

Second deliverable

Álvaro Martínez Arroyo

Daniel García Romero

par2303

Fall 2015-2016

Part I: OpenMP questionnaire

A) Basics

1. hello.c

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

24 veces, tantas como threads que ejecutan el printf.

Justificación: La arquitectura de nuestra máquina consta de 2 sockets, cada socket tiene 6 cores y por cada core tenemos 2 threads. En total, por tanto, 24 threads.

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

Ejecutando el comando `export OMP_NUM_THREADS = 4`.

Justificación: Mediante el comando anterior, indicamos que queremos que la región paralela de nuestro código sea ejecutada por 4 threads, y si cada uno de ellos ejecuta el printf, veremos 4 *Hello World!*

2. hello.c: Assuming the OMP NUM THREADS variable is set to 8 with "export OMP NUM THREADS=8"

1. Is the execution of the program correct? Which data sharing clause should be added to make it correct?

No. Hace falta añadir un `private(id)`.

Justificación: Con la directiva anterior, cada thread tendrá una copia local de la variable `id`, y de esta manera, el número de thread correspondiente a la llamada `omp_get_thread_num()` que mostrará cada uno será correcto.

2. Are the lines always printed in the same order? Could the messages appear intermixed?

No. Sí.

Justificación: El orden de impresión no se corresponde con el orden de ejecución.

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

1. How many "Hello world ..." lines are printed on the screen?

16 *Hello World!*

Justificación:

```
int main ()
{
    #pragma omp parallel
    printf("Hello world from the first parallel!\n");

    // El primer printf se muestra/ejecuta tantas veces como
    // OMP_NUM_THREADS, es decir, 8

    omp_set_num_threads(2); // OMP_NUM_THREADS = 2
    #pragma omp parallel
    printf("Hello world from the second parallel!\n");

    // El segundo printf se muestra siempre 2 veces
    // porque lo ejecutan 2 threads

    #pragma omp parallel num_threads(3)
    printf("Hello world from the third parallel!\n");

    // El tercer printf se muestra siempre 3 veces
    // porque lo ejecutan 3 threads

    #pragma omp parallel
    printf("Hello world from the fourth parallel!\n");

    // El cuarto printf se muestra siempre 2 veces
    // porque lo ejecutan 2 threads (OMP_NUM_THREADS = 2)

    srand(time(0));
    #pragma omp parallel num_threads(rand()%4+1) if (0)
    printf("Hello world from the fifth parallel!\n");

    // Con el if (0), que siempre devuelve false, fuerzas a que solo un
    // thread ejecute el quinto y último printf, por lo que éste se muestra
    // solo una vez

    return 0;
}
```

2. If the `if (0)` clause is commented in the last parallel directive, how many "Hello world ..." lines are printed on the screen?

Entre 16 y 19 *Hello World!*

Justificación: Si comentamos el `if (0)`, el quinto `printf` se mostrará tantas veces como threads lo ejecuten. El número de threads dependerá de $\text{rand()} \% 4 + 1$ (mínimo 1 y máximo 4).

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` and `firstprivate`)?

After first parallel (`shared`) x is: 8 (según ejecución, varía)

After second parallel (`private`) x is: 0

After third parallel (`first private`) x is: 0

En el caso del *shared*, todos los threads ejecutan el `++x`, pero dependiendo de la ejecución, del momento en que acceden a x, varía el resultado (ej.- con `x = 6`, el séptimo thread lee x y antes de que sume o de que actualice la variable, el último thread lee x, por lo que, tras incrementarla ambos, el valor mostrado será 7).

En el caso de *private* y *firstprivate*, como los threads tendrán una copia local que la usarán como variable temporal, al acabar la ejecución de los mismos, la variable x no se modificará.

2. What needs to be changed/added/removed in the first directive to ensure that the value after the first parallel is always 8?

Si ponemos:

```
#pragma omp parallel shared(x)
{
    #pragma omp atomic
    x++;
}
printf("After first parallel (shared) x is: %d\n",x);
```

El valor de x siempre será 8 porque *atomic* hará que solo un thread pueda ejecutar a la vez el `++x` y actualizar la posición de memoria correspondiente.

5.parallel.c

1. How many messages the program prints? Which iterations is each thread executing?

Thread ID 3 Iter 3
Thread ID 3 Iter 4
Thread ID 3 Iter 8
Thread ID 3 Iter 12
Thread ID 3 Iter 16
Thread ID 2 Iter 2

Thread ID 0 Iter 0
Thread ID 1 Iter 1

Justificación: Los threads leen y actualizan la misma variable *i*, por lo que, como hemos visto antes, el resultado que obtendremos dependerá de la ejecución, del momento en que los threads accedan a *i*.

2. What needs to be changed in the directive to ensure that each thread executes the appropriate iterations?

Si ponemos:

```
int main()
{
    int i;

    #pragma omp parallel num_threads(NUM_THREADS)
    {
        int id=omp_get_thread_num();

        #pragma omp parallel private(i)

        for (i=id; i < N; i=i+NUM_THREADS) {

            printf("Thread ID %d Iter %d\n",id,i);
        }
    }
    return 0;
}
```

Obtenemos la siguiente salida (el orden de mostrado puede variar):

```
Thread ID 1 Iter 1
Thread ID 1 Iter 5
Thread ID 1 Iter 9
Thread ID 1 Iter 13
Thread ID 1 Iter 17
Thread ID 2 Iter 2
Thread ID 2 Iter 6
Thread ID 2 Iter 10
Thread ID 2 Iter 14
Thread ID 2 Iter 18
Thread ID 0 Iter 0
Thread ID 0 Iter 4
Thread ID 0 Iter 8
Thread ID 0 Iter 12
Thread ID 0 Iter 16
Thread ID 3 Iter 3
Thread ID 3 Iter 7
Thread ID 3 Iter 11
Thread ID 3 Iter 15
Thread ID 3 Iter 19
```

Justificación: Poniendo como privada la *i*, cada thread tendrá una copia local de la variable, la cual actualizará y mostrará, y así evitaremos el problema que deriva de tener una *i* para todos y que hace que haya menos iteraciones de las deseadas.

6.datarace.c (execute several times before answering the questions)

1. Is the program always executing correctly?

No.

Justificación: La *x* es compartida por todos los threads, y todos ellos ejecutan el `++x`, pero dependiendo de la ejecución, del momento en que acceden a *x*, varía el resultado.

2. Add two alternative directives to make it correct. Which are these directives?

Alternativa 1 : *atomic*

```
int main()
{
    int i, x=0;
```

```

omp_set_num_threads(NUM_THREADS);

#pragma omp parallel private(i)
{
    int id=omp_get_thread_num();
    for (i=id; i < N; i+=NUM_THREADS) {
        #pragma omp atomic
        x++;
    }
}

if (x==N) printf("Congratulations!, program executed correctly (x = %d)\n", x);
else printf("Sorry, something went wrong, value of x = %d\n", x);

return 0;
}

```

Alternativa 2 : *reduction*

```

int main()
{
    int i, x=0;

    omp_set_num_threads(NUM_THREADS);

    #pragma omp parallel private(i) reduction(+:x)
    {
        int id=omp_get_thread_num();
        for (i=id; i < N; i+=NUM_THREADS) {
            x++;
        }
    }

    if (x==N) printf("Congratulations!, program executed correctly (x = %d)\n", x);
    else printf("Sorry, something went wrong, value of x = %d\n", x);

    return 0;
}

```

Justificación: El valor de x siempre será N porque *atomic* hará que solo un thread pueda ejecutar a la vez el $++x$ y actualizar la posición de memoria correspondiente. En el caso del *reduction*, cada thread tendrá una x privada donde irá guardando el incremento y cuando todos hayan acabado su parte, se sumarán estas x y el resultado se almacenará en x .

7.barrier.c

1. Can you predict the sequence of messages in this program? Do threads exit from the barrier in any specific order?

Sí, ya que el cálculo de los segundos que el thread estará “durmiendo” depende del id de éste.

No, el *barrier* lo único que hace es que un thread no pueda seguir con la ejecución del programa hasta que el resto de threads hayan llegado a ese punto del código.

B) Worksharing

1. for.c

1. How many iterations from the first loop are executed by each thread?

Going to distribute iterations in first loop ...

(1) gets iteration 2

(1) gets iteration 3

Going to distribute iterations in first loop ...

(0) gets iteration 0

(0) gets iteration 1

Going to distribute iterations in first loop ...

(2) gets iteration 4

(2) gets iteration 5

Going to distribute iterations in first loop ...

(5) gets iteration 10

(5) gets iteration 11

Going to distribute iterations in first loop ...

(4) gets iteration 8

(4) gets iteration 9

Going to distribute iterations in first loop ...

(7) gets iteration 14

(7) gets iteration 15

Going to distribute iterations in first loop ...

(6) gets iteration 12
(6) gets iteration 13
Going to distribute iterations in first loop ...
(3) gets iteration 6
(3) gets iteration 7

Cada thread ejecuta dos iteraciones en el primer bucle.

2. How many iterations from the second loop are executed by each thread?

Going to distribute iterations in second loop ...

(3) gets iteration 9
(4) gets iteration 11
(3) gets iteration 10
(0) gets iteration 0
(0) gets iteration 1
(0) gets iteration 2
(7) gets iteration 17
(7) gets iteration 18
(6) gets iteration 15
(6) gets iteration 16
(4) gets iteration 12
(2) gets iteration 6
(2) gets iteration 7
(2) gets iteration 8
(5) gets iteration 13
(5) gets iteration 14
(1) gets iteration 3
(1) gets iteration 4
(1) gets iteration 5

Los threads 0, 1 y 2 ejecutan tres iteraciones cada uno, mientras que los threads 3, 4, 5, 6 y 7 ejecutan dos.

3. Which directive should be added so that the first printf is executed only once by the first thread that finds it?

Si ponemos un *#pragma omp single* antes del printf, haremos que éste solo sea ejecutado por un thread (el primero que llegue a esa región de código).

2. schedule.c

1. Which iterations of the loops are executed by each thread for each schedule kind?

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: (2) gets iteration 8
Loop 1: (2) gets iteration 9
Loop 1: (2) gets iteration 10
Loop 1: (2) 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
Loop 2: (2) gets iteration 4
Loop 2: (2) gets iteration 5
Loop 2: (2) gets iteration 10
Loop 2: (1) gets iteration 2
Loop 2: (0) gets iteration 0
Loop 2: (0) gets iteration 1
Loop 2: (0) gets iteration 6
Loop 2: (0) gets iteration 7
Loop 2: (1) gets iteration 3
Loop 2: (2) gets iteration 11
Loop 2: (1) gets iteration 8
Loop 2: (1) gets iteration 9
Loop 3: (1) gets iteration 0
Loop 3: (1) gets iteration 1
Loop 3: (1) gets iteration 6
Loop 3: (1) gets iteration 7
Loop 3: (1) gets iteration 8
Loop 3: (1) gets iteration 9
Loop 3: (1) gets iteration 10
Loop 3: (1) gets iteration 11
Loop 3: (2) gets iteration 2
Loop 3: (2) gets iteration 3
Loop 3: (0) gets iteration 4
Loop 3: (0) gets iteration 5
Loop 4: (1) gets iteration 0
Loop 4: (1) gets iteration 1
Loop 4: (1) gets iteration 2
Loop 4: (1) gets iteration 3
Loop 4: (1) gets iteration 9
Loop 4: (1) gets iteration 10
Loop 4: (1) gets iteration 11
Loop 4: (0) gets iteration 4
Loop 4: (0) gets iteration 5
Loop 4: (0) gets iteration 6
Loop 4: (2) gets iteration 7
Loop 4: (2) gets iteration 8

Loop 1 → #pragma omp for schedule(static)
Loop 2 → #pragma omp for schedule(static,2)
Loop 3 → #pragma omp for schedule(dynamic,2)
Loop 4 → #pragma omp for schedule(guided,2)

Justificación: El schedule determina qué iteraciones del bucle for son ejecutadas por cada thread.

- Schedule(static): Se reparten las N iteraciones entre el número de threads. En nuestro caso, $12/3 = 4$ iteraciones por thread.

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: (2) gets iteration 8
Loop 1: (2) gets iteration 9
Loop 1: (2) gets iteration 10
Loop 1: (2) 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

- Schedule(static,2): Se reparten las N iteraciones entre el número de threads en secciones de 2 iteraciones. En nuestro caso, $12/3 = 4$ iteraciones para cada thread, pero ahora, a diferencia de antes, las 12 iteraciones se repartirán en packs de dos: el thread 0 hará las iteraciones 0,1,6,7; el thread 1, las iteraciones 2,3,8,9 y el thread 2, las iteraciones 4,5,10,11.

Loop 2: (2) gets iteration 4
Loop 2: (2) gets iteration 5
Loop 2: (2) gets iteration 10
Loop 2: (1) gets iteration 2
Loop 2: (0) gets iteration 0
Loop 2: (0) gets iteration 1
Loop 2: (0) gets iteration 6
Loop 2: (0) gets iteration 7
Loop 2: (1) gets iteration 3
Loop 2: (2) gets iteration 11
Loop 2: (1) gets iteration 8
Loop 2: (1) gets iteration 9

- `Schedule(dynamic,2)`: Se reparten 2 iteraciones a cada thread. Cuando uno de ellos acaba, coge otras 2 iteraciones más, y así hasta acabar las N iteraciones. En nuestro caso, el thread 1 se encarga de las iteraciones 0 y 1, el 2 de las iteraciones 2 y 3, y el 0 de las iteraciones 4 y 5. Como el thread 1 acaba primero las suyas, realiza las siguientes 2 iteraciones, y así hasta llegar a 12.

Loop 3: (1) gets iteration 0
Loop 3: (1) gets iteration 1
Loop 3: (1) gets iteration 6
Loop 3: (1) gets iteration 7
Loop 3: (1) gets iteration 8
Loop 3: (1) gets iteration 9
Loop 3: (1) gets iteration 10
Loop 3: (1) gets iteration 11
Loop 3: (2) gets iteration 2
Loop 3: (2) gets iteration 3
Loop 3: (0) gets iteration 4
Loop 3: (0) gets iteration 5

- `Schedule(guided,2)`: Como con `dynamic`, el reparto de iteraciones se realiza dinámicamente, durante la ejecución, pero ahora teniendo en cuenta que el número de iteraciones a hacer cada vez que estén sin trabajo decrecerá progresivamente. El límite de esta disminución lo marcará el chunk mínimo definido en la directiva, que es 2 en nuestro caso.

Loop 4: (1) gets iteration 0
Loop 4: (1) gets iteration 1
Loop 4: (1) gets iteration 2
Loop 4: (1) gets iteration 3
Loop 4: (1) gets iteration 9
Loop 4: (1) gets iteration 10
Loop 4: (1) gets iteration 11
Loop 4: (0) gets iteration 4
Loop 4: (0) gets iteration 5
Loop 4: (0) gets iteration 6
Loop 4: (2) gets iteration 7
Loop 4: (2) gets iteration 8

3. `nowait.c`

1. How does the sequence of `printf` change if the `nowait` clause is removed from the first `for` directive?

Con el *nowait*, los threads no se sincronizan al final del bucle y, por lo tanto, continúan con el loop 2 después de hacer el loop 1.

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

Tras comentar el *nowait* del primer loop, se sincronizan y por eso tenemos todos los `printf` del loop 1 seguidos de todos los `printf` del loop 2.

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

2. If the `nowait` clause is removed in the second `for` directive, will you observe any difference?

Loop 1: (0) gets iteration 0
Loop 1: (0) gets iteration 1
Loop 1: (3) gets iteration 6
Loop 1: (3) gets iteration 7
Loop 1: (1) gets iteration 2
Loop 1: (1) gets iteration 3
Loop 1: (2) gets iteration 4
Loop 1: (2) gets iteration 5
Loop 2: (3) gets iteration 6
Loop 2: (3) gets iteration 7
Loop 2: (2) gets iteration 4
Loop 2: (2) gets iteration 5
Loop 2: (0) gets iteration 0
Loop 2: (0) gets iteration 1
Loop 2: (1) gets iteration 2
Loop 2: (1) gets iteration 3

No observamos ninguna diferencia, ya que no hay otro bucle después de loop 2.

4. `collapse.c`

1. Which iterations of the loop are executed by each thread when the `collapse` clause is used?

(0) lter (0 0)
(0) lter (0 1)
(0) lter (0 2)
(0) lter (0 3)
(5) lter (3 1)
(5) lter (3 2)
(5) lter (3 3)
(3) lter (2 0)
(3) lter (2 1)
(3) lter (2 2)
(7) lter (4 2)
(7) lter (4 3)
(7) lter (4 4)
(6) lter (3 4)
(6) lter (4 0)
(6) lter (4 1)
(1) lter (0 4)
(1) lter (1 0)
(1) lter (1 1)

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

El thread 0 ejecuta 4 iteraciones y el resto de threads ejecutan 3 iteraciones.

Justificación: *collapse(2)* indica que se quieren paralelizar las iteraciones en los dos bucles for siguientes. Como $n = 5$, tenemos $5 \times 5 = 25$ iteraciones.

2. Is the execution correct if the collapse clause is removed? Which clause (different than collapse) should be added to make it correct?

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

Si comentamos el *collapse*, la ejecución no es correcta. Para que lo sea, hemos privatizado las variables *i* y *j*, por lo que cada thread tendrá una copia local de dichas variables, las cuales actualizará y mostrará, y así evitaremos el problema que deriva de tener una *i* y una *j* para todos y que hace que haya menos iteraciones de las deseadas.

C) Tasks

1. serial.c

1. Is the code printing what you expect? Is it executing in parallel?

Sí, la sucesión de Fibonacci es correcta y se ejecuta en secuencial, ya que solo lo ejecuta un thread.

Starting computation of Fibonacci for numbers in linked list
Finished computation of Fibonacci for numbers in linked list

0: 1 computed by thread 0
1: 1 computed by thread 0
2: 2 computed by thread 0
3: 3 computed by thread 0
4: 5 computed by thread 0
5: 8 computed by thread 0
6: 13 computed by thread 0
7: 21 computed by thread 0
8: 34 computed by thread 0
9: 55 computed by thread 0
10: 89 computed by thread 0
11: 144 computed by thread 0
12: 233 computed by thread 0
13: 377 computed by thread 0
14: 610 computed by thread 0
15: 987 computed by thread 0
16: 1597 computed by thread 0
17: 2584 computed by thread 0
18: 4181 computed by thread 0
19: 6765 computed by thread 0
20: 10946 computed by thread 0
21: 17711 computed by thread 0
22: 28657 computed by thread 0
23: 46368 computed by thread 0
24: 75025 computed by thread 0

2. parallel.c

1. Is the code printing what you expect? What is wrong with it?

No, la sucesión de Fibonacci no es correcta.

Starting computation of Fibonacci for numbers in linked list
Finished computation of Fibonacci for numbers in linked list

0: 4 computed by thread 2
1: 4 computed by thread 2
2: 8 computed by thread 2
3: 12 computed by thread 2
4: 20 computed by thread 2

5: 32 computed by thread 2
6: 52 computed by thread 2
7: 84 computed by thread 2
8: 136 computed by thread 2
9: 220 computed by thread 2
10: 356 computed by thread 2
11: 576 computed by thread 2
12: 932 computed by thread 2
13: 1508 computed by thread 2
14: 2440 computed by thread 2
15: 3948 computed by thread 2
16: 6388 computed by thread 2
17: 10336 computed by thread 2
18: 16724 computed by thread 2
19: 27060 computed by thread 2
20: 43784 computed by thread 1
21: 70844 computed by thread 2
22: 114628 computed by thread 3
23: 185472 computed by thread 1
24: 300100 computed by thread 2

2. Which directive should be added to make its execution correct?

Hemos añadido *#pragma omp single* antes del *#pragma omp task* para que solo un thread haga la llamada a *processwork*.

3. What would happen if the *firstprivate* clause is removed from the *task* directive? And if the *firstprivate* clause is ALSO removed from the *parallel* directive? Why are they redundant?

En lo que respecta al resultado obtenido, no afecta en nada. Son redundantes porque en las tasks, las variables son por defecto *firstprivate*.

4. Why the program breaks when variable p is not firstprivate to the task?

Si la variable p no es *firstprivate*, será compartida por todos los threads, y éstos, al hacer un acceso simultáneo a dicha variable, provocarán un *Segmentation Fault* en $p = p \rightarrow next$.

5. Why the firstprivate clause was not needed in 1.serial.c?

Porque como se ejecuta en secuencial, no tiene sentido querer definir variables privadas para cada thread, ya que no habrá más de uno.

Part II: Parallelization overheads

1. Which is the order of magnitude for the overhead associated with a parallel region (fork and join) in OpenMP? Is it constant? Reason the answer based on the results reported by the pi_omp_overhead.c code.

Enviamos el archivo que deseamos ejecutar a la cola:

```
par2303@boada-1:~/lab0/overheads$ qsub -l execution submit-omp-overhead.sh
```

Posteriormente, lo ejecutamos y abrimos el archivo que se ha generado para ver su salida:

```
par2303@boada-1:~/lab0/overheads$ ./submit-omp-overhead.sh pi_omp_overhead
par2303@boada-1:~/lab0/overheads$ emacs pi_omp_overhead_times.txt &
```

All overheads expressed in microseconds

Nthr Time Time per thread

2	2.1739	1.0870
3	1.9393	0.6464
4	2.3693	0.5923
5	2.5461	0.5092
6	2.6018	0.4336
7	2.8230	0.4033
8	3.4385	0.4298
9	3.4402	0.3822
10	3.6288	0.3629
11	3.6363	0.3306
12	3.5986	0.2999
13	4.2092	0.3238
14	3.9600	0.2829
15	4.3246	0.2883
16	4.5802	0.2863
17	4.3879	0.2581
18	4.6354	0.2575

```
19 4.6007 0.2421
20 5.0271 0.2514
21 4.8852 0.2326
22 5.0701 0.2305
23 5.2330 0.2275
24 5.2664 0.2194
Number pi after 1 iterations = 0.0000000000000000
Total execution time: 0.883859s
```

El orden de magnitud es en microsegundos y no es constante.

Podemos observar como el tiempo total de overhead va aumentando a la vez que aumentamos el número de threads, ya que cada thread añade un overhead al tiempo de ejecución total.

También vemos como disminuye el tiempo por overhead individual de manera inversamente proporcional al número de threads.

2. Which is the order of magnitude for the overhead associated with the execution of critical regions in OpenMP? How is this overhead decomposed? How and why does the overhead associated with critical increase with the number of processors? Identify at least three reasons that justify the observed performance degradation. Base your answers on the execution times reported by the pi omp.c and pi omp critical.c programs and their Paraver execution traces.

Primero de todo, abrimos el archivo submit-omp-i.sh y nos aseguramos que tenemos descomentado el programa que queremos que se ejecute junto con el número de threads deseado:

```
#setenv PROG pi_omp_i

setenv PROG pi_omp_critical_i

#setenv PROG pi_omp_lock_i

#setenv PROG pi_omp_atomic_i

#setenv PROG pi_omp_sumvector_i

#setenv PROG pi_omp_padding_i

make $PROG

#setenv OMP_NUM_THREADS 1

setenv OMP_NUM_THREADS 8
```

Una vez hecho esto, enviamos el archivo a la cola y lo ejecutamos.

Generaremos una traza tanto para OMP_NUM_THREADS 8 como para OMP_NUM_THREADS 1.

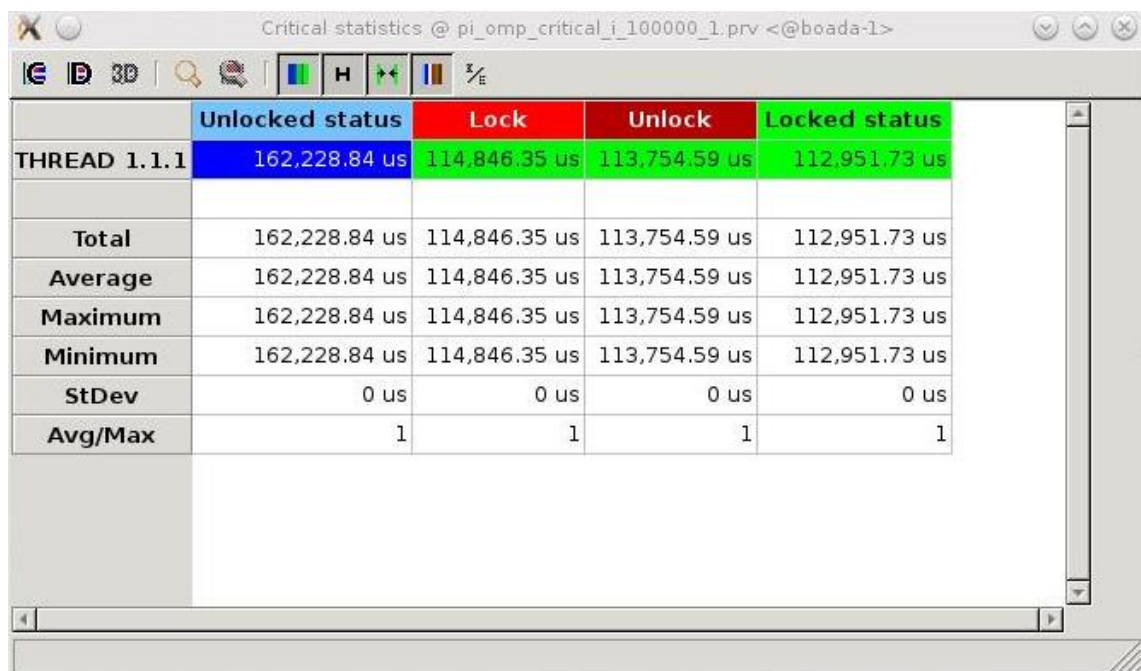
```
par2303@boada-1:~/lab0/overheads$ qsub -l execution ./submit-omp-i.sh
```

```
par2303@boada-1:~/lab0/overheads$ ./submit-omp-i.sh
```

Y podremos leer el siguiente mensaje que nos indica que la traza se ha creado correctamente.

```
mpi2prv: Congratulations! pi_omp_critical_i_100000_1.prv has been generated.
```

```
mpi2prv: Congratulations! pi_omp_critical_i_100000_8.prv has been generated.
```



	Unlocked status	Lock	Unlock	Locked status
THREAD 1.1.1	162,228.84 us	114,846.35 us	113,754.59 us	112,951.73 us
Total	162,228.84 us	114,846.35 us	113,754.59 us	112,951.73 us
Average	162,228.84 us	114,846.35 us	113,754.59 us	112,951.73 us
Maximum	162,228.84 us	114,846.35 us	113,754.59 us	112,951.73 us
Minimum	162,228.84 us	114,846.35 us	113,754.59 us	112,951.73 us
StDev	0 us	0 us	0 us	0 us
Avg/Max	1	1	1	1

Figura 1. Traza generada para pi_omp_critical_i con 1 thread

	Unlocked status	Lock	Unlock	Locked status
THREAD 1.1.1	108,787.12 us	316,635.22 us	27,334.29 us	22,676.50 us
THREAD 1.1.2	164,584.12 us	256,994.42 us	28,762.13 us	25,092.46 us
THREAD 1.1.3	89,705.85 us	331,806.58 us	29,024.24 us	24,896.46 us
THREAD 1.1.4	94,565.63 us	327,356.29 us	28,823.30 us	24,687.91 us
THREAD 1.1.5	90,868.94 us	331,590.20 us	28,531.03 us	24,442.97 us
THREAD 1.1.6	323,054.07 us	96,832.77 us	30,383.68 us	25,162.60 us
THREAD 1.1.7	341,490.12 us	78,540.26 us	30,134.63 us	25,268.12 us
THREAD 1.1.8	101,095.00 us	319,843.18 us	29,782.25 us	24,712.70 us
Total	1,314,150.86 us	2,059,598.93 us	232,775.55 us	196,939.72 us
Average	164,268.86 us	257,449.87 us	29,096.94 us	24,617.46 us
Maximum	341,490.12 us	331,806.58 us	30,383.68 us	25,268.12 us
Minimum	89,705.85 us	78,540.26 us	27,334.29 us	22,676.50 us
StDev	99,683.94 us	100,694.56 us	923.42 us	777.48 us
Avg/Max	0.48	0.78	0.96	0.97

Figura 2. Trazas generada para pi_omp_critical_i con 8 threads

El orden de magnitud es en microsegundos.

La ejecución del overhead se descompone en: unlocked status, lock, unlock y locked status.

Como podemos observar en las figuras 1 y 2, aumentando el número de threads, aumenta el tiempo de overhead.

- Fuentes de overhead

→ Creación y terminación de tareas

→ Sincronización

→ Competencia y espera por acceder a la variable sum

3. Which is the order of magnitude for the overhead associated with the execution of atomic memory accesses in OpenMP? How and why does the overhead associated with atomic increase with the number of processors? Reason the answers based on the execution times reported by the pi_omp.c and pi_omp_atomic.c programs.

Generaremos las siguientes trazas con 1 y 8 threads para el programa pi_omp_atomic tal y como hemos comentado en el apartado anterior.

	Running	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	29,597,235 ns	9,445 ns	25,663 ns	14,338 ns	2,045 ns
Total	29,597,235 ns	9,445 ns	25,663 ns	14,338 ns	2,045 ns
Average	29,597,235 ns	9,445 ns	25,663 ns	14,338 ns	2,045 ns
Maximum	29,597,235 ns	9,445 ns	25,663 ns	14,338 ns	2,045 ns
Minimum	29,597,235 ns	9,445 ns	25,663 ns	14,338 ns	2,045 ns
StDev	0 ns	0 ns	0 ns	0 ns	0 ns
Avg/Max	1	1	1	1	1

Figura 3. Traza generada para pi_omp_atomic_i con 1 thread

	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	30,214,014 ns	-	5,364,657 ns	4,043,336 ns	15,603 ns	2,028 ns
THREAD 1.1.2	9,447,987 ns	28,066,633 ns	45,280 ns	-	11,780 ns	-
THREAD 1.1.3	9,381,922 ns	28,116,602 ns	24,052 ns	-	7,783 ns	-
THREAD 1.1.4	9,426,619 ns	28,081,819 ns	14,093 ns	-	7,462 ns	-
THREAD 1.1.5	9,171,051 ns	27,990,874 ns	360,589 ns	-	8,105 ns	-
THREAD 1.1.6	9,425,242 ns	28,070,209 ns	27,059 ns	-	7,213 ns	-
THREAD 1.1.7	6,292,699 ns	28,119,495 ns	3,221,210 ns	-	6,763 ns	-
THREAD 1.1.8	9,388,001 ns	28,125,400 ns	9,144 ns	-	7,350 ns	-
Total	92,747,535 ns	196,571,032 ns	9,066,084 ns	4,043,336 ns	72,059 ns	2,028 ns
Average	11,593,441.88 ns	28,081,576 ns	1,133,260.50 ns	4,043,336 ns	9,007.38 ns	2,028 ns
Maximum	30,214,014 ns	28,125,400 ns	5,364,657 ns	4,043,336 ns	15,603 ns	2,028 ns
Minimum	6,292,699 ns	27,990,874 ns	9,144 ns	4,043,336 ns	6,763 ns	2,028 ns

Figura 4. Traza generada para pi_omp_atomic_i con 8 threads

El orden de magnitud es en nanosegundos.

Igual que pasaba con el critical, el atomic hará que solo un thread pueda actualizar a la vez la variable sum. Si hacemos una comparativa de las cuatro figuras vistas hasta ahora, vemos que con el atomic tarda menos porque solo se modifica una posición de memoria.

Si solo un thread puede ejecutar:

```
sum += 4.0/(1.0+x*x);
```

cuantos más threads se creen, tendremos más threads que tendrán que esperarse y, en consecuencia, aumentarán el overhead.

4. In the presence of false sharing (as it happens in pi omp sumvector.c), which is the additional average memory access time that you observe? What is causing this increase in the memory access time? Reason the answers based on the execution times reported by the pi omp sumvector.c and pi omp padding.c programs. Explain how padding is done in pi omp padding.c.

Generaremos las siguientes trazas con 8 threads para los programas pi_omp_sumvector y pi_omp_padding.

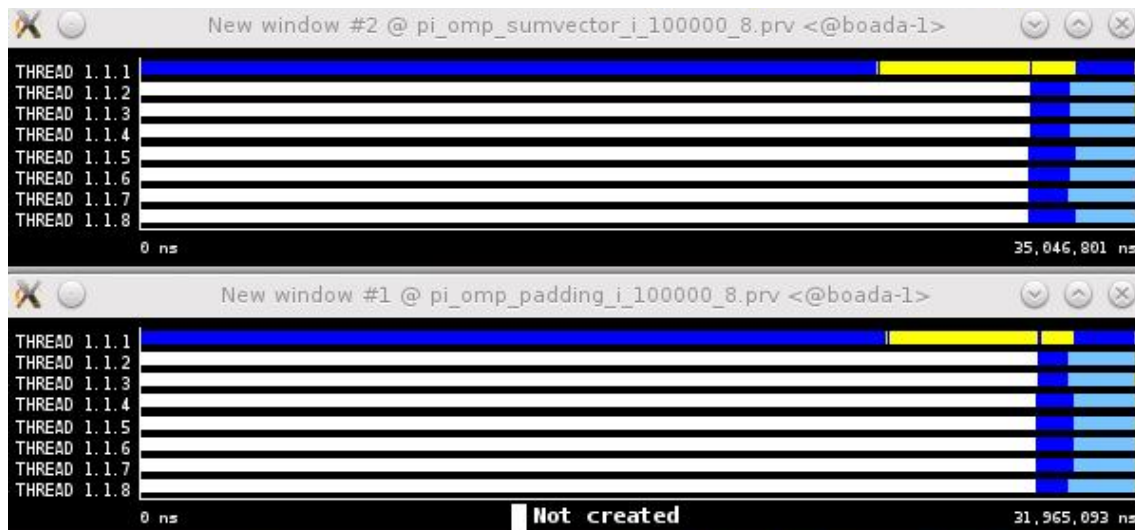


Figura 5. Trazas generadas para pi_omp_sumvector_i y pi_omp_padding_i con 8 threads

A partir de la figura 5, hemos obtenido los tiempos de ejecución para pi_omp_sumvector y pi_omp_padding de 35.046.801 ns y 31.965.093 ns, respectivamente. Por lo tanto:

Tiempo adicional de acceso a memoria = $35.046.801 - 31.965.093 = 3.081.708$ ns

Lo que provoca este incremento en el tiempo de acceso a memoria es el *false sharing*. El *false sharing* se produce cuando varios threads modifican diferentes posiciones de memoria que están en la misma línea de cache. Cuando un thread actualiza una de estas posiciones de memoria, los otros threads actualizan o invalidan su línea de cache correspondiente, dependiendo del protocolo de coherencia.

En este caso, la región de código que provoca *false sharing* es la siguiente:

```
#pragma omp parallel private(x)
{
    int myid = omp_get_thread_num();

    #pragma omp for
    for (long int i=0; i<num_steps; ++i) {
```

```

x = (i+0.5)*step;
sumvector[myid] += 4.0/(1.0+x*x);
}
}

```

En el `pi_omp_padding`, esto se soluciona convirtiendo `sumvector` en una matriz y forzando a que cada elemento se encuentre en una fila diferente, donde cada fila correspondería a una línea de cache.

En términos de código, sería lo siguiente:

```

#define CACHE_SIZE 64

double sumvector[NUMTHRDS][CACHE_SIZE/sizeof(double)];
...
#pragma omp parallel private(x)
{
    int myid = omp_get_thread_num();

    #pragma omp for
    for (long int i=0; i<num_steps; ++i) {
        x = (i+0.5)*step;
        sumvector[myid][0] += 4.0/(1.0+x*x);
    }
}

```


5. Complete the following table with the execution times of the different versions for the computation of Pi that we provide to you in this first laboratory assignment when executed with 100.000.000 iterations. The speed-up has to be computed with respect to the execution of the serial version. For each version and number of threads, how many executions have you performed?

Para obtener los tiempos, hemos hecho la media de cinco ejecuciones.

Un ejemplo de ejecución sería:

```
par2303@boada-1:~/lab0/overheads$ export OMP_NUM_THREADS=8
```

```
par2303@boada-1:~/lab0/overheads$ ./pi_omp_padding 100000000
```

version	1 processor	8 processors	speed-up
pi seq.c	0.793122 s	-	1
pi omp.c (sumlocal)	0.791727 s	0.206812 s	3.828244976
pi omp critical.c	1.830336 s	19.285296 s	0.09490836957
pi omp lock.c	1.795191 s	21.009271 s	0.08544756265
pi omp atomic.c	1.446720 s	5.906071 s	0.2449547254
pi omp sumvector.c	0.791820 s	0.666978 s	1.187175589
pi omp padding.c	0.788848 s	0.182111 s	4.331687817