Lliurament
Lab 2

Jacobo Moral
Carles Pàmies

Grup 41 – Par4101

23/10/17
Curs Tardor 2017-2018

OPTIONAL:
execution time for the multiple versions of Pi

As an optional part for this laboratory assignment, we ask to fill in a table (or draw a graph) with the execution time of the different versions of Pi explored in section 1.2 and the achieved speed-up S4 with respect to the sequential version pi-v0. Which are the most relevant conclusions you extract?

Versió	Temps (4 threads) per 100000000 iteracions (en segons)	Speed-up respecte pi_v0
Pi-v0	0.790896093	1
Pi-v1	1.766929898	0.44761034
Pi-v2	1.806194041	0,43788
Pi-v3	0.089855737	8,801843
Pi-v4	32.22464290	0,245432
Pi-v5	8.600862836	0,091955
Pi-v6	0.118499420	6,674261
Pi-v7	0.109246219	7,239574
Pi-v8	0.109775289	7,204682
Pi-v9	5.037827375	0,156992
Pi-v10	0.088923614	8,894106
Pi-v11	0.090142457	8,773847
Pi-v12	0.099904033	7,916558
Pi-v13	0.110237554	7,174471
Pi-v14	1.792722933	0,44117
Pi-v15	0.414496408	1,908089
Pi-v16	540.0033304*	0,001465
Pi-v17	0.142815448	5,537889

*S'ha calculat per 10,100, ..., 10000000 iteracions i s'ha fet una regressió lineal per veure quin temps estimat trigaria en executar-se amb el nombre d'iteracions com la resta.

En vermell, les versions que no calculen correctament pi (alguns casos el calculen bé, de casualitat, però), i en verd els que el calculen correctament sempre.

Podem observar com utilitzar la llibreria OpenMP no basta per paral·lelitzar el codi, o almenys no d'una forma correcta. Els exemples més clars són les versions que directament no calculen correctament el valor de pi. En el cas de la versió 1, a més de calcular malament el valor, triga el doble en temps. La versió 2 intenta millorar-lo però només s'aconsegueix que trigui més temps, ja que tots els threads executen totes les iteracions, i no arregla el problema del càlcul correcte de pi. A la versió 3, s'aconsegueix baixar dràsticament el temps d'execució assignant manualment les iteracions a cada thread. El problema del càlcul segueix, però, ja que ens trobem amb un *data race*.

La resta de versions afegeixen utilitats noves de OpenMP, com critical, atòmic, task i taskwait... També manualment es redueix la granularitat de les tasques en algunes versions.

Una de les utilitats amb més èxit, segons podem observar, és *Schedule* (versions 7-11). Cap d'aquestes versions té problemes de càlcul i els temps en general són bastant petits.

Com a conclusió, podem dir que l'ús d'OpenMP d'una forma raonada i sabent el que s'està fent i per a què serveix cada directiva, millora bastant els temps. A la taula anterior, podem veure com s'han aconseguit alguns speed-up de gairebé 9, només utilitzant 4 threads. Per tant, amb els 24 de boada, o els milers dels supercomputadors es poden aconseguir speed-ups molt grans (sempre que la granularitat de les tasques sigui petita per tal de poder donar una tasca a cada thread).
I no només l'ús incorrecte d'OpenMP genera temps inclús més grans que en seqüencial, sino que també pot causar errors greus en el propi càlcul.

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 cops, tants com threads té la màquina (2 sockets x 6 cores/socket x 2 threads/socket = 24 threads).

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

Afegint OMP_NUM_THREADS=4 abans de la crida al programa, en la mateixa línia de comanda (o export OMP_NUM_THREADS=4, i no fa falta escriure-ho cada cop).
Així indiquem que volem utilitzar 4 threads a les regions paral·leles (cada thread executa el printf una sola vegada.

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? (i.e., prints a sequence of "(Thid) Hello (Thid) world!" being Thid the thread identifier) Which data sharing clause should be added to make it correct?.

No executa el “Hello” i el “world” del mateix ID seguits, per tant no és correcte. Falta afegir *private(id)*. Amb aquesta sentència, cada thread utilitzarà una còpia local de la variable id.

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

Les línies no sempre s’imprimeixen en el mateix ordre, ja que els missatges poden aparèixer intercalats.
Això es deu que l’ordre en que s’imprimeixen per pantalla no es l’ordre en que s’executen.

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?

S'imprimeixen 16 línies de *Hello world ...*

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

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

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

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

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

    return 0;
}
```

Al primer printf s'imprimiran 8, un per cada thread (ja que tenim export OMP_NUM_THREADS = 8).
Al segon 2, ja que la sentència omp_set_num_threads(2) limita l'execució a només dos threads.
Al tercer 3, degut a num_threads(3).
Al quart 2, perquè omp_set_num_threads(2) afecta a tot el codi següent exepcte que digui el contrari (com el cas del tercer printf).
Al cinquè 1, perquè només s'executa un cop si la sentència if retorna fals. I if(0) sempre retorna fals.

Per tant, tenim que s'executa 8+2+3+2+1 vegades = 16.

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 i 19.

La raó és la mateixa que abans excepte el cinquè printf. Si aquest no està, la directiva num_threads(rand%4+1) ens diu que serà executat per un nombre aleatori (d'entre 1 i 4) threads.

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 (a vegades 7)
After second parallel (private) x is: 0
After third parallel (first private) x is: 0

Al Shared, això ocorre perquè el més probable i comú és que cada thread llegeixi el valor de la variable, li sumi 1, i la guardi. Després un altre thread farà el mateix, i això 8 cops farà que la x valgui 8. Però, pot passar que entre que un thread llegeix el valor i l'actualitzi, un altre faci el mateix, alhesores dos threads llegiran el mateix valor i li sumaran 1, en comptes de 2 (1 cadascun).

Amb el private, i el first private, cada thread té una còpia que no es compartida com abans, sino que quan acaba l'execució de cada thread, es borra. Per tant, el printf imprimirà el valor de la x sense cap modificació (ja que aquesta ha estat local i després s'ha esborrat).

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

```
#pragma omp parallel{
#pragma omp critical(x)
        ++x
}
```

El que fa critical es limitar el nombre de threads que executen la regió crítica (++x) a 1 alhora, per tant no pot passar l'explicat al punt anterior.
També funcionarà amb atomic en comptes de critical.

5.parallel.c

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

Thread ID 2 Iter 2
Thread ID 0 Iter 0
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
Thread ID 2 Iter 6
Thread ID 2 Iter 10
Thread ID 2 Iter 14
Thread ID 2 Iter 18
Thread ID 0 Iter 4
Thread ID 0 Iter 8
Thread ID 0 Iter 12
Thread ID 0 Iter 16
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

El text de dalt és un exemple, ja que no sempre la sortida no és sempre exactament la mateixa. Cada core sempre executa les mateixes iteracions, i sempre cadascun ho fa tot de cop i en ordre ascendent d'iteracions. El que canvia a cada execució es l'ordre de cada thread.

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

Tant sols hem de posar la i com a privada amb *#pragma omp parallel private(i)* abans del for.

6.datarace.c

1. Is the program always executing correctly?

No s'està executant correctament, ja que la variable x està compartida. Tot i que moltes vegades sí que es correcte, el resultat podrà variar alhesores del moment en que cada thread accedeix a la variable i en quin ordre.

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

Una consisteix en crear una regió de mutua exclusió entre els threads al voltant de la variable x, per a que aquesta només sigui modificada per un thread a la vegada; *#pragma omp critical(x)* a dins del for.

L’altre manera consisteix en posar *#pragma omp atomic* abans del *++x* per tal de garantir que l’acces a la variable x per a fer l’actualització es fa de manera atòmica.

7.barrier.c

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

No podem predir la seqüència sencera dels missatges, però si una part: ja que els diferents threads s’executen alhora i no sabem qui entra primer a les funcions, no sabem en quin ordre s’escriuran els missatges de “going to sleep in...”. Però la diferència de temps entre un i l’altre es mínima. I com que la diferència de temps en què estan dormint és més gran que un segon, sempre executaran l’escriptura quan s’aixequen al mateix ordre (0,1,2,3). Però, un altre cop no sabem quin ordre tindran els missatges “We are all awake!”, per la mateixa raó que abans.

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 ...
(7) gets iteration 14
(7) gets iteration 15
Going to distribute iterations in first loop ...
(3) gets iteration 6
(3) gets iteration 7
Going to distribute iterations in first loop ...
(1) gets iteration 2
(1) gets iteration 3
Going to distribute iterations in first loop ...
(5) gets iteration 10
(5) gets iteration 11
Going to distribute iterations in first loop ...
(2) gets iteration 4
Going to distribute iterations in first loop ...
(0) gets iteration 0
(0) gets iteration 1
Going to distribute iterations in first loop ...
(4) gets iteration 8
(4) gets iteration 9
(2) gets iteration 5
Going to distribute iterations in first loop ...
(6) gets iteration 12
(6) gets iteration 13
```

Cada thread executa 2 iteracions.

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

```
Going to distribute iterations in second loop ...
(6) gets iteration 15
(6) gets iteration 16
(5) gets iteration 13
(5) gets iteration 14
(3) gets iteration 9
(3) gets iteration 10
(7) gets iteration 17
(7) gets iteration 18
(4) gets iteration 11
(4) gets iteration 12
(1) gets iteration 3
(1) gets iteration 4
```

(1) gets iteration 5
(0) gets iteration 0
(0) gets iteration 1
(0) gets iteration 2
(2) gets iteration 6
(2) gets iteration 7
(2) gets iteration 8

Els primers tres threads (0, 1 i 2) executen 3 iteracions cadascun, la resta 2.

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

Hem de escriure #pragma omp single abans del printf. Així farem que s'executi pel primer dels threads que arribi.

2.schedule.c

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

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 1: (0) gets iteration 0
Loop 1: (0) gets iteration 1
Loop 1: (0) gets iteration 2
Loop 1: (0) gets iteration 3

El primer loop s'executa amb Schedule(static). Amb aquesta directiva, es reparteixen les iteracions entre els threads per igual. Per tant, com que tenim 12 iteracions i 3 threads, cadascun en farà 4 (el primer thread la 0, 1, 2, 3; el segon thread la 5, 6...).

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 2
Loop 2: (1) gets iteration 3
Loop 2: (1) gets iteration 8

Loop 2: (1) gets iteration 9
Loop 2: (2) gets iteration 4
Loop 2: (2) gets iteration 5
Loop 2: (2) gets iteration 10
Loop 2: (2) gets iteration 11

El segon loop s'executa amb Schedule(static,2). Es reparteixen igual que abans però en conjunts del segon paràmetre (2). Per tant, el thread 0 farà les iteracions 0 i 1, el thread 1 farà la 2 i la 3 i el thread 2 la 4 i la 5. I es torna a començar pel thread 0.

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

El tercer loop es fa amb Schedule(dynamic,2). Amb aquesta configuració cada thread agafa també dos iteracions (pel paràmetre). La diferència és que amb dynamic (en comptes de static), les iteracions no es reparteixen equitativament, sino que quan un thread acaba les que té assignades, se li assignen més. És per això que al nostre cas el thread 2 en fa més.

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

Loop 4 s'executa amb Schedule(guided,2). Amb guided, cada thread reb unes iteracions durant l'execució. Però el nombre d'iteracions que rebrà variarà fins a un mínim de 2 (pel paràmetre).

3.nowait.c

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

Sortida de nowait sense modificacions.

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

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

Després de treure el “nowait” del primer loop, es sincronitzen i per això surten les sortides agrupades, no com abans.

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

Loop 1: (1) gets iteration 2
Loop 1: (1) gets iteration 3

Loop 1: (3) gets iteration 6
Loop 1: (3) gets iteration 7
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: (1) gets iteration 2
Loop 2: (1) gets iteration 3
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: (3) gets iteration 6
Loop 2: (3) gets iteration 7

No observem diferència perquè és l'últim loop. Si n'hi haguès un altre sí que veuríem diferencia.

4.collapse.c

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

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

(1) Iter (0 4)
(1) Iter (1 0)
(1) Iter (1 1)

El primer thread (0) executa quatre iteracions, i la resta tres. Surten 25 línies perquè el paràmetre (2) de collapse indica en quants bucles es volen paral·lelitzar. Com que es 2, i la n es 5 → $5^2 = 25$.

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

No és correcte. La solució seria una altra: privatitzar variables “i” i “j”, així cada thread tindra les seves pròpies còpies de les variables localment. Ja que si són compartides, hi haurà moltes menys iteracions de les que volem.

5.ordered.c

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

Loop 2 - (0) gets iteration 11
Loop 2 - (1) gets iteration 12
Loop 2 - (2) gets iteration 13
Loop 2 - (3) gets iteration 14
Loop 2 - (7) gets iteration 15

Traça generada amb el codi original.

1. How can you avoid the intermixing of printf messages from the two loops?

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

Treient la directiva “nowait” al final del primer bucle. Això torna a posar la barrera implícita del programa.

2. How can you ensure that a thread always executes two consecutive iterations in order during the execution of the first loop?

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

Canviant la directiva “schedule(dynamic)” per “schedule(static,2). Això assigna al principi del programa dos iteracions consecutives a cada thread.

6.doacross.c

1. In which order are the ”Outside” and ”Inside” messages printed?

Outside from 1 executing 5

Outside from 7 executing 2
Outside from 2 executing 6
Outside from 4 executing 1
Inside from 4 executing 1
Outside from 6 executing 4
Inside from 7 executing 2
Outside from 7 executing 10
Outside from 3 executing 7
Outside from 5 executing 3
Inside from 5 executing 3
Outside from 5 executing 11
Inside from 6 executing 4
Inside from 2 executing 6
Outside from 2 executing 13
Outside from 6 executing 12
Inside from 1 executing 5
Inside from 3 executing 7
Outside from 3 executing 15
Outside from 1 executing 14
Outside from 0 executing 8
Inside from 0 executing 8
Inside from 7 executing 10
Inside from 6 executing 12
Inside from 1 executing 14
Outside from 4 executing 9
Inside from 4 executing 9
Inside from 5 executing 11
Inside from 2 executing 13
Inside from 3 executing 15

L'única regla d'ordre que hi ha és la d'executar un inside concret (from a executing b) només si ja s'ha executat abans el seu outside respectiu (from a executing b). La raó d'això és la directiva ordered depend (sink: i-2)

2. In which order are the iterations in the second loop nest executed?

Computing iteration 1 1
Computing iteration 2 1
Computing iteration 1 2
Computing iteration 1 3
Computing iteration 2 2
Computing iteration 3 1
Computing iteration 1 4
Computing iteration 2 3
Computing iteration 3 2
Computing iteration 4 1
Computing iteration 2 4

Computing iteration 3 3
Computing iteration 4 2
Computing iteration 3 4
Computing iteration 4 3
Computing iteration 4 4

Sempre en el mateix ordre. Això és degut a les dependències que es generen amb
a1[i][j] = 3.45;
i
c1[i][j] = b1[i][j] / 2.19;
respecte *b1[i][j] = a1[i][j] * (b1[i-1][j] + b1[i][j-1]);*

3. What would happen if you remove the invocation of sleep(1). Execute several times to answer in the general case.

Computing iteration 1 1
Computing iteration 1 2
Computing iteration 1 3
Computing iteration 1 4
Computing iteration 2 1
Computing iteration 2 2
Computing iteration 2 3
Computing iteration 2 4
Computing iteration 3 1
Computing iteration 3 2
Computing iteration 3 3
Computing iteration 3 4
Computing iteration 4 1
Computing iteration 4 2
Computing iteration 4 3
Computing iteration 4 4

Com que el codi ja no s'atura a cada iteració, s'executa molt més ràpid i les dependències no suposen cap problema perquè . Per tant, l'execució es fa en ordre.

C) Tasks

1.serial.c

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

Staring 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

Si, com podem observar el programa fa l'execucio dels 25 primers elements de la successió de fibonacci i en paral.lel, ja que només executa un thread.

2.parallel.c

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

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

0: 4 computed by thread 0
1: 4 computed by thread 0
2: 8 computed by thread 0

3: 12 computed by thread 0
4: 20 computed by thread 0
5: 32 computed by thread 0
6: 52 computed by thread 0
7: 84 computed by thread 0
8: 136 computed by thread 0
9: 220 computed by thread 0
10: 356 computed by thread 0
11: 576 computed by thread 0
12: 932 computed by thread 0
13: 1508 computed by thread 0
14: 2440 computed by thread 0
15: 3948 computed by thread 0
16: 6388 computed by thread 0
17: 10336 computed by thread 0
18: 16724 computed by thread 2
19: 27060 computed by thread 3
20: 43784 computed by thread 3
21: 70844 computed by thread 3
22: 114628 computed by thread 1
23: 185472 computed by thread 2
24: 300100 computed by thread 1

No, aquest cop el codi no fa bé la seva funció al utilitzar més d'un thread per a resoldre la successió.

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

Hem hagut d'afegir *#pragma omp single* abans de recorrer el struct p per a que un thread pugui crear les travesses de p i així els altres cooperin per a executarles.

Staring computation of Fibonacci for numbers in linked list
Finished computation of Fibonacci for numbers in linked list
0: 1 computed by thread 3
1: 1 computed by thread 0
2: 2 computed by thread 3
3: 3 computed by thread 0
4: 5 computed by thread 3
5: 8 computed by thread 3
6: 13 computed by thread 3
7: 21 computed by thread 3
8: 34 computed by thread 3
9: 55 computed by thread 3
10: 89 computed by thread 3
11: 144 computed by thread 3
12: 233 computed by thread 0
13: 377 computed by thread 3
14: 610 computed by thread 0

15: 987 computed by thread 3
16: 1597 computed by thread 0
17: 2584 computed by thread 3
18: 4181 computed by thread 0
19: 6765 computed by thread 3
20: 10946 computed by thread 1
21: 17711 computed by thread 0
22: 28657 computed by thread 3
23: 46368 computed by thread 1
24: 75025 computed by thread 0

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?

No té cap efecte en la succesió fibonacci ja que els atributs de data-sharing de les tasques per defecte són firstprivate.

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

Si no la declarem com a firstprivate la utilitzen tots els threads alhora i en algun moment un thread voldra accedir a un element de p al que ja ha accedit un altre, provocant un segmentation fault.

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

No era necessària perquè només teniem un thread, i per tant si treballa amb una còpia propia o la general és el mateix.

3.taskloop.c

1. Execute the program several times and make sure you are able to explain when each thread in the threads team is actually contributing to the execution of work (tasks) generated in the taskloop.

```
I am thread 3 and going to create T1 and T2
I am still thread 3 after creating T1 and T2, ready to enter in the taskwait
Thread 0 going to sleep for 5 seconds
Thread 1 finished the execution of task creating T3 and T4
Thread 1 finished the creation of all tasks in taskloop TL
Thread 1 executing loop body (1, 0)
Thread 2 going to sleep for 10 seconds
Thread 1 executing loop body (2, 0)
Thread 1 executing loop body (2, 1)
Thread 1 executing loop body (3, 0)
Thread 1 executing loop body (3, 1)
Thread 0 weaking up after a 5 seconds siesta, willing to work ...
Thread 0 executing loop body (4, 0)
I am still thread 3, but now after exiting from the taskwait
Thread 3 executing loop body (5, 0)
Thread 1 executing loop body (3, 2)
Thread 0 executing loop body (4, 1)
Thread 3 executing loop body (5, 1)
Thread 1 executing loop body (6, 0)
Thread 0 executing loop body (4, 2)
Thread 3 executing loop body (5, 2)
Thread 1 executing loop body (6, 1)
Thread 0 executing loop body (4, 3)
Thread 3 executing loop body (5, 3)
Thread 1 executing loop body (6, 2)
Thread 0 executing loop body (7, 0)
Thread 3 executing loop body (5, 4)
Thread 1 executing loop body (6, 3)
Thread 2 weaking up after a 10 seconds siesta, willing to work ...
Thread 2 executing loop body (8, 0)
Thread 0 executing loop body (7, 1)
Thread 3 executing loop body (9, 0)
Thread 1 executing loop body (6, 4)
Thread 2 executing loop body (8, 1)
Thread 0 executing loop body (7, 2)
Thread 3 executing loop body (9, 1)
Thread 1 executing loop body (6, 5)
Thread 2 executing loop body (8, 2)
Thread 0 executing loop body (7, 3)
Thread 3 executing loop body (9, 2)
Thread 2 executing loop body (8, 3)
Thread 0 executing loop body (7, 4)
Thread 3 executing loop body (9, 3)
Thread 2 executing loop body (8, 4)
Thread 0 executing loop body (7, 5)
```

```
Thread 3 executing loop body (9, 4)
Thread 2 executing loop body (8, 5)
Thread 0 executing loop body (7, 6)
Thread 3 executing loop body (9, 5)
Thread 2 executing loop body (8, 6)
Thread 3 executing loop body (9, 6)
Thread 2 executing loop body (8, 7)
Thread 3 executing loop body (9, 7)
Thread 3 executing loop body (9, 8)
```

Exemple d’execució del programa.

Al principi tant sols un dels threads contribuieix a l’execució del bucle. Quan el primer thread que s’ha posat a dormir 5 segons es desperta, s’incorpora a executar el bucle. Més tard, el thread que crea T1 i T2 i que entra al taskwait, surt del taskwait i es posa a executar també el bucle. Finalment, es desperta l’últim dels threads que s’havia posat a dormir 10 segons i s’incorpora a l’execució del bucle.

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 parallel.c code.

Nthr	Overhead	Overhead per thread
2	1.7742	0.8871
3	1.7290	0.5763
4	2.0894	0.5223
5	2.5821	0.5164
6	2.8494	0.4749
7	2.6050	0.3721
8	2.9591	0.3699
9	3.1366	0.3485
10	3.4084	0.3408
11	3.9462	0.3587
12	3.3697	0.2808
13	3.5061	0.2697
14	4.0999	0.2928
15	3.7124	0.2475
16	4.4888	0.2806
17	4.7324	0.2784
18	4.1741	0.2319
19	4.5276	0.2383
20	4.2939	0.2147
21	4.2383	0.2018
22	4.6911	0.2132
23	5.0704	0.2205
24	4.8994	0.2041

Per generar aquesta traça, hem executat l'arxiu mitjançant la cua (#qsub -l execution submit-omp.sh pi_omp_parallel 1 24) i després hem obert l'arxiu txt que s'ha generat.

Podem veure com l'overhead no és constant, i que depén dels threads. Però, també podem observar com la dependència de l'overhead amb els threads es va fent lineal a partir d'uns 12 threads. Per tant podem dir que sí que es constant l'overhead de cada thread amb el temps (evidentment l'overhead total no ho pot ser perquè dependrà del nombre de threads).

Aquest overhead per thread és de l'ordre de 1 microsegon inicialment, però amb el temps (amb molts threads) podem dir que es de l'ordre de 0.2 microsegons.

2. Which is the order of magnitude for the overhead associated with the creation of a task and its synchronization at taskwait in OpenMP? Is it constant? Reason the answer based on the results reported by the pi omp tasks.c code.

Ntasks	Overhead per task
2	0.1282
4	0.1164
6	0.1154
8	0.1149
10	0.1215
12	0.1236
14	0.1234
16	0.1224
18	0.1233
20	0.1225
22	0.1220
24	0.1212
26	0.1204
28	0.1194
30	0.1196
32	0.1194
34	0.1204
36	0.1193
38	0.1190
40	0.1188
42	0.1185
44	0.1183
46	0.1182
48	0.1185
50	0.1181
52	0.1181
54	0.1179
56	0.1177
58	0.1176
60	0.1175
62	0.1175
64	0.1172

En aquest cas tenim només un thread i múltiple tasques. L'overhead depèn del nombre de tasques (no surt a les dades però el podem imaginar) i l'overhead per task podem veure que és constant.
Podem observar com gairebé és constant, i de l'ordre del voltant de 0.1 microsegons (mínim de 0.1149 μs i el màxim de 0.1282μs).

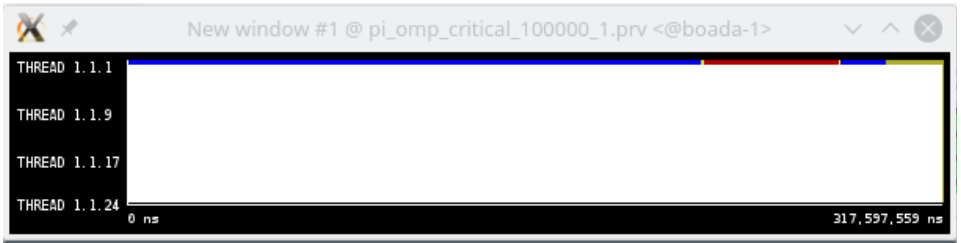
3. Which is the order of magnitude for the overhead associated with the execution of critical regions in OpenMP? How is this overhead decomposed? How and why does the overhead associated with critical increase with the number of processors? Identify at least three reasons that justify the observed performance 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.

Per generar les traces dels exercicis següents, hem seguit les passes descrites al primer entregable.

Critical statistics @ pi_omp_critical_100000_8.prv <@boada-1>

	Unlocked status	Lock	Unlock	Locked status
THREAD 1.1.1	270,674.24 us	43,447.96 us	3,594.62 us	3,090.97 us
THREAD 1.1.2	281,898.17 us	33,363.90 us	2,848.56 us	2,697.16 us
THREAD 1.1.3	279,302.53 us	35,707.99 us	3,059.89 us	2,737.39 us
THREAD 1.1.4	282,611.30 us	32,047.39 us	3,484.92 us	2,664.19 us
THREAD 1.1.5	268,350.18 us	46,900.41 us	2,994.94 us	2,562.27 us
THREAD 1.1.6	267,568.08 us	45,627.65 us	3,712.47 us	3,899.59 us
THREAD 1.1.7	271,674.84 us	41,921.01 us	3,303.60 us	3,908.35 us
THREAD 1.1.8	278,666.25 us	36,071.13 us	2,829.76 us	3,240.66 us
THREAD 1.1.9	320,807.80 us	-	-	-
THREAD 1.1.10	320,807.80 us	-	-	-
THREAD 1.1.11	320,807.80 us	-	-	-
THREAD 1.1.12	320,807.80 us	-	-	-
THREAD 1.1.13	320,807.80 us	-	-	-
THREAD 1.1.14	320,807.80 us	-	-	-
THREAD 1.1.15	320,807.80 us	-	-	-
THREAD 1.1.16	320,807.80 us	-	-	-
THREAD 1.1.17	320,807.80 us	-	-	-
THREAD 1.1.18	320,807.80 us	-	-	-
THREAD 1.1.19	320,807.80 us	-	-	-
THREAD 1.1.20	320,807.80 us	-	-	-
THREAD 1.1.21	320,807.80 us	-	-	-
THREAD 1.1.22	320,807.80 us	-	-	-
THREAD 1.1.23	320,807.80 us	-	-	-
THREAD 1.1.24	320,807.80 us	-	-	-
Total	7,333,670.41 us	315,087.44 us	25,828.76 us	24,800.59 us
Average	305,569.60 us	39,385.93 us	3,228.60 us	3,100.07 us
Maximum	320,807.80 us	46,900.41 us	3,712.47 us	3,908.35 us
Minimum	267,568.08 us	32,047.39 us	2,829.76 us	2,562.27 us
StDev	21,806.30 us	5,396.61 us	321.32 us	510.58 us
Avg/Max	0.95	0.84	0.87	0.79

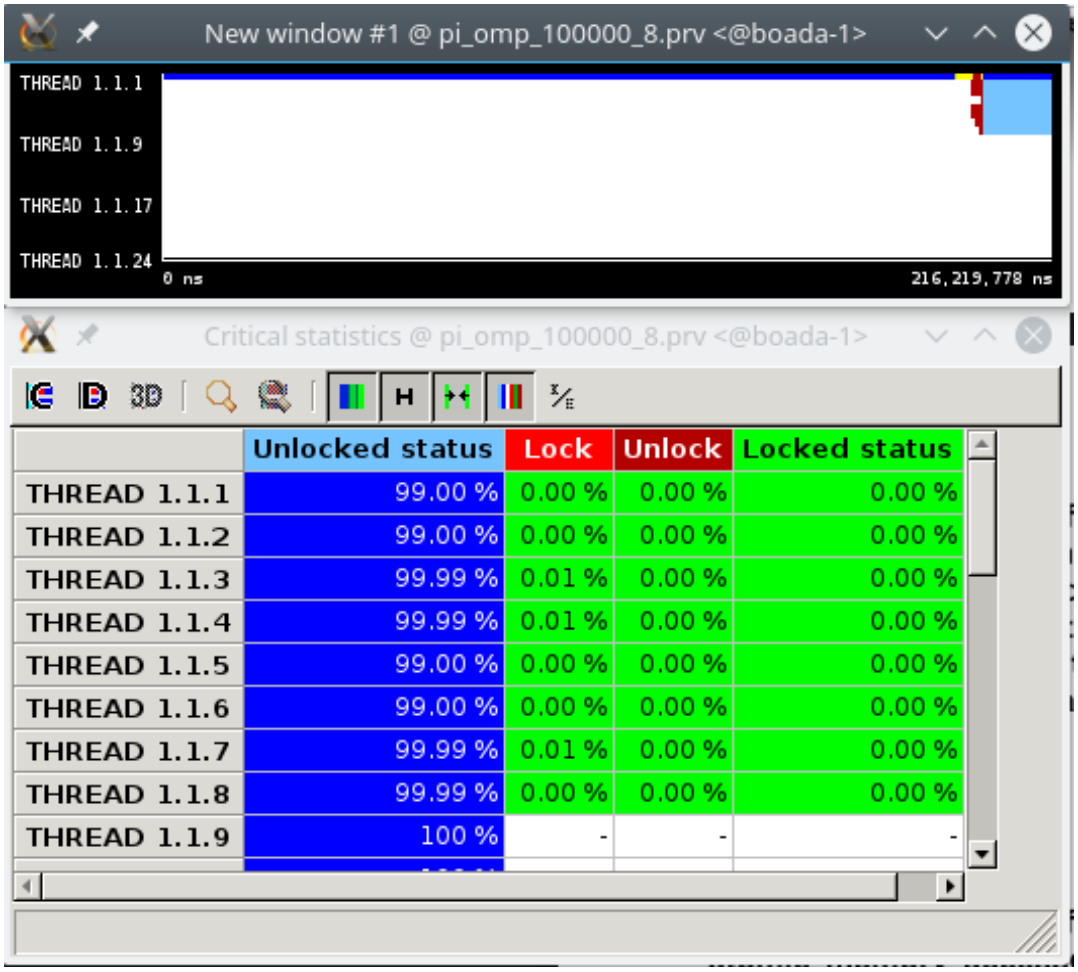
Imatge 1: Pi_omp_critical, 8 threads



Critical statistics @ pi_omp_critical_100000_1.prv <@boada-1>

	Unlocked status	Lock	Unlock	Locked status
THREAD 1.1.1	277,804.63 us	13,765.99 us	12,426.36 us	13,600.57 us
THREAD 1.1.2	317,597.56 us	-	-	-
THREAD 1.1.3	317,597.56 us	-	-	-
THREAD 1.1.4	317,597.56 us	-	-	-
THREAD 1.1.5	317,597.56 us	-	-	-
THREAD 1.1.6	317,597.56 us	-	-	-
THREAD 1.1.7	317,597.56 us	-	-	-
THREAD 1.1.8	317,597.56 us	-	-	-
THREAD 1.1.9	317,597.56 us	-	-	-
THREAD 1.1.10	317,597.56 us	-	-	-
THREAD 1.1.11	317,597.56 us	-	-	-
THREAD 1.1.12	317,597.56 us	-	-	-
THREAD 1.1.13	317,597.56 us	-	-	-
THREAD 1.1.14	317,597.56 us	-	-	-
THREAD 1.1.15	317,597.56 us	-	-	-
THREAD 1.1.16	317,597.56 us	-	-	-
THREAD 1.1.17	317,597.56 us	-	-	-
THREAD 1.1.18	317,597.56 us	-	-	-
THREAD 1.1.19	317,597.56 us	-	-	-
THREAD 1.1.20	317,597.56 us	-	-	-
THREAD 1.1.21	317,597.56 us	-	-	-
THREAD 1.1.22	317,597.56 us	-	-	-
THREAD 1.1.23	317,597.56 us	-	-	-
THREAD 1.1.24	317,597.56 us	-	-	-
Total	7,582,548.49 us	13,765.99 us	12,426.36 us	13,600.57 us
Average	315,939.52 us	13,765.99 us	12,426.36 us	13,600.57 us
Maximum	317,597.56 us	13,765.99 us	12,426.36 us	13,600.57 us
Minimum	277,804.63 us	13,765.99 us	12,426.36 us	13,600.57 us
StDev	7,951.67 us	0 us	0 us	0 us
Avg/Max	0.99	1	1	1

Imatge 2: Pi_omp_critical 1 thread



Imatge 3: Pi_omp amb 8 threads

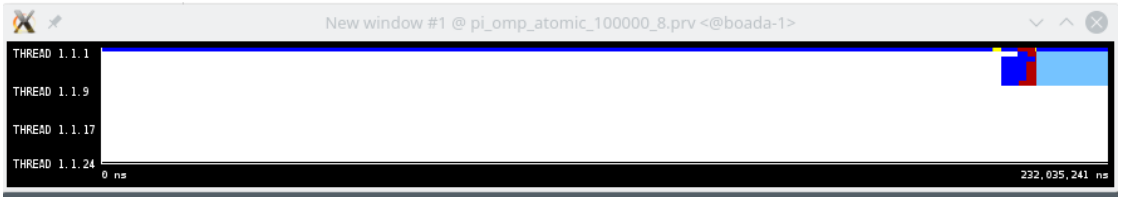
Com podem observar a les imatges 1 i 2:

1. L'ordre de magnitut és de desenes de miler de microsegons.
2. L'execució de l'overhead es descomosa en Unlocked status, Lock, Unlock i Locked status
3. El nombre de threads fa augmentar el temps d'overhead.

Les tres raons que justifiquen la degradació de rendiment són:

1. Sincronització
2. Creació/destrucció de tasques
3. Competència entre threads per accedir a la regió crítica.

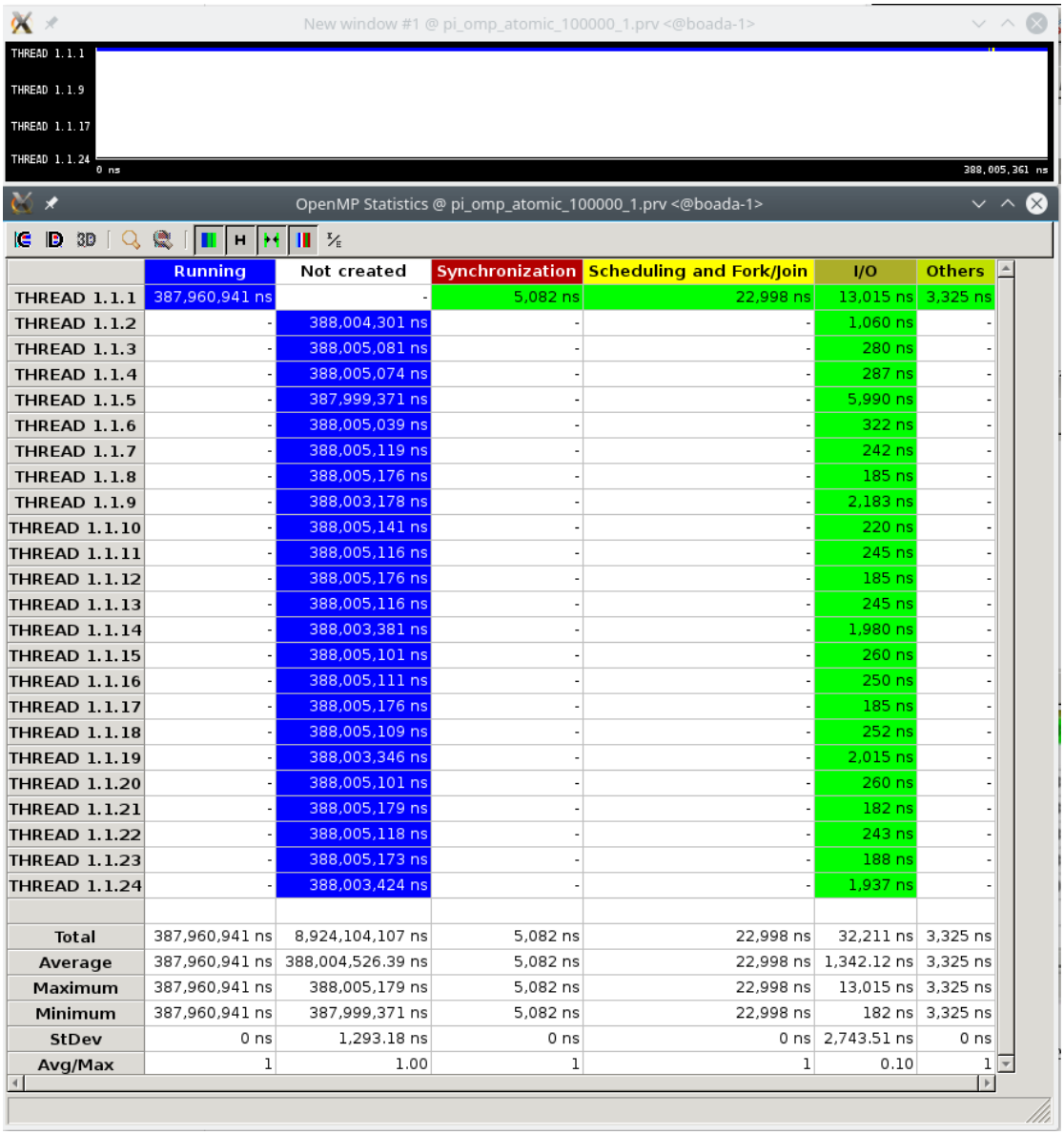
4. Which is the order of magnitude for the overhead associated with the execution of atomic memory accesses in OpenMP? How and why does the overhead associated with atomic increase with the number of processors? Reason the answers based on the execution times reported by the pi omp.c and pi omp atomic.c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



OpenMP Statistics @ pi_omp_atomic_100000_8.prv <@boada-1>

	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	226,596,013 ns	-	3,882,352 ns	1,540,993 ns	12,613 ns	3,270 ns
THREAD 1.1.2	2,423,126 ns	210,821,025 ns	1,858,571 ns	-	12,657 ns	-
THREAD 1.1.3	8,005,748 ns	206,980,928 ns	15,570 ns	-	9,025 ns	-
THREAD 1.1.4	6,118,813 ns	206,980,873 ns	2,006,658 ns	-	6,190 ns	-
THREAD 1.1.5	6,211,874 ns	206,980,893 ns	1,912,662 ns	-	6,178 ns	-
THREAD 1.1.6	6,223,216 ns	206,980,845 ns	1,902,406 ns	-	5,955 ns	-
THREAD 1.1.7	6,220,072 ns	206,980,963 ns	1,903,857 ns	-	10,097 ns	-
THREAD 1.1.8	4,289,909 ns	206,979,421 ns	3,801,699 ns	-	5,760 ns	-
THREAD 1.1.9	-	232,034,870 ns	-	-	371 ns	-
THREAD 1.1.10	-	232,030,591 ns	-	-	4,650 ns	-
THREAD 1.1.11	-	232,034,714 ns	-	-	527 ns	-
THREAD 1.1.12	-	232,035,039 ns	-	-	202 ns	-
THREAD 1.1.13	-	232,034,941 ns	-	-	300 ns	-
THREAD 1.1.14	-	232,035,061 ns	-	-	180 ns	-
THREAD 1.1.15	-	232,033,121 ns	-	-	2,120 ns	-
THREAD 1.1.16	-	232,034,812 ns	-	-	429 ns	-
THREAD 1.1.17	-	232,035,061 ns	-	-	180 ns	-
THREAD 1.1.18	-	232,034,946 ns	-	-	295 ns	-
THREAD 1.1.19	-	232,033,406 ns	-	-	1,835 ns	-
THREAD 1.1.20	-	232,034,932 ns	-	-	309 ns	-
THREAD 1.1.21	-	232,035,063 ns	-	-	178 ns	-
THREAD 1.1.22	-	232,035,061 ns	-	-	180 ns	-
THREAD 1.1.23	-	232,035,061 ns	-	-	180 ns	-
THREAD 1.1.24	-	232,033,400 ns	-	-	1,841 ns	-
Total	266,088,771 ns	5,165,255,027 ns	17,283,775 ns	1,540,993 ns	82,252 ns	3,270 ns
Average	33,261,096.38 ns	224,576,305.52 ns	2,160,471.88 ns	1,540,993 ns	3,427.17 ns	3,270 ns
Maximum	226,596,013 ns	232,035,063 ns	3,882,352 ns	1,540,993 ns	12,657 ns	3,270 ns
Minimum	2,423,126 ns	206,979,421 ns	15,570 ns	1,540,993 ns	178 ns	3,270 ns
StDev	73,089,979.37 ns	11,299,895.15 ns	1,149,334.78 ns	0 ns	4,081.80 ns	0 ns
Avg/Max	0.15	0.97	0.56	1	0.27	1

Imatge 4: Pi_omp_atomic amb 8 threads



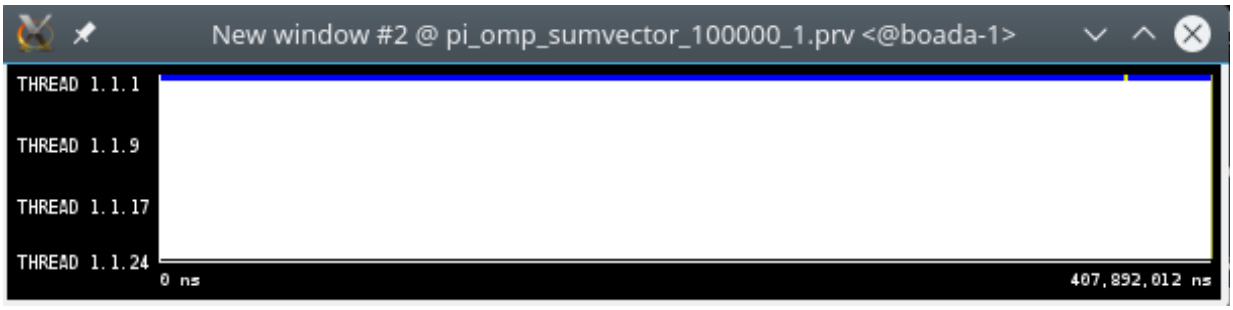
OpenMP Statistics @ pi_omp_atomic_100000_1.prv <@boada-1>

	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	387,960,941 ns	-	5,082 ns	22,998 ns	13,015 ns	3,325 ns
THREAD 1.1.2	-	388,004,301 ns	-	-	1,060 ns	-
THREAD 1.1.3	-	388,005,081 ns	-	-	280 ns	-
THREAD 1.1.4	-	388,005,074 ns	-	-	287 ns	-
THREAD 1.1.5	-	387,999,371 ns	-	-	5,990 ns	-
THREAD 1.1.6	-	388,005,039 ns	-	-	322 ns	-
THREAD 1.1.7	-	388,005,119 ns	-	-	242 ns	-
THREAD 1.1.8	-	388,005,176 ns	-	-	185 ns	-
THREAD 1.1.9	-	388,003,178 ns	-	-	2,183 ns	-
THREAD 1.1.10	-	388,005,141 ns	-	-	220 ns	-
THREAD 1.1.11	-	388,005,116 ns	-	-	245 ns	-
THREAD 1.1.12	-	388,005,176 ns	-	-	185 ns	-
THREAD 1.1.13	-	388,005,116 ns	-	-	245 ns	-
THREAD 1.1.14	-	388,003,381 ns	-	-	1,980 ns	-
THREAD 1.1.15	-	388,005,101 ns	-	-	260 ns	-
THREAD 1.1.16	-	388,005,111 ns	-	-	250 ns	-
THREAD 1.1.17	-	388,005,176 ns	-	-	185 ns	-
THREAD 1.1.18	-	388,005,109 ns	-	-	252 ns	-
THREAD 1.1.19	-	388,003,346 ns	-	-	2,015 ns	-
THREAD 1.1.20	-	388,005,101 ns	-	-	260 ns	-
THREAD 1.1.21	-	388,005,179 ns	-	-	182 ns	-
THREAD 1.1.22	-	388,005,118 ns	-	-	243 ns	-
THREAD 1.1.23	-	388,005,173 ns	-	-	188 ns	-
THREAD 1.1.24	-	388,003,424 ns	-	-	1,937 ns	-
Total	387,960,941 ns	8,924,104,107 ns	5,082 ns	22,998 ns	32,211 ns	3,325 ns
Average	387,960,941 ns	388,004,526.39 ns	5,082 ns	22,998 ns	1,342.12 ns	3,325 ns
Maximum	387,960,941 ns	388,005,179 ns	5,082 ns	22,998 ns	13,015 ns	3,325 ns
Minimum	387,960,941 ns	387,999,371 ns	5,082 ns	22,998 ns	182 ns	3,325 ns
StDev	0 ns	1,293.18 ns	0 ns	0 ns	2,743.51 ns	0 ns
Avg/Max	1	1.00	1	1	0.10	1

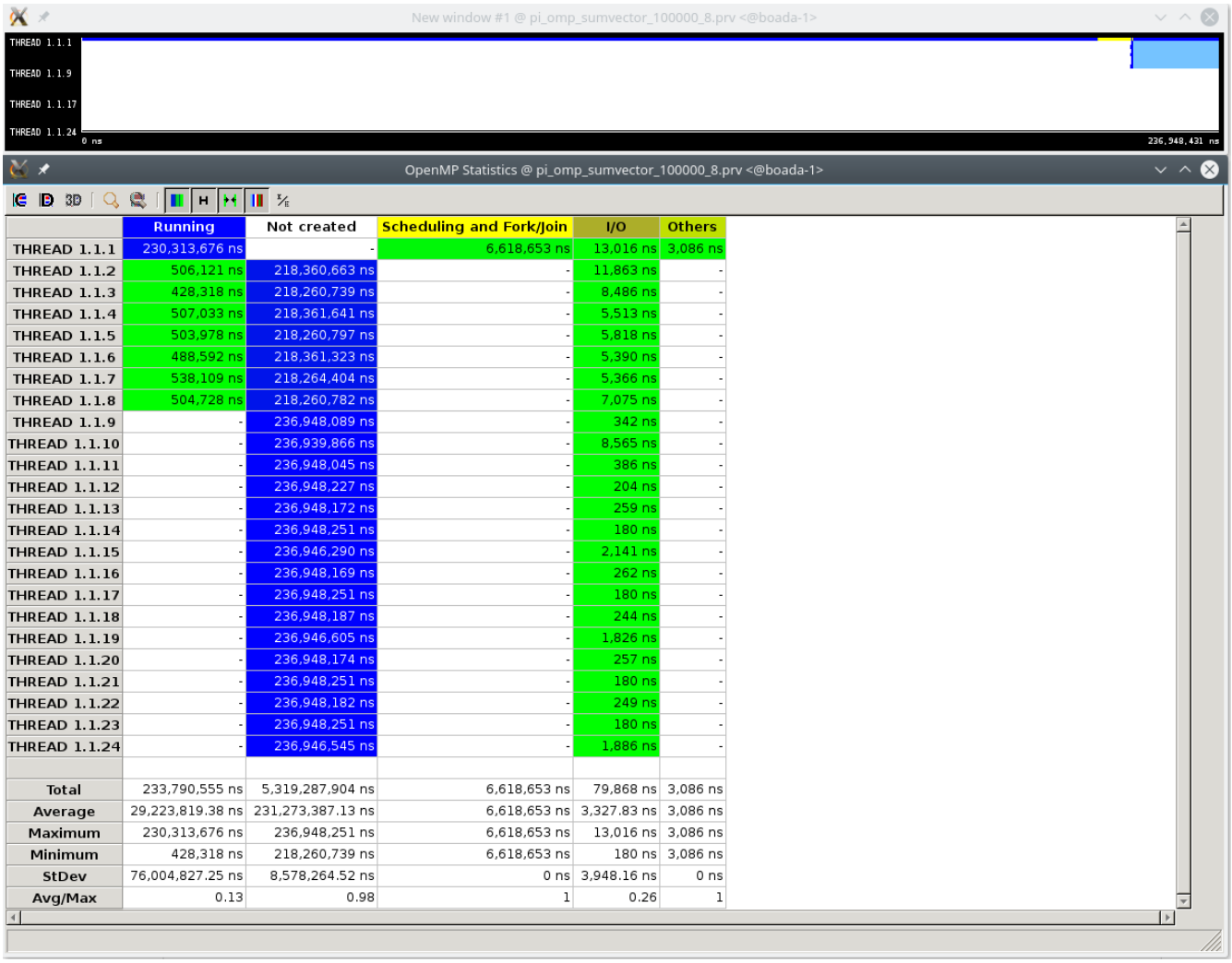
Imatge 5: Pi_omp_atomic amb 1 thread

L'ordre de magnitut és de nanosegons.
Podem observar a les imatges 4 i 5, juntament amb les imatges anteriors, que, l'overhead (sync, schedulling/fork/join/i/o...) augmenta si augmentem el nombre de threads. Això ocorre perquè només un thread pot executar una regió atomica alhora. Per tant, quants més threads, més s'hauran d'esperar, i així augmentant el temps.

5. In the presence of false sharing (as it happens in pi omp sumvector.c), which is the additional average time for each individual access to memory 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.

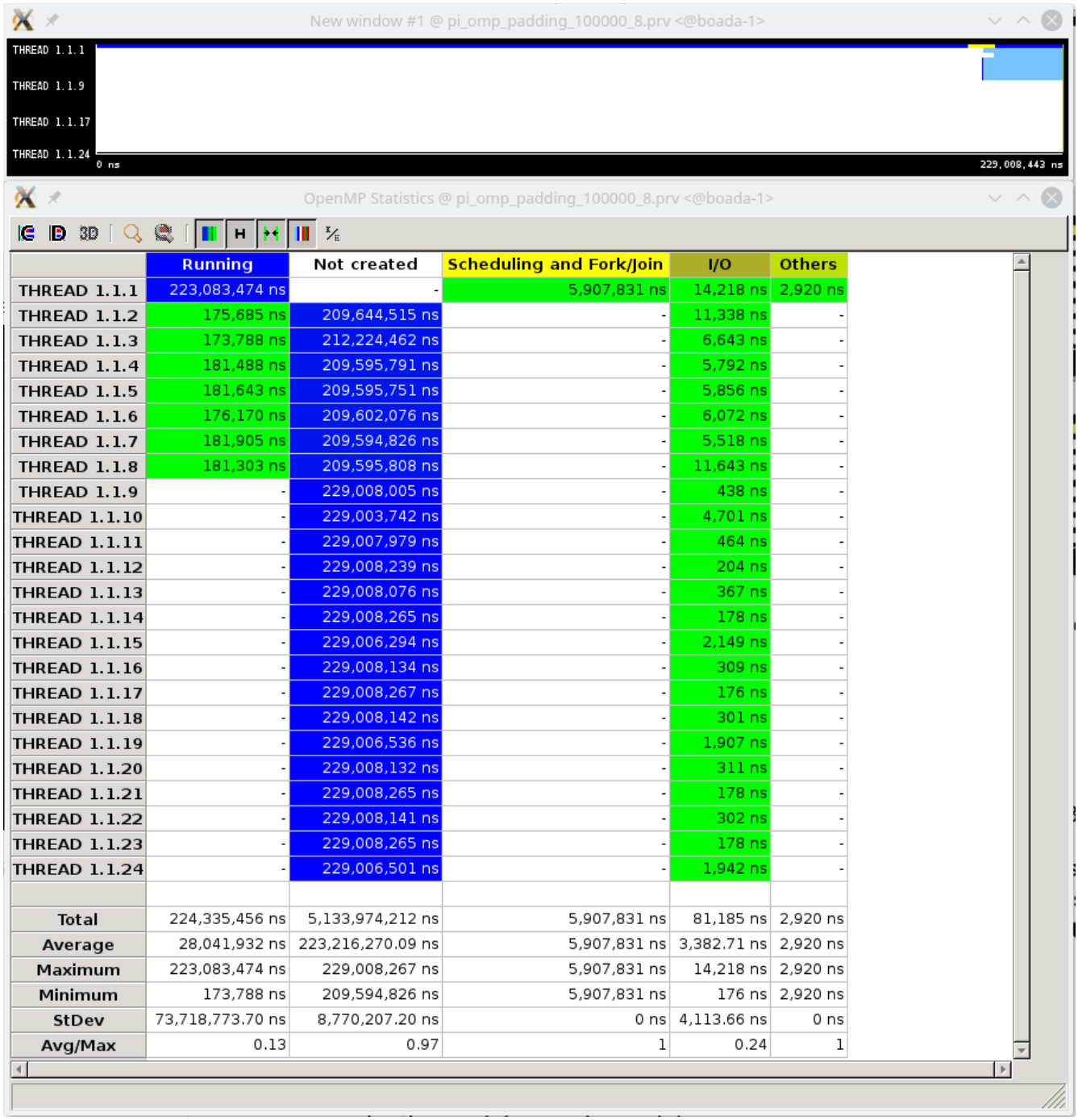


Imatge 6: Pi_omp_sumvector amb 1 thread



	Running	Not created	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	230,313,676 ns	-	6,618,653 ns	13,016 ns	3,086 ns
THREAD 1.1.2	506,121 ns	218,360,663 ns	-	11,863 ns	-
THREAD 1.1.3	428,318 ns	218,260,739 ns	-	8,486 ns	-
THREAD 1.1.4	507,033 ns	218,361,641 ns	-	5,513 ns	-
THREAD 1.1.5	503,978 ns	218,260,797 ns	-	5,818 ns	-
THREAD 1.1.6	488,592 ns	218,361,323 ns	-	5,390 ns	-
THREAD 1.1.7	538,109 ns	218,264,404 ns	-	5,366 ns	-
THREAD 1.1.8	504,728 ns	218,260,782 ns	-	7,075 ns	-
THREAD 1.1.9	-	236,948,089 ns	-	342 ns	-
THREAD 1.1.10	-	236,939,866 ns	-	8,565 ns	-
THREAD 1.1.11	-	236,948,045 ns	-	386 ns	-
THREAD 1.1.12	-	236,948,227 ns	-	204 ns	-
THREAD 1.1.13	-	236,948,172 ns	-	259 ns	-
THREAD 1.1.14	-	236,948,251 ns	-	180 ns	-
THREAD 1.1.15	-	236,946,290 ns	-	2,141 ns	-
THREAD 1.1.16	-	236,948,169 ns	-	262 ns	-
THREAD 1.1.17	-	236,948,251 ns	-	180 ns	-
THREAD 1.1.18	-	236,948,187 ns	-	244 ns	-
THREAD 1.1.19	-	236,946,605 ns	-	1,826 ns	-
THREAD 1.1.20	-	236,948,174 ns	-	257 ns	-
THREAD 1.1.21	-	236,948,251 ns	-	180 ns	-
THREAD 1.1.22	-	236,948,182 ns	-	249 ns	-
THREAD 1.1.23	-	236,948,251 ns	-	180 ns	-
THREAD 1.1.24	-	236,946,545 ns	-	1,886 ns	-
Total	233,790,555 ns	5,319,287,904 ns	6,618,653 ns	79,868 ns	3,086 ns
Average	29,223,819.38 ns	231,273,387.13 ns	6,618,653 ns	3,327.83 ns	3,086 ns
Maximum	230,313,676 ns	236,948,251 ns	6,618,653 ns	13,016 ns	3,086 ns
Minimum	428,318 ns	218,260,739 ns	6,618,653 ns	180 ns	3,086 ns
StDev	76,004,827.25 ns	8,578,264.52 ns	0 ns	3,948.16 ns	0 ns
Avg/Max	0.13	0.98	1	0.26	1

Imatge 7: Pi_omp_sumvector amb 8 threads



	Running	Not created	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	223,083,474 ns	-	5,907,831 ns	14,218 ns	2,920 ns
THREAD 1.1.2	175,685 ns	209,644,515 ns	-	11,338 ns	-
THREAD 1.1.3	173,788 ns	212,224,462 ns	-	6,643 ns	-
THREAD 1.1.4	181,488 ns	209,595,791 ns	-	5,792 ns	-
THREAD 1.1.5	181,643 ns	209,595,751 ns	-	5,856 ns	-
THREAD 1.1.6	176,170 ns	209,602,076 ns	-	6,072 ns	-
THREAD 1.1.7	181,905 ns	209,594,826 ns	-	5,518 ns	-
THREAD 1.1.8	181,303 ns	209,595,808 ns	-	11,643 ns	-
THREAD 1.1.9	-	229,008,005 ns	-	438 ns	-
THREAD 1.1.10	-	229,003,742 ns	-	4,701 ns	-
THREAD 1.1.11	-	229,007,979 ns	-	464 ns	-
THREAD 1.1.12	-	229,008,239 ns	-	204 ns	-
THREAD 1.1.13	-	229,008,076 ns	-	367 ns	-
THREAD 1.1.14	-	229,008,265 ns	-	178 ns	-
THREAD 1.1.15	-	229,006,294 ns	-	2,149 ns	-
THREAD 1.1.16	-	229,008,134 ns	-	309 ns	-
THREAD 1.1.17	-	229,008,267 ns	-	176 ns	-
THREAD 1.1.18	-	229,008,142 ns	-	301 ns	-
THREAD 1.1.19	-	229,006,536 ns	-	1,907 ns	-
THREAD 1.1.20	-	229,008,132 ns	-	311 ns	-
THREAD 1.1.21	-	229,008,265 ns	-	178 ns	-
THREAD 1.1.22	-	229,008,141 ns	-	302 ns	-
THREAD 1.1.23	-	229,008,265 ns	-	178 ns	-
THREAD 1.1.24	-	229,006,501 ns	-	1,942 ns	-
Total	224,335,456 ns	5,133,974,212 ns	5,907,831 ns	81,185 ns	2,920 ns
Average	28,041,932 ns	223,216,270.09 ns	5,907,831 ns	3,382.71 ns	2,920 ns
Maximum	223,083,474 ns	229,008,267 ns	5,907,831 ns	14,218 ns	2,920 ns
Minimum	173,788 ns	209,594,826 ns	5,907,831 ns	176 ns	2,920 ns
StDev	73,718,773.70 ns	8,770,207.20 ns	0 ns	4,113.66 ns	0 ns
Avg/Max	0.13	0.97	1	0.24	1

Imatge 8: Pi_omp_padding amb 8 threads

5. In the presence of false sharing (as it happens in pi omp sumvector.c), which is the additional average time for each individual access to memory 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.

Comparant les imatges 6 i 7, podem veure el temps adicional que emplea per accedir a memòria:
236.948.431ns - 229.008.443ns = 7.939.998ns.

Aquest augment de temps d'accés a memòria es deu a l'anomenat *false sharing*. Consisteix en la modificació de diferents posicions de memòria que estan a la mateixa línia de caché per part de diferents threads. Quan un thread modifica una posició, els altres threads invaliden la línia sencera.

```
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);
}
```

Imatge 9. Codi provocant de false sharing

La imatge 9 mostra la secció del codi que provoca el *false sharing* al codi de pi_omp_sumvector.c

```
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);
}
```

Imatge 10. Codi que soluciona false sharing

La imatge 10 mostra la seccio de codi de pi_omp_padding.c que soluciona *false sharing* utilitzant una matriu en comptes d'un vector.

6. Write down a table (or draw a plot) showing the execution times for the different versions of the Pi computation that we provide to you in this laboratory assignment (session 3) when executed with 100.000.000 iterations. and the speed–up achieved with respect to the execution of the serial version pi seq.c. For each version and number of threads, how many executions have you performed?

Versió	Temps per 1 thread (segons)	Temps per 8 threads (segons)	Speed-up respecte a pi_seq.c
Pi_seq	0.790152		1
Pi_omp	0.790941	0.157598	5.01372
Pi_omp_critical	1.791986	31.03918	0.254566
Pi_omp_atomic	1.469923	7.159772	0.110824
Pi_omp_sumvector	0.792085	0.619652	1.275154
Pi_omp_padding	0.791447	0.136580	5.785269

Per executar hem fet servir una commanda com la següent:

Per 1 thread: #OMP_NUM_THREADS = 1 ./pi_omp_critical 100000000 1
Per 8 threads: #OMP_NUM_THREADS = 8 ./pi_omp_atomic 100000000 8

Per calcular la mitjana d'execucions hem fet servir un excel.
Hem executat cada versió amb cada nombre de threads exactament 6 vegades i hem fet la mitjana. En total 66 execucions (6 amb 1 thread, 5 amb 8 threads, tot multiplicat per 6). No hem fet l'execució de pi_omp_parallel i pi_omp_tasks pel fet que trigaven un temps massa gran.