

LABORATORI 0

Second deliverable

Critical statistics @ pi_omp_critical_i_100000_8.prv				
	Unlocked status	Lock	Unlock	Locked status
THREAD 1.1.1	39.61 %	49.82 %	5.58 %	4.00 %
THREAD 1.1.2	19.87 %	69.77 %	5.18 %	5.18 %
THREAD 1.1.3	40.60 %	49.23 %	5.31 %	4.86 %
THREAD 1.1.4	19.90 %	70.20 %	4.96 %	4.94 %
THREAD 1.1.5	39.00 %	49.83 %	5.27 %	4.90 %
THREAD 1.1.6	19.63 %	70.47 %	4.94 %	4.96 %
THREAD 1.1.7	22.24 %	67.72 %	5.02 %	5.02 %
THREAD 1.1.8	55.60 %	34.09 %	5.27 %	5.04 %
Total	257.45 %	461.13 %	41.53 %	39.89 %
Average	32.18 %	57.64 %	5.19 %	4.99 %
Maximum	55.60 %	70.47 %	5.58 %	5.18 %
Minimum	19.63 %	34.09 %	4.94 %	4.86 %
StDev	12.72 %	12.84 %	0.20 %	0.09 %
Avg/Max	0.58	0.82	0.93	0.96

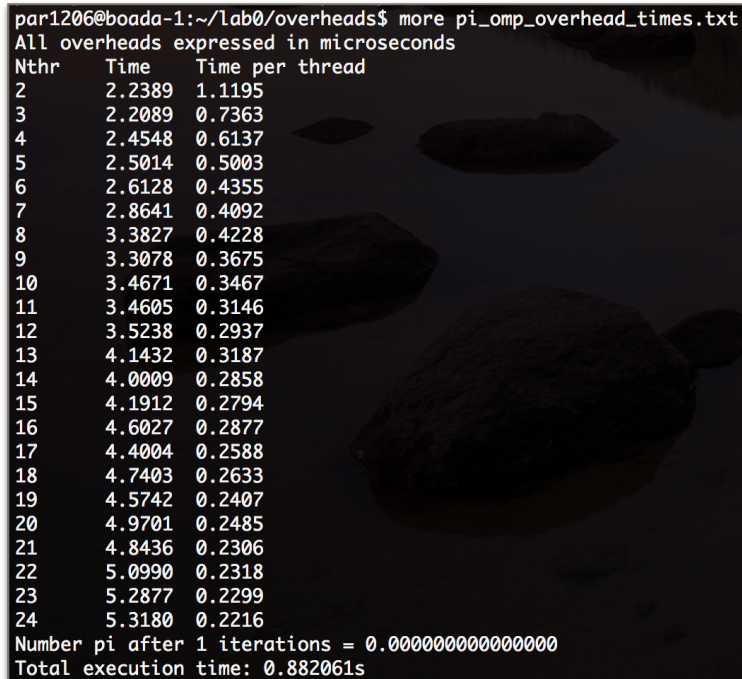
GRUP 12 - 06

Daniel Gil
Carles Viñeta

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.

Comandes que hem executat

```
# make
# qsub -l execution submit-omp-overhead.sh
# more pi_omp_overhead_times.txt
```



```
par1206@boada-1:~/lab0/overheads$ more pi_omp_overhead_times.txt
All overheads expressed in microseconds
Nthr   Time    Time per thread
2      2.2389   1.1195
3      2.2089   0.7363
4      2.4548   0.6137
5      2.5014   0.5003
6      2.6128   0.4355
7      2.8641   0.4092
8      3.3827   0.4228
9      3.3078   0.3675
10     3.4671   0.3467
11     3.4605   0.3146
12     3.5238   0.2937
13     4.1432   0.3187
14     4.0009   0.2858
15     4.1912   0.2794
16     4.6027   0.2877
17     4.4004   0.2588
18     4.7403   0.2633
19     4.5742   0.2407
20     4.9701   0.2485
21     4.8436   0.2306
22     5.0990   0.2318
23     5.2877   0.2299
24     5.3180   0.2216
Number pi after 1 iterations = 0.0000000000000000
Total execution time: 0.882061s
```

L'ordre de magnitud és de microsegons.

El temps no és constant, a mesura que augmentem els threads, el temps de fork i join augmenta però el temps de thread disminueix.

Càlcul aproximat amb les iteracions executades i les dades aconseguides:

En funció de l'overhead el #threads s'incrementa, quan el #threads és molt elevat, és molt estable el temps per thread.

Per calcular el cost fixe, agafem el Temps de 2 Threads (2.2389) i li restem el Temps d'un thread (0.2216) * 2 (ja que hem agafat el Temps de 2 threads). Per tant:

$$\text{Cost fixe} = 2.2389 - 0.22 * 2 = 1.8$$

Una vegada tenim el cost fixe, podem calcular el cost

$$\text{cost} = 1.8 (\text{cost fixe}) + 0.22 (\text{cost variable}) * (\text{\#threads})$$

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.

Primer de tot, podem veure a la Figura 2 que l'ordre de magnitud de l'overhead associat a les regions crítiques del nostre programa és entre 2 i 3 segons per cada thread afegit. Cal tenir en compte que aquest overhead està descompost en tres regions: fork, locks i joins. Com es pot veure a la Figura 7, la regió que més aporta a l'overhead és la de locks. Això és degut a que els diferents threads s'estan disputant constantment l'accés a la variable sum, com es pot veure a la Figura 5.

A cada thread afegit augmenta l'overhead degut a que aquest també té en compte la variable crítica sum, el que el fa incrementar considerablement la regió del locked. A més, també s'augmenta les regions de fork i join, tot i que és amb menys mesura.

No obstant, aquest problema el podríem resoldre amb una variable local sum per cada thread, tal i com ho té en compte el programa pi_omp. Podem veure doncs, observant en la taula (Figura 3) de pi_omp que de un a vuit threads, el temps d'execució millora degut a que la regió de lock només la trobarem al moment de sumar totes les variables locals i no per cada iteració del bucle interior, ja que fa disminuir amb gran mesura la regió locked.

#Threads	User	System	Elapsed	%CPU
1	0.78	0.00	0:00.81	97

Figura 1: Taula dels temps del programa pi_seq executat amb 100,000,000 d'iteracions.

#Threads	User	System	Elapsed	%CPU
1	1.82	0.00	0:01.85	98
2	19.42	0.00	0:10.00	194
3	36.53	0.00	0:13.55	269
4	42.03	0.19	0:15.63	347
5	69.88	0.00	0:17.05	409
6	100.04	3.90	0:19.86	523
7	154.55	5.35	0:24.70	647
8	192.10	5.11	0:27.91	706

Figura 2: Taula de temps del programa pi_omp_critical executat amb 100,000,000 d'iteracions.

#Threads	User	System	Elapsed	%CPU
1	0.79	0.00	0:00.79	99
2	0.80	0.00	0:00.40	197
3	0.80	0.00	0:00.27	295
4	0.81	0.00	0:00.21	388
5	0.85	0.00	0:00.17	486
6	0.85	0.00	0:00.14	582
7	0.82	0.03	0:00.12	679
8	0.87	0.00	0:00.11	771

Figura 3: Taula de temps del programa pi_omp executat amb 100,000,000 d'iteracions.

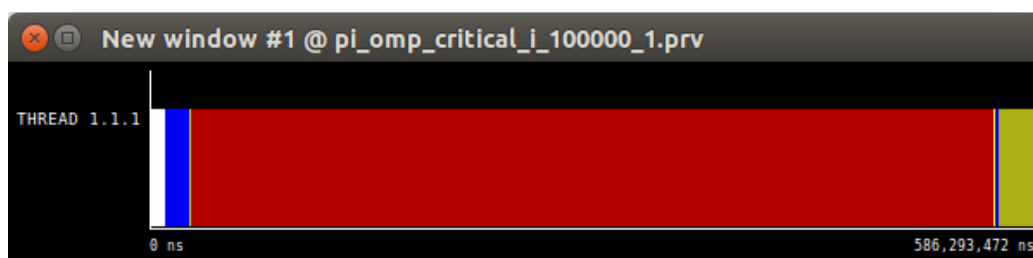


Figura 4: Timeline de pi_omp_critical per 1 thread.

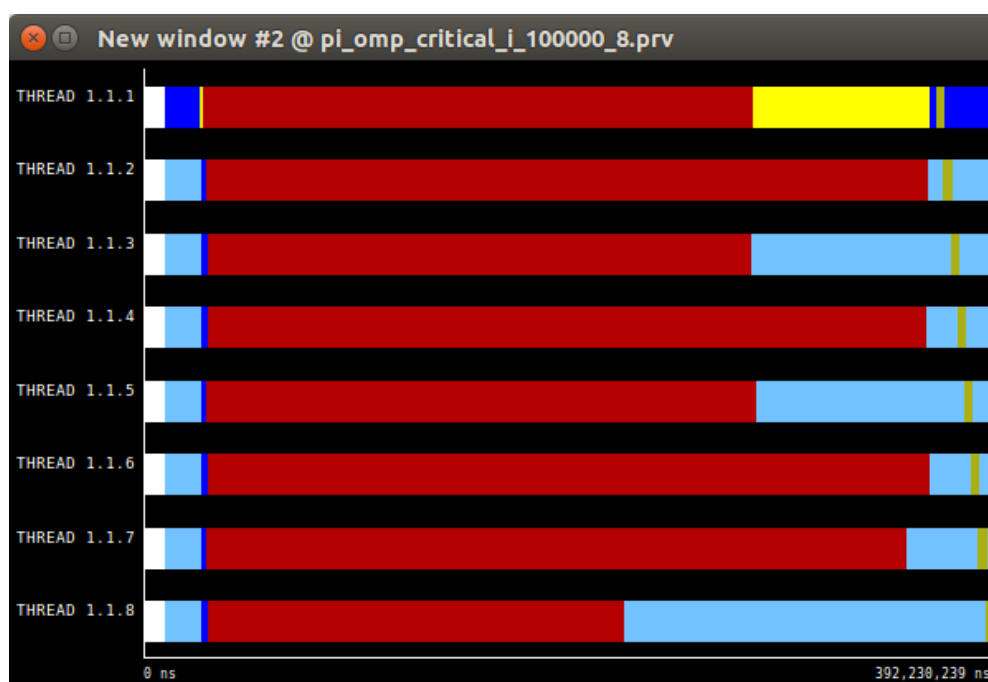


Figura 5: Timeline de pi_omp_critical per 8 threads.

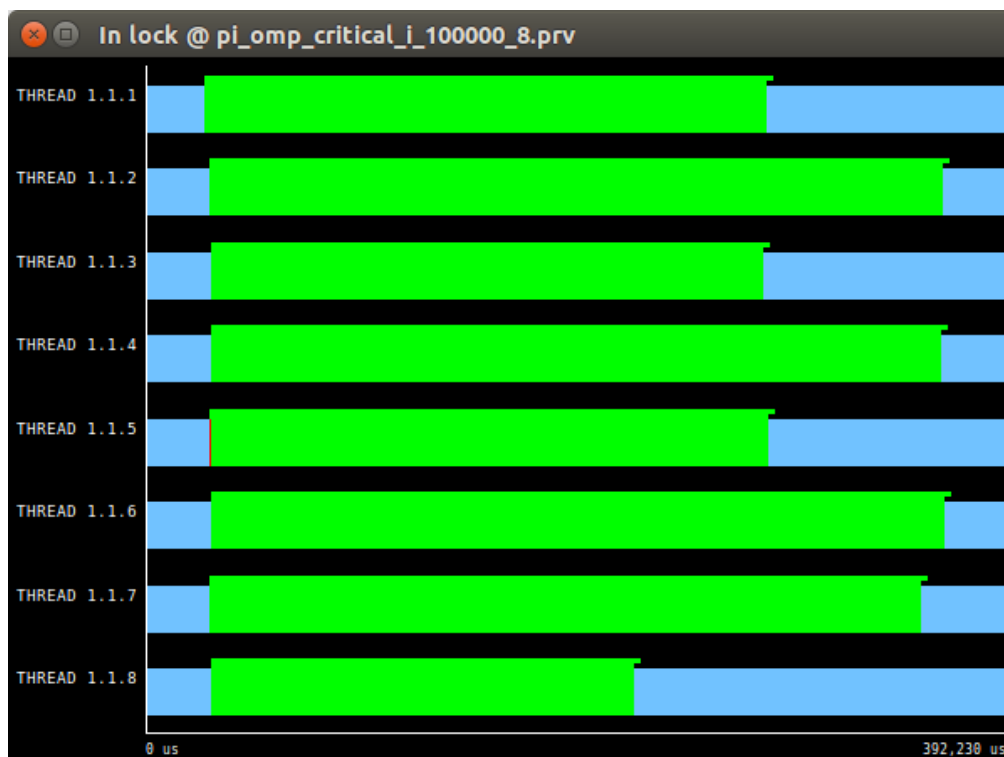


Figura 6: Timeline de pi_omp_critical per 8 threads, utilitzant el perfil omp_in_lock.

Critical statistics @ pi_omp_critical_i_100000_8.prv				
	Unlocked status	Lock	Unlock	Locked status
THREAD 1.1.1	39.61 %	49.82 %	5.58 %	4.00 %
THREAD 1.1.2	19.87 %	69.77 %	5.18 %	5.18 %
THREAD 1.1.3	40.60 %	49.23 %	5.31 %	4.86 %
THREAD 1.1.4	19.90 %	70.20 %	4.96 %	4.94 %
THREAD 1.1.5	39.00 %	49.83 %	5.27 %	4.90 %
THREAD 1.1.6	19.63 %	70.47 %	4.94 %	4.96 %
THREAD 1.1.7	22.24 %	67.72 %	5.02 %	5.02 %
THREAD 1.1.8	55.60 %	34.09 %	5.27 %	5.04 %
Total	257.45 %	461.13 %	41.53 %	39.89 %
Average	32.18 %	57.64 %	5.19 %	4.99 %
Maximum	55.60 %	70.47 %	5.58 %	5.18 %
Minimum	19.63 %	34.09 %	4.94 %	4.86 %
StDev	12.72 %	12.84 %	0.20 %	0.09 %
Avg/Max	0.58	0.82	0.93	0.96

Figura 7: Taula obtinguda gràcies a l'execució del programa pi_omp_critical utilitzant el perfil omp_critical_profile.

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.

Tenint en compte la figura 8, es pot determinar l'ordre de magnitud de l'overhead. Aquest es pot considerar despreciable, ja que l'atomicitat de l'operació garanteix, a diferència de la versió pi_omp_critical, que no hi haurà locks intermitjos entre els diferents threads.

Tot i això, aquesta atomicitat, no ens serveix, ja que el temps d'execució paral·lela de cada thread no està disminuint, per tant, ens serà difícil igualar el rendiment de la versió amb variables locals (pi_omp) per cada thread. És a dir, si els locks no apareixen no és garantia que els threads no continuïn accedint a una mateixa variable compartida per tots.

#Threads	User	System	Elapsed	%CPU
1	1.44	0.00	0:01.45	99
2	16.02	0.00	0:08.20	195
3	26.72	0.00	0:09.24	289
4	35.61	0.00	0:09.11	390
5	41.83	0.00	0:09.34	447
6	50.48	0.00	0:08.68	581
7	52.47	0.00	0:08.09	648
8	78.77	0.00	0:09.96	790

Figura 8: Taula de temps del programa pi_omp_atomic executat amb 100,000,000 d'iteracions.

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.

El primer pas és comparar el codi dels dos programes (A: pi_omp_sumvector i B: pi_omp_padding). Així doncs, veiem en la figura 9 que l'única diferència és la posició on el thread guarda els càlculs que es realitza.

<pre>for (int i=0; i<NUMTHRDS; i++) sumvector[i] =0.0; #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); } } for (int i=0; i<NUMTHRDS; i++) sum += sumvector[i];</pre>	A	<pre>for (int i=0; i<NUMTHRDS; i++) sumvector[i][0] =0.0; #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); } } for (int i=0; i<NUMTHRDS; i++) sum += sumvector[i][0];</pre>	B
--	---	--	---

Figura 9: Secció de codi de pi_omp_sumvector i pi_omp_padding, respectivament.

En el segon cas, el programa pi_omp_padding utilitza una matriu per les variables locals de cada thread (la primera posició de cada línia de la matriu correspon a una variable local). Per contra, en el pi_omp_sumvector s'utilitza un vector.

Aquest canvi en pi_omp_padding es realitza per evitar el 'false sharing'. Cal dir que el false sharing té lloc quan diferents threads modifiquen adreces de memòria que es troben en una mateixa línia de cache. Així doncs, en el pi_omp_padding, les direccions a les que s'accedeix cada thread seran mostrades en diferents línies de cache, eliminant els requests a causa del 'false sharing'. En conseqüència, el número d'accessos augmenta i el temps d'execució també.

Per acabar, cal dir que el temps addicional d'accés a memòria de pi_omp_padding respecte a pi_omp_sumvector serà la resta dels temps d'execució. Finalment, es pot observar gràcies a les dades obtingudes que el 'false sharing' provoca un temps extra de l'ordre de 0.45 segons aproximadament.

Resumint, al fer el canvi de vector a matriu (amplada -> amplada de la caché i alçada -> nombre de processadors), considerarem que la dada que volem és la que es troba a la posició **M[0][num_thread]**.

#Threads	User	System	Elapsed	%CPU
1	0.82	0.00	0:00.85	97
2	1.68	0.00	0:00.87	193
3	1.96	0.00	0:00.72	271
4	2.19	0.00	0:00.59	369
5	2.55	0.00	0:00.57	444
6	3.45	0.00	0:00.59	579
7	3.75	0.00	0:00.59	629
8	4.44	0.00	0:00.58	764

Figura 10: Taula de temps del programa pi_omp_sumvector executat amb 100,000,000 d'iteracions.

#Threads	User	System	Elapsed	%CPU
1	0.79	0.00	0:00.80	99
2	0.80	0.00	0:00.40	197
3	0.81	0.00	0:00.27	296
4	0.82	0.00	0:00.21	388
5	0.84	0.00	0:00.17	479
6	0.86	0.00	0:00.15	578
7	0.87	0.00	0:00.12	675
8	0.87	0.00	0:00.11	771

Figura 11: Taula de temps del programa pi_omp_padding executat amb 100,000,000 d'iteracions.

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?

version	1 processor (seg)	8 processors (seg)	speed-up
pi_seq.c	0.81	-	1.00
pi_omp.c (sumlocal)	0.79	0.11	7.18
pi_omp_critical.c	1.85	27.91	0.06
pi_omp_lock.c	1.55	14.21	0.10
pi_omp_atomic.c	1.45	9.87	0.14
pi_omp_subvector.c	0.86	0.58	1.47
pi_omp_padding.c	0.80	0.11	7.27

Com a resultats de la taula, cal dir que hem executat cinc vegades el programa per cada versió i el número de threads. A més, hem calculat la mitjana aritmètica dels seus temps, tot i que abans, descartant els dos extrems de major a menor.

Després d'obtenir els speed-up's de les diferents versions, podem observar que totes les versions, menys pi_omp_critical, pi_omp_lock i pi_omp_atomic, redueixen els temps d'execució (del processador 1 respecte al processador 8). Per contra, aquestes tres versions esmentades, tenen speed-up's negatius, ja que els hi augmenta el temps d'execució.

Per altra banda, quan obtenim un speed-up positiu, significa que s'ha produït una disminució del temps d'execució, per exemple, pi_omp_padding redueix el temps d'execució, ja que s'ha reduït 'false sharing'.