PAR

Second Deliverable

Marlen Avila, Ricard Meyerhofer
Group 2104
Curs 2015-2016 Q1

Second Deliverable PAR

Part I: OpenMP questionnare

A- Basics

1.hello.c

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

R/ If #pragma omp parallel num_threads(x) isn't specified, the number of threads executed will be the default number (which is 24).

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

R/ OMP_NUM_THREADS=4 ./1.hello

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?.

R/ No, because the id variable is shared by all the threads. To make it correct we add this: **#pragma omp critical.** Provides a region of mutual exclusion where only one thread can be working at any given time.

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

R/ No, because we can't guarantee which thread is the first to execute.

R/ No with the critical clause, because that code region will be done only by just one thread, so it's impossible that messages appear intermixed.

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?

R/ 16.

- 8 lines from the first parallel due to the **export 8**.
- 2 lines from the second parallel due to the **omp_set_num_threads(2).**
- 3 lines from the third parallel due to the **#pragma omp parallel num_threads(3).**
- 2 lines because **omp_set_num_threads(2)** prevails.
- 1 line from the last one (will be explained above).

-

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

R/ It depends on the random value assigned. With the **if(0)** the num_threads(rand()%4+1) is ignored and is considered as 1.

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)?

R/ - After first parallel (shared) x is: 5,8,6,etc.. (race condition).

- After second parallel (private) x is: 0 (the value is 0 because we're printing the x declared outside this pragma. If the attribute is private, the x variable we see inside the pragma is a new one [independent for each thread and with an undefined value, not the one declared before]).

- After third parallel (first private) x is: 0 (same as private attribute).

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

R/ It needs to be **added:**

```
#pragma omp parallel shared(x)
{
        #pragma omp atomic
        x++;
}
```

5.parallel.c

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

R/ The i variable is shared by all the threads, so we know that we'll have a race condition issue. Because of that, in each execution we can have a different number of printed lines. If the race condition wouldn't exist, each thread would print 5 messages (N =20 and NUM_THREADS = 4), but in our case this won't happen. A thread can change the i value when another different thread is iterating.

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

R/ We've modified the code and this is our version:

```
int i,id;
omp_set_num_threads(NUM_THREADS);
#pragma omp parallel private(i,id)
{
        id=omp_get_thread_num();
        for (i=id; i < N; i=i+NUM_THREADS) {
                printf("Thread ID %d Iter %d\n",id,i);
        }
}
```

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

1. Is the program always executing correctly?

R/ No, there are errors in some executions

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

R/

Alternative 1:

```
#pragma omp atomic
        x++;
```

Alternative 2:

```
 #pragma omp parallel private(i) reduction (+: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?

R/ No, we can't predict the following instructions :

"printf("(%d) going to sleep for %d seconds ...\n",myid,2+myid*3);"

"printf("(%d) We are all awake!\n",myid);"

because the sequence of thread execution is unknown. But in the middle part ("**printf("(%d) wakes up and enters barrier ...\n",myid")** we'll always know the sequence that will be executed because it depends on the sleep time of each thread in the first part.

R/ No because the barrier does wait all the threads and once all of them are at the barrier point, it happens the same that in the first execution (we cannot know which thread will be executed).

B) Worksharing

1.for.c

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

R/ 2 iterations/threads because we have N = 16 and 8 threads and with the for clause the number of iterations is divided among the number of threads. There are no race conditions issues because the induction variable (i) is private by default.

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

R/ In this case, the number of iterations executed by each thread is not the same for all of them.Because we have to divide 19 iterations among 8 threads so we'll have some threads with 3 iterations (0,1,2) and the rest with 2 iterations

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

R/ We've added the single clause:

#pragma omp single

printf("Going to distribute iterations in first loop ...\n");

With this clause only one thread of the team executes the structured block.

2.schedule.c

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

R/

Loop 1:

All the threads will execute the same number of iterations (4 consecutive iterations each thread)

Loop 2:

All the threads will execute 4 iterations, but this time not in a consecutive way. Instead of that, the first thread will execute the first 2 iterations, the next one will execute the following 2, etc..

Loop 3:

The number of iterations that each thread will execute is unknown because the iterations are assigned dynamically.

Loop 4:

R/ In this case, with the guided clause, we have 12 iterations divided among 3 threads, so the first thread will execute the first 4 iterations. Next, we do the same division but this time the rest of iterations (8) among the same number of threads,

and so on until we're over. The chunk is 2, and it limits the minimum number of iterations per thread that we can do when we're decreasing.

3.nowait.c

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

R/ With the nowait clause in the first loop, the messages are printed "intermixed": in our case, when the first thread has executed the two first iterations of that loop, it doesn't wait and jumps to the second loop to execute the other two iterations. That will happen with all the other threads.

If we remove the nowait clause, then the threads will execute their two corresponding iterations of the first loop and they will wait to execute the other ones of the second loop until the first loop is finished (due to the implicit barrier).

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

R/ No, any difference will be noticed since there are no tasks to be executed by the threads.

4.collapse.c

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

R/

Thread	Iterations(i,j)
0	(0,0), (0,1), (0,2), (0,3)
1	(0,4), (1,0), (1,1)
2	(1,2), (1,3), (1,4)
3	(2,0), (2,1), (2,2)
4	(2,3), (2,4), (3,0)
5	(3,1), (3,2), (3,3)
6	(3,4), (4,0), (4,1)
7	(4,2), (4,3), (4,4)

With the collapse clause we're reducing the granularity of work that each thread does. In addition we avoid a possible race condition because the induction variables (i and j) of both loops are private.

2. Is the execution correct if the collapse clause is removed? Which clause (different than collapse) should be added to make it correct?.
R/ The execution isn't correct since the induction variables (i and j) aren't private anymore and that creates a race condition.
R/ To fix that we've added the **private(j)** clause in the **#pragma omp parallel for**.
Although the result is correct and the total number of iterations is the same, the number of threads used is different (also the number of iterations executed by each thread).
In the collapse clause we're doing a better task repartition than with the private clause.

C) Tasks

1.serial.c

1. Is the code printing what you expect? Is it executing in parallel?
R/ Is printing the expected results. Is not executing in parallel because all the iterations are computed by the same thread. In addition there isn't any parallel clause in the code.

2.parallel.c

1. Is the code printing what you expect? What is wrong with it?
R/ No is not printing the expected result.
R/ The fibonacci result isn't correct because the task distribution (#pragma omp task) is done by the four threads due to the #pragma omp parallel above. The induction variable (i) is now 4 instead of 1, and the error spreads.

2. Which directive should be added to make its execution correct?
R/ We have added the **#pragma omp single** after the **#pragma omp parallel firstprivate(p) num_threads(4)**.
Now with this directive only one thread will create the tasks of the traversal and not all of them.

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?
R/ Nothing, the execution will be the same as it was with the clause because we still have the first pragma.
R/ Segmentation Fault the reason is explained in the question 4.

R/ They are redundant because both firstprivates do the same.

4. Why the program breaks when variable p is not firstprivate to the task?
R/ Because we try to acces p when its value is NULL (due to the fact that p is in a race condition zone).

5. Why the firstprivate clause was not needed in 1.serial.c?
R/ The firstprivate is usefull in a parallel implementation. In a serial implementation it won't affect the execution because the point of firstprivate is that the variable inside the construct is a new variable of the same type but initialized to the original value (if we have just one thread there won't be a race condition so there's no point in adding the firstprivate).

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.
R/
par2104@boada-1:~/lab0/overheads$ cat pi_omp_overhead_times.txt
(All overheads expressed in microseconds)

Nthr	Time	Time per thread
2	2.1266	1.0633
3	1.9478	0.6493
4	2.4131	0.6033
5	2.5402	0.5080
6	2.6676	0.4446
7	2.9351	0.4193
8	3.4857	0.4357
9	3.3844	0.3760
10	3.5999	0.3600
11	3.6324	0.3302

12	3.6586	0.3049
13	4.1836	0.3218
14	3.9500	0.2821
15	4.1116	0.2741
16	4.5103	0.2819
17	4.3328	0.2549
18	5.1901	0.2883
19	4.5420	0.2391
20	5.1551	0.2578
21	4.7734	0.2273
22	5.1483	0.2340
23	5.3914	0.2344
24	5.3534	0.2231

Total execution time: 0.890435s

R/ With the results shown in the table above, we can see that from the Nthr 21, the execution time per thread is almost the same. In conclusion, that will be the overhead time (aprox 0.23 microseconds) because there are a lot more threads than the ones we need to calculate the pi number, so the job they're doing is mostly syncronization tasks, thread creations, fork/join, etc.instead of calculating.

R/ The overhead time is not a constant value because it depends on a lot other factors and not just the program execution we've done.

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.

R/

The order of magnitude of both is:

par2104@boada-1:~/pene/lab0/overheads$ ./pi_omp_critical 100000000
Number pi after 100000000 iterations = 3.141592653589677
Total execution time: **40.242931s**
Time per critical = 40.24/numsteps = **4.024*10^-7;**
par2104@boada-1:~/pene/lab0/overheads$ ./pi_omp 100000000
Number pi after 100000000 iterations = 3.141592653589783
Time per critical = Total execution time: **0.091565s**

The overhead is decomposed in locks and unlocks which will increase with the number of threads.

Execution time with 1 thread:

par2104@boada-1:~/lab0/overheads$ ./run-omp.sh pi_omp 1 1
Number pi after 1 iterations = 3.200000000000000
Total execution time: 0.000120s

par2104@boada-1:~/lab0/overheads$ ./run-omp.sh pi_omp_critical 1 1
Number pi after 1 iterations = 3.200000000000000
Total execution time: 0.000123s

par2104@boada-1:~lab0/overheads$ ./run-omp.sh pi_seq 100000000 1
Number pi after 100000000 iterations = 3.141592653590426
Total execution time: 0.791917s

Execution time with 8 threads:

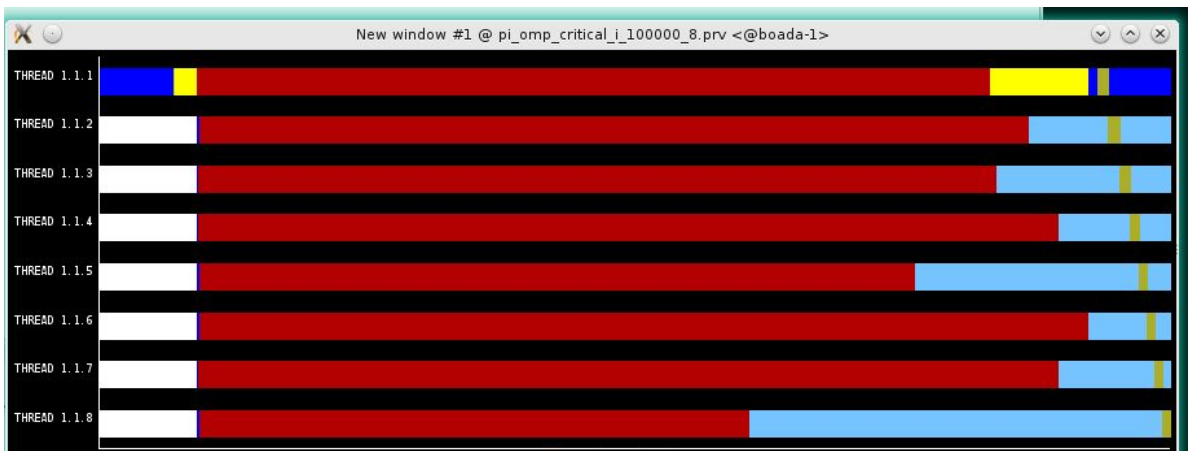
par2104@boada-1:~/lab0/overheads$ ./run-omp.sh pi_omp 1 8
Number pi after 1 iterations = 3.200000000000000
Total execution time: 0.005278s

par2104@boada-1:~/lab0/overheads$ ./run-omp.sh pi_omp_critical 1 8
Number pi after 1 iterations = 3.200000000000000
Total execution time: 0.002865s

par2104@boada-1:~lab0/overheads$ ./run-omp.sh pi_seq 100000000 8
Number pi after 100000000 iterations = 3.141592653590426
Total execution time: 0.845595s

Paraver omp_critical with 1 thread:

Paraver omp-critical with 8 threads:



	Unlocked status	Lock	Unlock	Locked status
THREAD 1.1.1	30.19 %	60.46 %	4.63 %	4.72 %
THREAD 1.1.2	27.93 %	60.98 %	5.58 %	5.51 %
THREAD 1.1.3	30.15 %	60.07 %	5.05 %	4.73 %
THREAD 1.1.4	24.40 %	65.53 %	5.12 %	4.94 %
THREAD 1.1.5	37.75 %	52.62 %	4.81 %	4.82 %
THREAD 1.1.6	21.57 %	68.99 %	4.69 %	4.74 %
THREAD 1.1.7	24.73 %	64.99 %	5.15 %	5.14 %
THREAD 1.1.8	53.19 %	37.60 %	4.60 %	4.62 %
Total	249.91 %	471.24 %	39.63 %	39.23 %
Average	31.24 %	58.90 %	4.95 %	4.90 %
Maximum	53.19 %	68.99 %	5.58 %	5.51 %
Minimum	21.57 %	37.60 %	4.60 %	4.62 %
StDev	9.49 %	9.25 %	0.31 %	0.27 %
Avg/Max	0.59	0.85	0.89	0.89

The first execution with one thread is faster than the execution with 8 threads. The calculus part is small and the lock status represents about a 60% of the execution time.

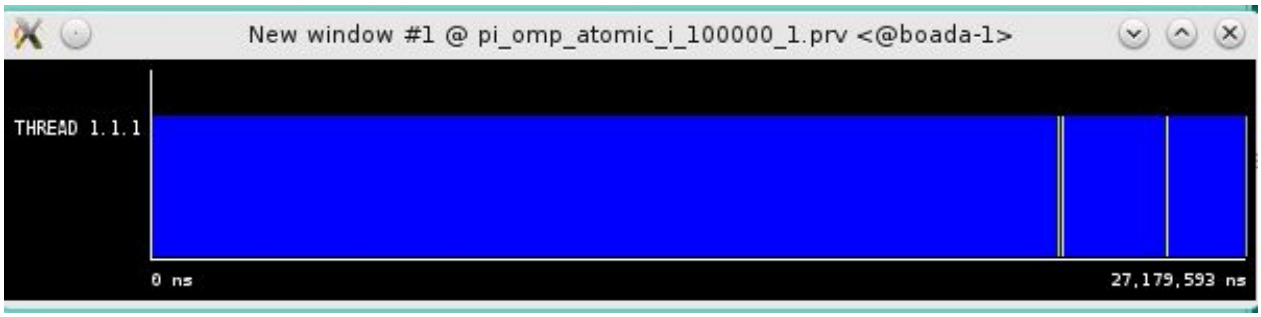
In conclusion if we use more threads the execution time will be higher due to the overhead times.

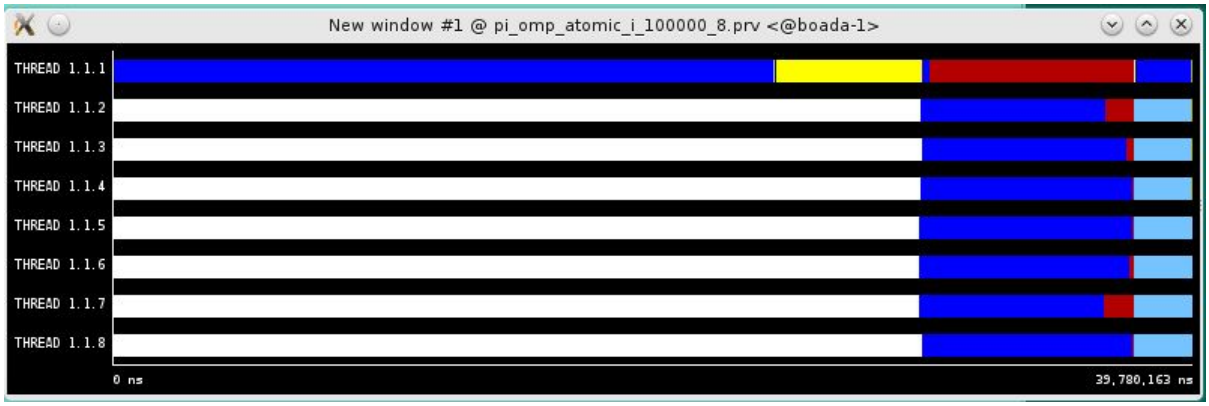
3. Which is the order of magnitude for the overhead associated with the execution of atomic memory accesses in OpenMP? How and why does the overhead associated with atomic increase with the number of processors? Reason the answers based on the execution times reported by the pi omp.c and pi omp atomic.c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

Execution with 1 thread:
par2104@boada-1:~//lab0/overheads$ ./pi_omp_atomic 100000
Number pi after 100000 iterations = 3.141592653598162
Total execution time: 0.002635s

Execution with 8 threads:
par2104@boada-1:~/lab0/overheads$ ./pi_omp_atomic 100000
Number pi after 100000 iterations = 3.141592653598091
Total execution time: 0.030559s

Overhead per thread: (E8Threads-E1Threads)/8 = 3.5*10^-3



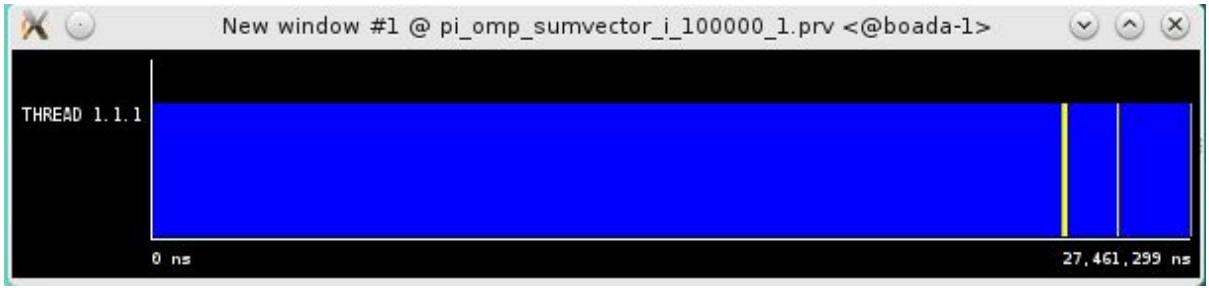


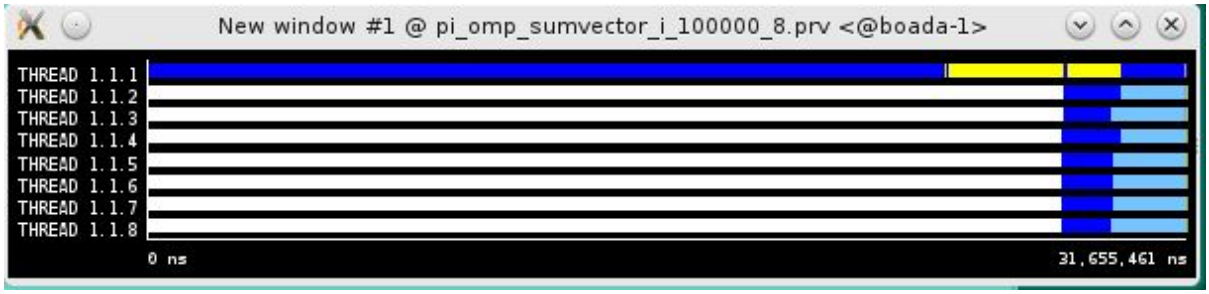
It happens the same that in the 2nd question because essentially, an atomic is approximately the same as a parallel since the code with the atomic has to finish its execution to let another thread starts its own.
Although it's faster than with the critical, the lock part is still more relevant to the execution.

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.
R/

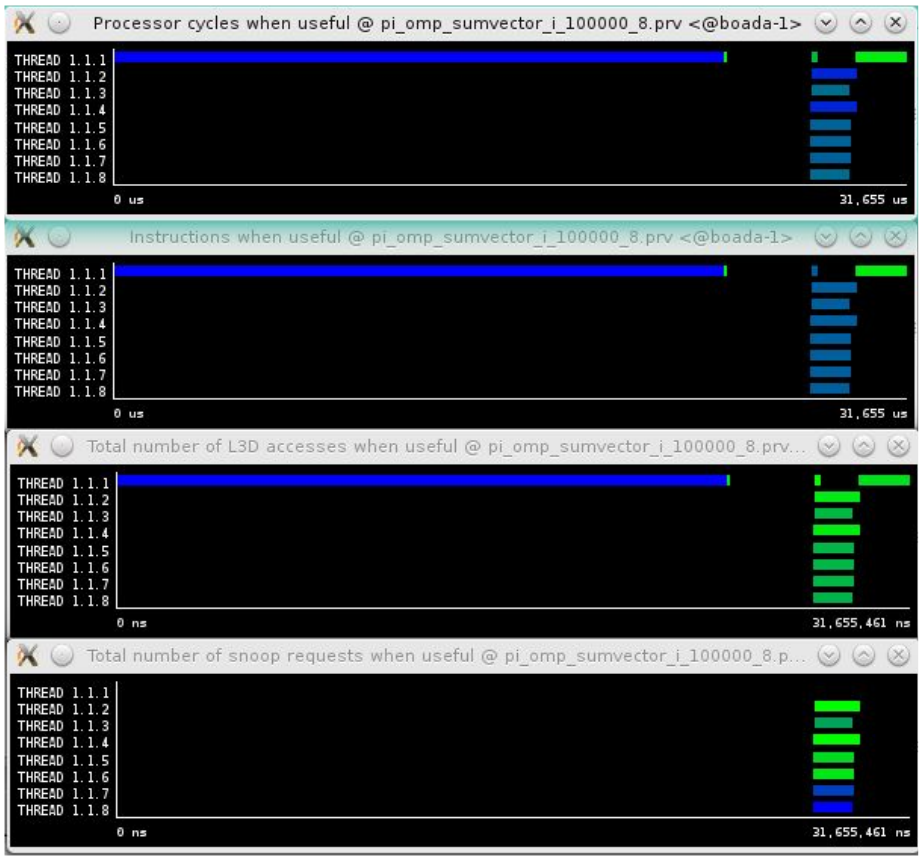
par2104@boada-1:~//lab0/overheads$ **OMP_NUM_THREADS=1**
par2104@boada-1:~/lab0/overheads$ **./pi_omp_sumvector 100000**
Number pi after 100000 iterations = 3.141592653598162
Total execution time: **0.000826s**
par2104@boada-1:~lab0/overheads$ **OMP_NUM_THREADS=8**
par2104@boada-1:~/lab0/overheads$ **./pi_omp_sumvector 100000**

Number pi after 100000 iterations = 3.141592653598125
Total execution time: **0.008400s**
(E8Threads-E1Threads)/Nthreads = 0.007574/8;





The time increase of the memory access is due to that the threads share the same cache line although they don't share the same data, because of that every time a thread needs to read this data has to load all the entire line again.
The more threads we have, the more false sharing, which also implies more access memory time and execution time.



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	8 processors	Speed-Up(/seq)
pi_seq.c	0.792412	-	
pi_omp.c (sumlocal)	0.789323	0.119267	6.64
pi_omp_critical.c	1.828295s	18.430813s	0.043
pi_omp_lock.c	1.792628s	51.760780s	0.015

pi_omp_atomic.c	1.445285s	9.616563s	0.082
pi_omp_sumvector.c	0.791293s	0.672219s	1.17
pi_omp_padding.c	0.788475s	0.185948s	4.26

We’ve done 2 executions.