela, los threads escriben una variable local copia de x.

En la cuarta, el valor es 125. El motivo es, que al final de todo, se suman las variables locales individuales de los threads (debido al + en la cláusula reduction).

**5.datarace.c**

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

No debería devolver siempre el mismo resultado. Hemos comprobado que en varias ocasiones, el resultado no es el correcto. Esto se debe a que, aunque la variable i sea privada para cada thread, la variable maxvalue es compartida entre todos los threads.

**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.**

La primera alternativa es añadir la cláusula “#pragma omp critical” antes del for, puesto que crea una región donde cada thread lo ejecuta individualmente, a cambio de tener muchos overheads de sincronización.

La segunda alternativa es la cláusula “#pragma omp barrier”, que hace que todos los threads esperen a que el resto termine de ejecutar el for, por lo tanto recorrerán todo el vector y encontrarán siempre el valor máximo, que en este caso se encuentra en la última posición del vector.

**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).**

La idea principal sería descomponer la ejecución del bucle en los threads, de forma que cada uno revise 20/8 elementos. De esos elementos, sacará el valor máximo y lo guardará en una variable local. Al terminar las iteraciones, se hará una comparación entre todas las variables locales y se elegirá el máximo de los máximos, de forma que siempre obtendremos el resultado correcto.

### **6.datarace.c**

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

No. Al igual que en el caso anterior, las data races impiden el correcto funcionamiento del programa, ya que se encuentra el valor máximo menos veces de las que aparece en el vector.

**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.**

La primera solución es añadir “reduction(+:countmax)” después del private. Este añadido hará que se almacene el resultado parcial del número de máximos, y al final de todo se sumarán.

La segunda opción es la cláusula atomic justo antes de incrementar la variable countmax, dentro del for. Cada thread ejecutará esta operación una única vez.

### **7.datarace.c**

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

La ejecución del programa es parcialmente correcta. Sí que encuentra el valor máximo del vector (maxvalue) debido al reduction, pero no cuenta bien el número de veces que aparece (countmax). Esto se debe a que la variable countmax tiene una datarace. Se suman los valores de todos los threads, y esos valores pueden ser incorrectos.

**2. Write a correct way to synchronise the execution of implicit tasks (threads) for this program.**

Para corregir el programa, lo hemos dividido en 2 fases: en la primera calcularemos maxvalue y en la segunda countmax, cada fase cuenta con una sección paralela propia.

```

int main()
{
    int i, maxvalue=0;
    int countmax = 0;

    omp_set_num_threads(8);
    #pragma omp parallel private(i) reduction(max : maxvalue)
    {
        int id = omp_get_thread_num();
        int howmany = omp_get_num_threads();
        for (i=id; i < N; i+=howmany) {
            if (vector[i] > maxvalue) maxvalue = vector[i];
        }
    }
    #pragma omp parallel private(i) reduction(+: countmax)
    {
        int id = omp_get_thread_num();
        int howmany = omp_get_num_threads();
        for (i=id; i < N; i+=howmany) {
            if (vector[i] == maxvalue) countmax++;
        }
    }

    if ((maxvalue==15) && (countmax==3))
        printf("Program executed correctly - maxvalue=%d found %d times\n", maxvalue,
            countmax);
    else printf("Sorry, something went wrong - maxvalue=%d found %d times\n", maxvalue,
        countmax);

    return 0;
}

```

### **8.barrier.c**

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

No podemos predecir la secuencia, sólo podemos saber que los mensajes de irse a dormir se imprimirán antes que los de despertarse, y que los mensajes de despertarse se mostrarán antes de cruzar la barrera. Aparte de eso, no podemos saber nada más.

## 2. A very practical introduction to OpenMP 2

### PREGUNTAS

#### 1.single.c

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

Lo que hace la cláusula nowait después del single es decirle a los threads que no tienen que esperar al final del bloque. En su lugar, pueden entrar a la siguiente iteración y continuar con la ejecución del programa.

##### 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?

```
par2306@boada-1:~/lab2/openmp/day2$ ./1.single
Thread 2 executing instance 1 of single
Thread 0 executing instance 0 of single
Thread 1 executing instance 2 of single
Thread 3 executing instance 3 of single
Thread 0 executing instance 5 of single
Thread 3 executing instance 7 of single
Thread 2 executing instance 4 of single
Thread 1 executing instance 6 of single
Thread 3 executing instance 8 of single
Thread 0 executing instance 10 of single
Thread 2 executing instance 9 of single
Thread 1 executing instance 11 of single
Thread 3 executing instance 13 of single
Thread 2 executing instance 12 of single
Thread 0 executing instance 14 of single
Thread 1 executing instance 15 of single
Thread 1 executing instance 16 of single
Thread 2 executing instance 19 of single
Thread 0 executing instance 18 of single
Thread 3 executing instance 17 of single
```

Por definición de la cláusula de nowait. Cuando el primer thread ejecuta el bucle, los siguientes hilos ejecutan las iteraciones siguientes, por lo tanto colaboran en la ejecución del programa, reduciendo su tiempo de ejecución. Además, antes del nowait hay un pragma omp parallel, por lo que todos los threads se meterán en el bucle. Se ejecutan en ráfagas de 4 ejecuciones porque estamos ejecutando el programa con 4 threads.

#### 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 en ninguna parte del código está escrita la sentencia “#pragma omp parallel”.



```

par2306@boada-1:~/lab2/openmp/Day2$ ./2.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
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

```

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

Hemos añadido dos cláusulas pragma dentro del primer bucle. Primero hemos añadido un “#pragma omp parallel”, que hace que el código se ejecute en paralelo, y luego “#pragma omp single”, que hace que cada iteración se ejecute una única vez.

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

La cláusula firstprivate crea copias privadas de la variable *p* para cada thread. Cada variable es una instancia de la original, inicializada a su mismo valor debido a que *p* se declara antes de que empiece la sección paralela. Cuando la comentamos, hacemos que la variable *p* se comparta con los 4 threads, y esto causa una data race para que modifiquen su valor. Si ejecutamos varias veces el programa, funciona correctamente, pero vemos que el código es ejecutado por un único thread, que va cambiando.

### 3.taskloop.c

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

El primer bucle (`#pragma omp taskloop grainsize (VALUE)`) reparte  $N/VALUE$  iteraciones entre el grainsize. Este número de iteraciones se ejecuta consecutivamente por cada tarea. Tenemos 4 threads,  $VALUE = 4$  y  $N = 12$ , por lo que el grainsize creará 3 tareas de 4 iteraciones cada una.

En el segundo bucle (`#pragma omp taskloop num_tasks (VALUE)`), el programa reparte  $N/VALUE$  iteraciones entre el nº de threads. Se crearán nºthreads tareas de  $N/VALUE$  iteraciones. En nuestro caso, se han creado 4 tareas de 3 iteraciones, que al final será el mismo número de iteraciones que en el bucle superior.

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

#### **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?**

Al actualizar el valor, tenemos 12 iteraciones y un grainsize de 5, así que se crearán  $12/5 \approx 2$  tareas de 6 iteraciones cada una en el primer bucle.  $12/5$  no da valor entero, por lo que el valor del grainsize pasa de ser 5 a 6.

Para el segundo bucle, tenemos 12 iteraciones y 5 tareas, por lo que se crearán 2 o 3 tareas de 3 o 2 iteraciones, uno la primera mitad y otro en la segunda, sumando 12 de todas formas.

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

Sí. El bucle se dividiría en `num_tasks` tareas y cada tarea ejecutaría  $N/\text{grainsize}$  iteraciones.

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

Si descomentamos la cláusula nogroup, se anula el taskgroup implícito que tiene el taskloop. Por lo que los threads podrán ejecutar las tareas que vayan encontrando. En nuestro caso, se ejecutan primero los dos printf fuera de los bucles y luego ejecuta los dos bucles, uno detrás de otro.

#### **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 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
for (i=0; i< SIZE/2; i++) {
    #pragma omp task firstprivate(sum)
    sum += X[i];
}

#pragma omp taskloop grainsize(BS) firstprivate(sum)
for (i=SIZE/2; i< SIZE; i++) sum += X[i];

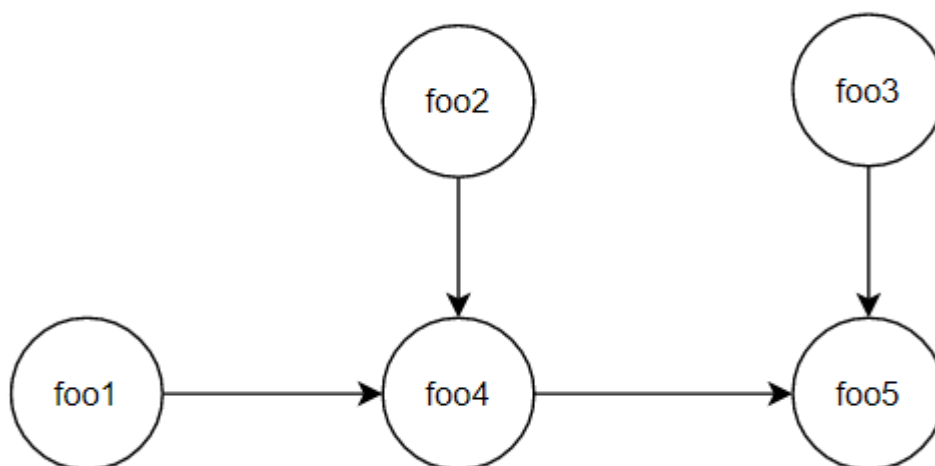
printf("Value of sum after reduction in combined task and taskloop = %d\n", sum);
}

return 0;
}

```

### 5.synctasks.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.**

```
int a, b, c, d;
int main(int argc, char *argv[]) {

    #pragma omp parallel
    #pragma omp single
    {
        printf("Creating task foo1\n");
        #pragma omp task
        foo1();

        printf("Creating task foo2\n");
        #pragma omp task
        foo2();

        printf("Creating task foo4\n");
        #pragma omp taskwait
        #pragma omp task
        foo4();

        printf("Creating task foo3\n");
        #pragma omp task
        foo3();

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

**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.**

```
int a, b, c, d;
int main(int argc, char *argv[]) {

    #pragma omp parallel
    #pragma omp single
    {
        #pragma omp taskgroup
        {

            printf("Creating task foo1\n");
            #pragma omp task
            foo1();

            printf("Creating task foo2\n");
            #pragma omp task
            foo2();

        }
        #pragma omp taskgroup
        {

            printf("Creating task foo4\n");
            #pragma omp task
            foo4();

            printf("Creating task foo3\n");
            #pragma omp task
            foo3();

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

## Overheads:

### Synchronisation overheads

**1. If executed with only 1 thread and 100.000.000 iterations, do you notice any major overhead in the execution time caused by the use of the different synchronisation mechanisms? You can compare with the baseline execution time of the sequential version in pi sequential.c.**

En cada versión se puede observar una cantidad apreciable de overhead, pero es más notable en la versión critical, que introduce overheads de hasta 2 segundos. En el resto de versiones no es muy apreciable, pero en la versión critical, el overhead dura más que la ejecución secuencial.

**2. If executed with 4 and 8 threads and the same number of iterations, do the 4 programs benefit from the use of several processors in the same way? Can you guess the reason for this behaviour?**

Ninguna versión se beneficiará del paralelismo. Cuantos más threads hay, más overhead se introduce en la ejecución de las diferentes versiones del programa. Esto se debe a que, en los dos primeros casos, se pierde el paralelismo porque se ejecuta una instrucción a la vez por thread. En el caso de sumlocal y reduction, los overheads son menores porque los threads esperan una vez el cálculo está finalizado, así que la parte previa la han podido hacer paralelizada.

	CRITICAL	ATOMIC	SUMLOCAL	REDUCTION	SEQUENTIAL
<b>1 thread (us)</b>	2617962.0000	6144.0000	4808.0000	5750.0000	1792457.104
<b>4 threads (us)</b>	37923318.7500	5208062.7500	11638.5000	12767.7500	-
<b>8 threads (us)</b>	34049198.8750	751059.0000	21311.1250	19931.1250	-

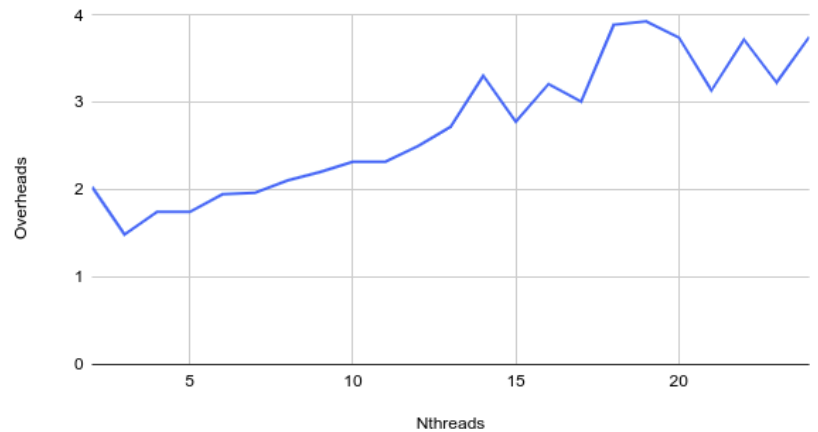
### Thread creation and termination

**How does the overhead of creating/terminating threads vary with the number of threads used? Which is the order of magnitude for the overhead of creating/terminating each individual thread in the parallel region?**

Si observamos el output del programa, no se puede ver un incremento del tiempo del overhead lineal. Sí que va aumentando con el número de threads, pero con una forma no muy bonita, como podemos ver en el gráfico. En general, este gráfico afirma que cuantos más threads hay, más overhead de sincronización hay, pero menos overhead por thread, ya que cada uno añadirá menos carga al programa.

All overheads expressed in microseconds		
Nthr	Overhead	Overhead per thread
2	2.0335	1.0168
3	1.4897	0.4966
4	1.7492	0.4373
5	1.7475	0.3495
6	1.9503	0.3251
7	1.9682	0.2812
8	2.1070	0.2634
9	2.2054	0.2450
10	2.3133	0.2313
11	2.3234	0.2112
12	2.5008	0.2084
13	2.7237	0.2095
14	3.3085	0.2363
15	2.7791	0.1853
16	3.2121	0.2008
17	3.0109	0.1771
18	3.8921	0.2162
19	3.9297	0.2068
20	3.7431	0.1872
21	3.1385	0.1495
22	3.7205	0.1691
23	3.2306	0.1405
24	3.7519	0.1563

Overheads i Nthreads



### Task creation and synchronisation

**How does the overhead of creating/synchronising tasks vary with the number of tasks created? Which is the order of magnitude for the overhead of creating/synchronising each individual task?**

Al igual que en el caso anterior, cuantas más tareas se crean, más overhead hay en total. En este caso observamos una relación lineal más bonita que el gráfico anterior. No obstante, observamos que el overhead por tarea se mantiene constante, ya que no depende del número de tareas. Las pequeñas variaciones que hay se deben a la carga de trabajo de la máquina.

All overheads expressed in microseconds		
Ntasks	Overhead	Overhead per task
2	0.1506	0.0753
4	0.4208	0.1052
6	0.6476	0.1079
8	0.8096	0.1012
10	1.0054	0.1005
12	1.2077	0.1006
14	1.4073	0.1005
16	1.6128	0.1008
18	1.7999	0.1000
20	1.9664	0.0983
22	2.2025	0.1001
24	2.3980	0.0999
26	2.6105	0.1004
28	2.7966	0.0999
30	2.9936	0.0998
32	3.2023	0.1001
34	3.4015	0.1000
36	3.6166	0.1005
38	3.8076	0.1002
40	4.0144	0.1004
42	4.1981	0.1000
44	4.4193	0.1004
46	4.6119	0.1003
48	4.8025	0.1001
50	5.0127	0.1003
52	5.2044	0.1001
54	5.3993	0.1000
56	5.6015	0.1000
58	5.8097	0.1002
60	5.9974	0.1000
62	6.2116	0.1002
64	6.4063	0.1001

Overheads i Ntasks

