

# LLIURAMENT LAB 2

GRUP 13 – PAR 1305

ALBERT FIGUERA

23/10/17 Curs Tardor 2017-2018

## EXERCICI OPCIONAL

*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  $S_4$  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

Les caselles en vermell son les versions que no calculen correctament pi.

Apreciem que amb l'ús de la llibreria OMP no és suficient per paral·lelitzar el codi. En les versions que no calculen correctament pi ho apreciem clarament. Observem que en la versió 1 el temps d'execució és el doble. En la versió 2 intenta millorar-ho però no ho aconsegueix i augmenta el temps d'execució, degut a que tots els threads executen totes les iteracions. En canvi a la versió 3 s'aconsegueix beixar notoriament el temps assignant manualment les iteracions a cada hread. No ho calcula bé pero ja no tenim un data race.

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.

### **Conclusió**

Podem afirmar que un correcte ús de la llibreria OpenMP millora el temps d'execució. Observem uns speed-ups de gairebé 9, amb l'ús de 4 threads. Per altra banda també podem afirmar que el mal ús pot generar temps majors que el seqüencial i errors de càlcul.

# 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 "Hello world"** (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?

**OMP\_NUM\_THREADS = 4** abans de crida del programa

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

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 és correcte.** No executa el world i el hello del mateix ID seguits.  
Farà falta un **private(id)**.

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

Els missatges suren intercalas degut això les línies no surten en el mateix ordre.

### 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 línies

```
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 excepte 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.

Lo mateix que abans excepte pel 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 el més probable es que a cada thread llegeixi el valor i sumi 1. Després l'altre thread farà el mateix i així 8 cops. X = 8.

Amb el private i el first private cada thread té una còpia que no es compartida. Per tant imprimirà el valor de la x sense ninguna 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 }
```

Critical limita el nombre de threads que executen la regió crítica (++x) a 1 alhora. No passarà lo del 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?

El programa treura per pantalla un indefinite nombre de missatges amb el seu numero d'iteracions i.

Exemple:

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

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

Només hem de posar la i com a privada **#pragma omp parallel private(i)** abans del for

## 6.datarace.c

1. Is the program always executing correctly?

Al estar la variable x compartida el programa no s'està executant correctament. El resultat pot variar en el moment en que cada thread accedeix a la variable també depèn de l'ordre.

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

Crear una regió de mutual exclusió entre els threads respecte la variable x.

**#pragma omp critical(x)** en el for.

Garantir l'accés a x per a fer l'actualització de manera atòmica.

**#pragma omp atomic** abans del ++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?

No, només una part degut a que els diferents threads s'executen alhora i no sabem quin serà el que entra primer a les funcions, no sabem els missatges de "going to sleep in...". Però la diferència 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?*

Cada thread executa dues iteracions.

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

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?*

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...).

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.

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 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

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

Es sincronitzen i surten agrupades.

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



No hi ha diferència.

#### 4.collapse.c

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

Primer thread(0) 4 iteracions la resta 3. Surten 25 línies perquè el paràmetre (2) de collapse indica en quants bucles es volen paral·lelitzar.  $5^2 = 25$ .

*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.

Privatitzar les variables 'i' i 'j' perquè així cada thread tingui les seves propies còpies de les variables localment.

#### 5.ordered.c

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

Treient "nowait" al final del primer bucle.

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

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?*

Els missatges externs apareixen en un ordre indefinit i els missatges de l'interior apareixen ordenats.

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

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.*

Al no aturar-se a cada iteració, s'executa molt més ràpid i les dependències no suposen cap problema. L'execució es fa en ordre.

## C) Tasks

### *1.serial.c*

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

Sí era el que esperavem. Com podem observar el programa fa l'execució 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?*

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 d'afegir la directiva `#pragma omp single` al bucle `while`.

*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 successió 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 voldrà 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 necessari perquè només teníem un thread.

### *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.*

A l'inici només un dels threads contribueix a l'execució del bucle. Comença a contribuir a les tasques del taskloop quan es desperten de la "siesta".

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

➤ (#qsub -l execution submit-omp.sh pi\_omp\_parallel 1 24

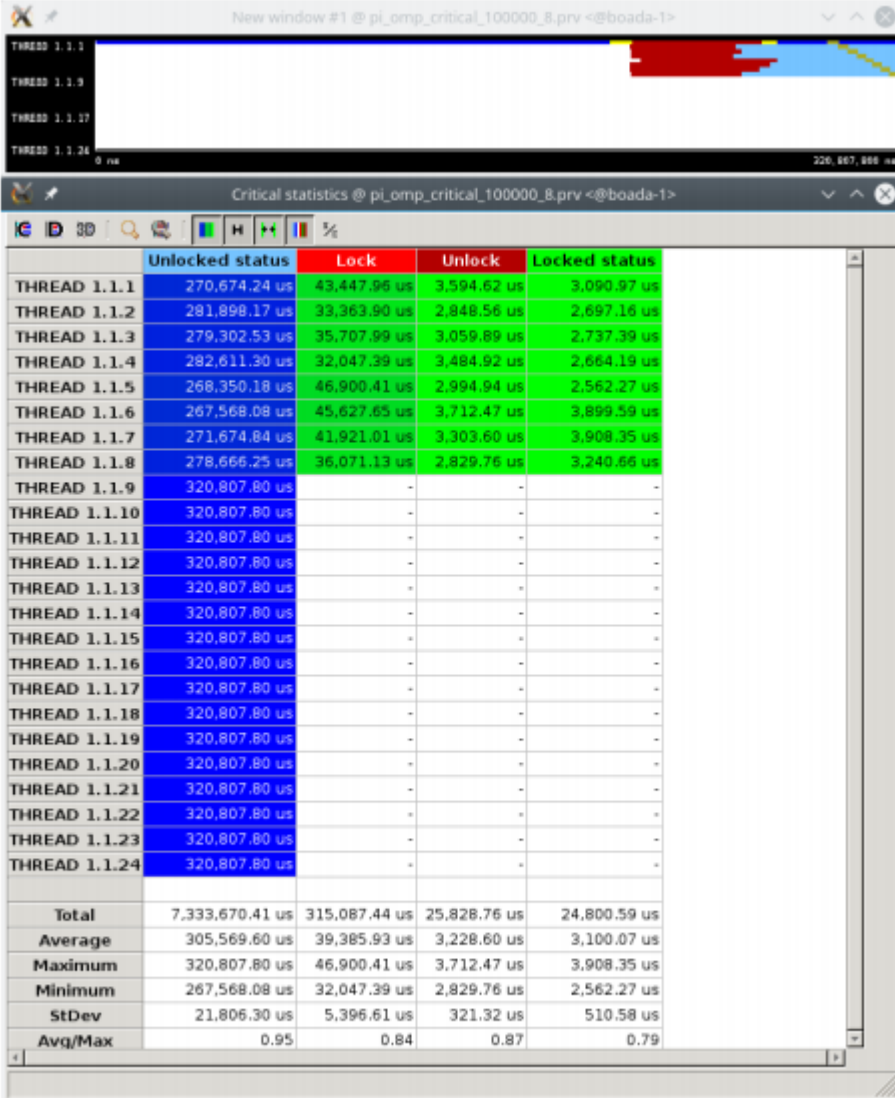
Observem com l'overhead no és constant i depèn directament del threads. Veiem com la dependència de l'overhead amb els threads es lineal a partir dels 12 threads. Amb això podem afirmar que amb el temps l'overhead es constant de cada thread. D'ordre de 0'2 microsegons.

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

Un únic thread i múltiples tasques. Podem afirmar que l'overhead depèn del nombre de tasques. Observem que l'overhead per task es constant de l'ordre del voltant de 0'1 microsegons (mínim de 0.1149  $\mu$ s i el màxim de 0.1282 $\mu$ s).

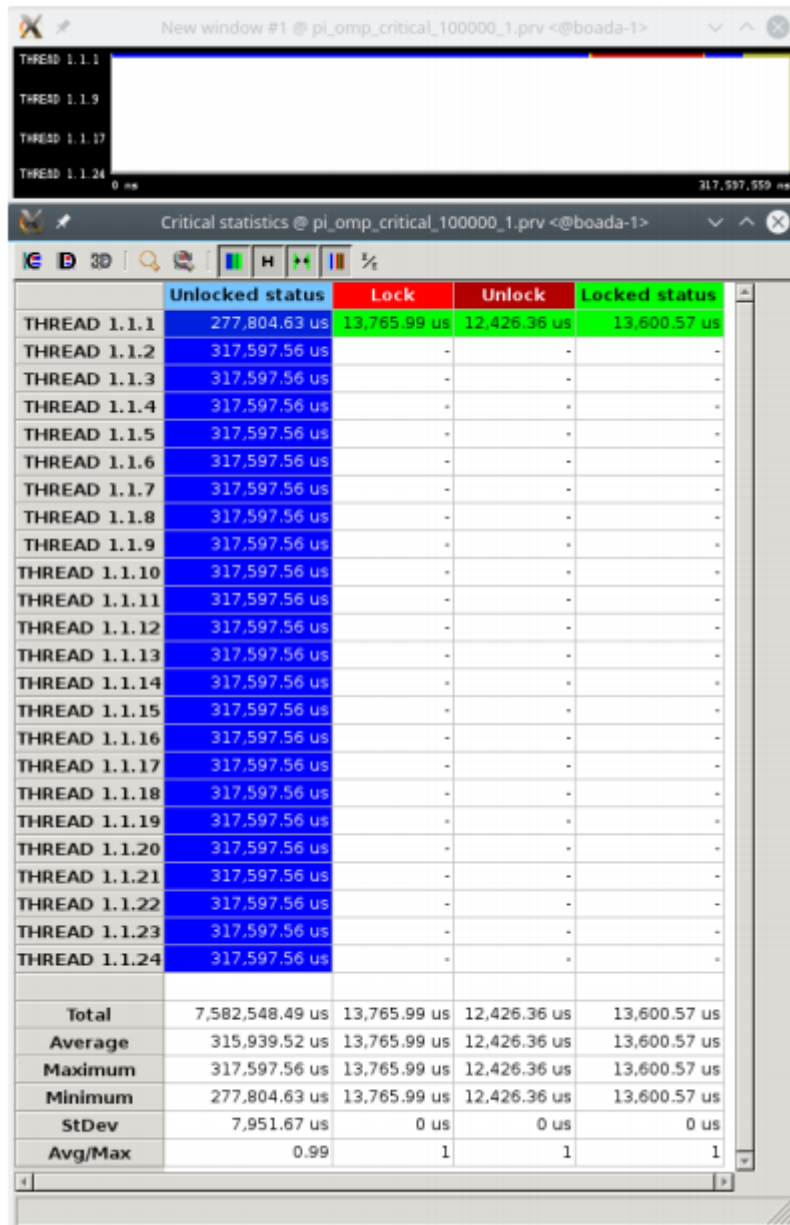
3. Which is the order of magnitude for the overhead associated with the execution of critical regions in OpenMP? How is this overhead decomposed? How and why does the overhead associated with critical increase with the number of processors? Identify at least three reasons that justify the observed performance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



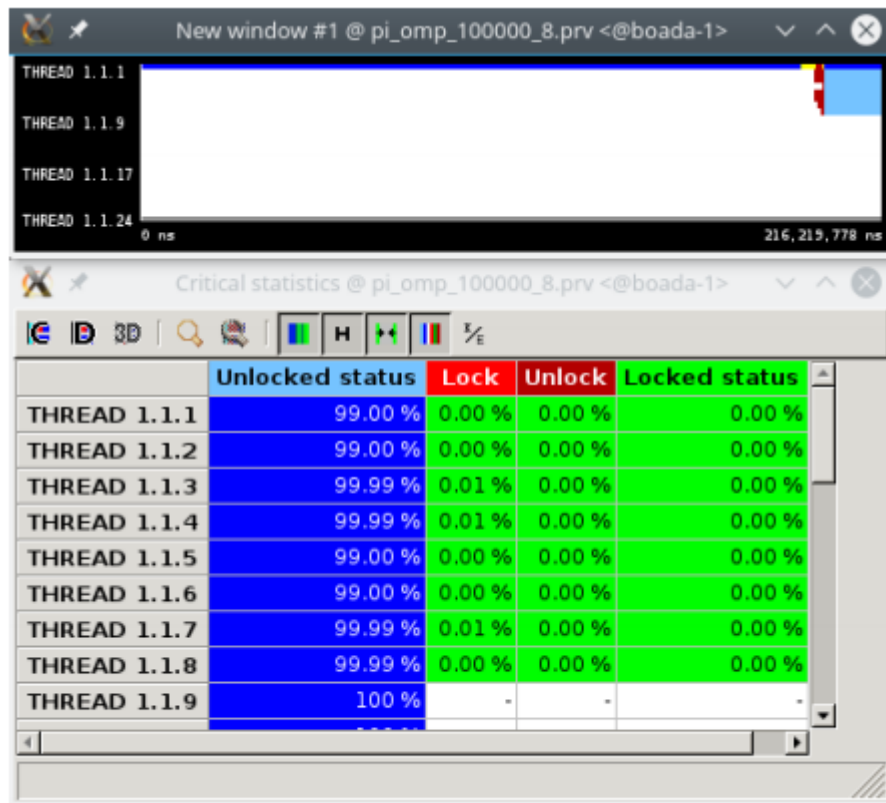
The image shows a Paraver execution trace for the program `pi_omp_critical_100000_8.prv`. The top window displays a timeline for threads 1.1.1, 1.1.9, 1.1.17, and 1.1.24. The bottom window shows critical statistics for the same program, detailing the execution times for various threads and overall statistics.

	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
Avq/Max	0.95	0.84	0.87	0.79

Imatge 1: `Pi_omp_critical`, 8 threads



Imatge 2: Pi\_omp\_critical 1 thread



Imatge 3: Pi\_omp amb 8 threads

A les imatges 1 i 2 observem que l'ordre de magnituds és de desenes de milers de microsegons, també podem afirmar que l'execució de l'overhead es descompon en Unlocked status, Lock, Unlock i Locked estatus 3. Per últim observem que ell nombre de threads augmenta el temps d'overhead.

Les raons que justifiquen la degradació del rendiment són la sincronització, la creació/destrucció de tasques i la 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.

	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	-
<b>Total</b>	266,088,771 ns	5,165,255,027 ns	17,283,775 ns	1,540,993 ns	82,252 ns	3,270 ns
<b>Average</b>	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
<b>Maximum</b>	226,596.013 ns	232,035,063 ns	3,882,352 ns	1,540,993 ns	12,657 ns	3,270 ns
<b>Minimum</b>	2,423,126 ns	206,979,421 ns	15,570 ns	1,540,993 ns	178 ns	3,270 ns
<b>StDev</b>	73,089,979.37 ns	11,299,895.15 ns	1,149,334.78 ns	0 ns	4,081.80 ns	0 ns
<b>Avg/Max</b>	0.15	0.97	0.56	1	0.27	1

Imatge 4: `Pi_omp_atomic` amb 8 threads

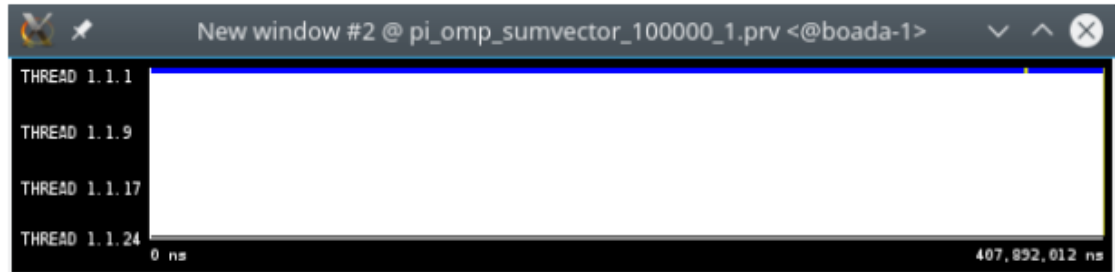


	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	387,960,941 ns	-	5,082 ns	22,998 ns	13,015 ns	3,325 ns
THREAD 1.1.2	-	388,004,301 ns	-	-	1,060 ns	-
THREAD 1.1.3	-	388,005,081 ns	-	-	280 ns	-
THREAD 1.1.4	-	388,005,074 ns	-	-	287 ns	-
THREAD 1.1.5	-	387,999,371 ns	-	-	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	-
<b>Total</b>	387,960,941 ns	8,924,104,107 ns	5,082 ns	22,998 ns	32,211 ns	3,325 ns
<b>Average</b>	387,960,941 ns	388,004,526.39 ns	5,082 ns	22,998 ns	1,342.12 ns	3,325 ns
<b>Maximum</b>	387,960,941 ns	388,005,179 ns	5,082 ns	22,998 ns	13,015 ns	3,325 ns
<b>Minimum</b>	387,960,941 ns	387,999,371 ns	5,082 ns	22,998 ns	182 ns	3,325 ns
<b>StDev</b>	0 ns	1,293.18 ns	0 ns	0 ns	2,743.51 ns	0 ns
<b>Avg/Max</b>	1	1.00	1	1	0.10	1

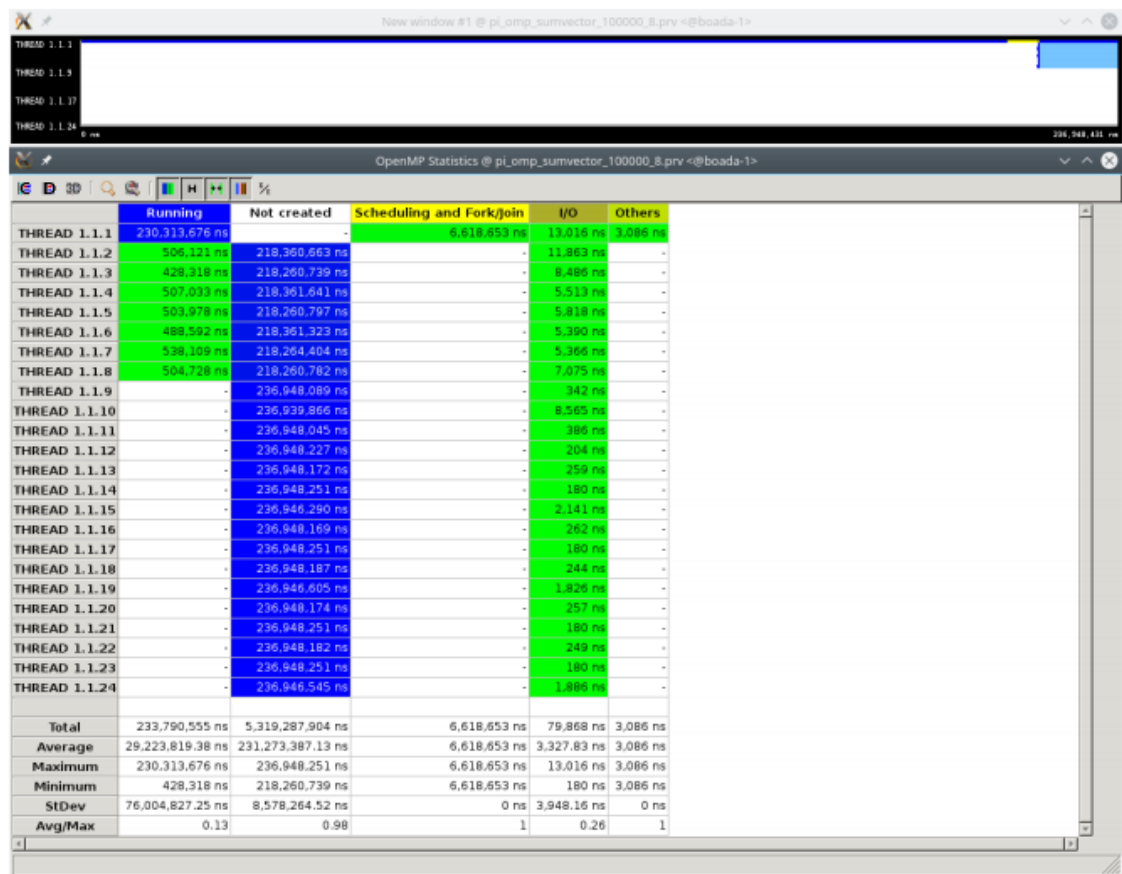
Imatge 5: Pi\_omp\_atomic amb 1 thread

Ordre de nano segons. Observem, a les imatges 4 i 5 juntament amb les anterior que l'overhead augmenta amb el nombre de threads degut a que només un thread pot executar una regió atòmica alhora. Podem afirmar que quants més threads més augmentarà el temps (degut a que hauran d'esperar més)

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



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

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

L'augment de temps d'accés a memòria es deu al false sharing.

```
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

```

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*

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

- #OMP\_NUM\_THREADS = 1 ./pi\_omp\_critical 100000000 1
- #OMP\_NUM\_THREADS = 8 ./pi\_omp\_atomic 100000000 8