LAB 2 DELIVERABLE

PAR 2108

Raül Montoya Pérez
Miguel Ángel Álvarez Vázquez

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 times, because our computer has 24 threads, as we saw in the last deliverable, and each thread execute the print.

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

We have to use the command line “OMP_NUM_THREADS=4” to use only 4 threads in the parallel region and print only 4 times the message (one per thread). If we want to use always 4 threads we have to write “export” before that command line.

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?.

The execution of the program is not correct. We have to add the clause private(id) in the pragma declaration in order to fix it, because with that clause each thread will have a local copy of the variable id and each thread will print the correct number of id.

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

No, that’s because the printing of the message is managed by the operating system so the messages can be printed in different order. Consequently, the messages could appear intermixed for the same reason.

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?

There are 16 lines printed. The first parallel prints the firsts 8 lines, 1 per thread (assuming the OMP NUM THREADS=8). The second parallel prints 2 messages, 1 per thread because we set the value of num_threads to 2. The third parallel prints 3 lines, one per thread because we added the clause num_threads(3) in the pragma declaration to use only 3 threads. The fourth parallel prints 2 messages, 1 per thread because the number of threads is set to 2 before the second parallel. Finally, the fifth parallel prints only one line because the clause if(0) int the pragma declaration always returns false so the print is only executed by one thread.

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

Between 16 and 19 messages. The fifth parallel prints a message per thread and the number of threads depend of the random number generated between 1 and 4.

4.data sharing.c

1. Which is the value of variable x after the execution of each parallel region with different data-sharing attribute (shared, private and firstprivate)?

The value of the variable x after the first parallel(shared) is 8 but depending on the execution can be different, after the second parallel(private) is 0 and after the third parallel(firstprivate) is 0. That's because in the first parallel(shared) the threads can read the value of x before the previous thread increment in 1 that value, so the value of x after the execution can be different depending of the execution of the program. This can't happen in the second and the third parallel because with the clauses private and firstprivate a copy of the variable x is created and used by each thread so the value of the original x is not modified.

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

We can add the pragma omp atomic inside the pragma omp parallel shared(x), so the value of x will be always 8 because only one thread could increment the value of x at the same time. There is another possible solution that is removing the clause shared in the pragma declaration and putting reduction(+:x), with this, the value of x will be always 8 because each thread will have a local variable x so when all the threads finish the local variable of each thread will be added and the original variable x will have that number.

5.parallel.c

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

The program prints 20 messages:

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 0 Iter 0
Thread ID 0 Iter 4	Thread ID 0 Iter 8
Thread ID 0 Iter 12	Thread ID 0 Iter 16
Thread ID 2 Iter 2	Thread ID 2 Iter 6
Thread ID 2 Iter 10	Thread ID 2 Iter 14
Thread ID 2 Iter 18	Thread ID 1 Iter 1
Thread ID 1 Iter 5	Thread ID 1 Iter 9
Thread ID 1 Iter 13	Thread ID 1 Iter 17

Depending of the execution we will have different results because all the threads read and write the same variable i.

2. What needs to be changed in the directive to ensure that each thread executes the appropriate iterations?.
We have to add “pragma omp parallel private(i)” before the loop so each thread will have a local copy of the variable i and all the threads will execute the same number of iterations.

6.datarace.c (execute several times before answering the questions)
1. Is the program always executing correctly?
The program sometimes is not executed correctly because all the threads share the same variable x and a thread can read the value x before being incremented by the previous thread, so the results will be different depending of the execution.

2. Add two alternative directives to make it correct. Which are these directives?
We can use “pragma omp atomic” inside the loop before increment the variable x, so the increment will be only done by one thread at the same time and the execution always will be correct.

We can also add the clause reduction(+:x) in the pragma omp parallel declaration to create a local variable x for each thread and when all the threads finish their work, the local variables will be added and the original x variable will have that number.

7.barrier.c
1. Can you predict the sequence of messages in this program? Do threads exit from the barrier in any specific order?
Yes, we can predict the sequence of messages because the time that the thread will be sleeping depends of his id number so first will appear 4 “going to sleep” messages with no order, then will appear 4 “wakes up” messages ordered by id of thread and finally will appear 4 “we are all awake” messages with no order.

No, the barrier just forces the thread to wait until all the other threads finish so when all the threads finish depending of the execution the result will be different.

B) Worksharing
1.for.c
1. How many iterations from the first loop are executed by each thread?
Each thread executes 2 iterations of the loop. That’s because “pragma omp for” distributes the 16 iterations of the loop between the 8 threads.

2. How many iterations from the second loop are executed by each thread?
Threads 0,1,2 execute 3 iterations, the others 2 iterations. That’s because the distribution method of “pragma omp parallel for".

3. Which directive should be added so that the first printf is executed only once by the first thread that finds it?.
We have to add the directive pragma omp single before the first printf so only the first thread will print that message.

2.schedule.c
1. Which iterations of the loops are executed by each thread for each schedule kind?
On the first loop there is a static distribution with no chunk specification that distributes equally the number of iterations between the threads, thread 0 gets the firsts 4 iterations, thread 1 the next 4 iterations and thread 2 the last 4 iterations.

On the second loop there is a static distribution with value 2 in the chunk clause that distributes equally in packages of 2 between the threads.

On the third loop there is a dynamic distribution with value 2 in the chunk clause that distributes in packages of 2 iterations between the threads assigning them to the first thread that arrives.

On the fourth loop there is a guided distribution with value 2 in the chunk clause that distributes the iterations between the threads in packages until no chunks remains to be assigned.

3.nowait.c
1. How does the sequence of printf change if the nowait clause is removed from the first for directive?
If we remove the nowait clause from the first for directive, all the threads will wait at the end of the loop until all the threads finish their work due to the barrier that is implicit in the pragma omp for directive so the first loop messages will be printed before the messages of the second loop.

2. If the nowait clause is removed in the second for directive, will you observe any difference?
If we remove the nowait clause from the second loop clause, we won't observe any difference because the program ends after that loop.

4.collapse.c
1. Which iterations of the loop are executed by each thread when the collapse clause is used?
The collapse clause says how many loops are associated with the loop construction so the number of iterations will be 25 because n = 5 and n*n = 25. The iterations are distributed equally between the 8 threads due to the "pragma omp parallel for" directive so the thread 0 will execute 4 iterations and the others 3 iterations.

2. Is the execution correct if the collapse clause is removed? Which clause (different than collapse) should be added to make it correct?.
The execution is not correct if we remove the collapse clause because the threads share variables i and j so the program executes less iterations that it should. In order to fix this, we have to add the clause private (j) to make a local copy of that variable in each thread so all the iterations of the loop will be executed.

5.ordered.c
1. How can you avoid the intermixing of printf messages from the two loops?
We can avoid the intermixing of printf messages from the two loops removing the clause nowait from the first loop so the threads will wait until all the threads finish the first loop and then the second loop will be executed.

2. How can you ensure that a thread always executes two consecutive iterations in order during the execution of the first loop?
If we put the value 2 inside the schedule clause of the first loop, the iterations will be assigned to the threads in packets of 2 iterations so the thread will always execute two consecutive iterations in order.

6.doacross.c
1. In which order are the ”Outside” and ”Inside” messages printed?
First “Outside” messages are printed and when i-2 is loaded then the “Inside” message is printed.

2. In which order are the iterations in the second loop nest executed?
First it will de executed the first iteration(i,j = 1,1) and then there is no order but only can be executed an iteration i,j if (i-1,j) or (i,j-1) was load before.

3. What would happen if you remove the invocation of sleep(1). Execute several times to answer in the general case.
The only thing that change if we remove the sleep(1) is that it’s more possible that the iteration with i = 4 appears before because there is no time between the executions so with the sleep(1) tried to establish a order with the iterations with less value of i first because it was made to waste time between the iterations.

C) Tasks
1.serial.c
1. Is the code printing what you expect? Is it executing in parallel?
Yes, the program prints the Fibonacci for the numbers between 0 and 24 included as we expected.

No, it’s not executing in parallel, thread 0 computes the Fibonacci for all of the numbers of the list.

2.parallel.c

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

No, the Fibonacci sequence of the 25 numbers in the list are not correct.

It's wrong because the values printed are 4 times bigger that they should.

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

We have to add the directive "pragma omp single" after the "pragma omp parallel.." directive in order to let only one thread to call processwork at the same time.

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?

If we remove the firstprivate clause form the task directive the result is the same because in tasks the variables are firstprivate by default.

If we remove the firstprivate from the parallel directive the result is also the same, so the firstprivate clause is redundant for the same reason.

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

The program breaks because if p is not firstprivate all the threads use that variable simultaneously so when the program updates the value of p with p->next, it will be a Segmentation fault.

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

Because it's not executed in parallel so only one thread do the work.

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.

First of all one of the 4 threads that execute the parallel region executes the code due to the "pragma omp single" declaration prints the line "I am thread x and going to create T1 and T2" and create two task (T1 and T2). The main thread prints "I am still thread x after creating T1 and T2, ready to enter in the taskwait" and waits until the other tasks finish. The first task created by the thread x goes to sleep 5 seconds and the other one creates a task (T3) and one of the tasks go to sleep 10 seconds and the other creates another task (T4). After that a message "thread y finished the execution of task creating T3 and T4" will be printed by main task that created T4. That main task that created T4 executes the loop body and the thread that went to sleep for 5 seconds wakes up and execute the loop body too. Now the thread x (who created T1) exit from the taskwait and execute the loop body too. After that the thread that went to sleep 10 seconds wakes up and execute the loop body like all the other threads.

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.

All overheads expressed in microseconds

Nthr	Overhead	Overhead per thread
2	1.7685	0.8842
3	1.7314	0.5771
4	2.2573	0.5643
5	2.8974	0.5795
6	2.9367	0.4894
7	2.6975	0.3854
8	3.0968	0.3871
9	3.5233	0.3915
10	3.3078	0.3308
11	4.0173	0.3652
12	3.4189	0.2849
13	3.9637	0.3049
14	4.2337	0.3024
15	3.9318	0.2621
16	4.7325	0.2958
17	4.8284	0.2840
18	4.3512	0.2417
19	4.5775	0.2409
20	4.4511	0.2226
21	4.8696	0.2319
22	5.0773	0.2308
23	5.2068	0.2264
24	5.8645	0.2444

As we can see in the list of results, the magnitude of the overhead is not constant and the order arriving to 0,2 microseconds approximately. .

Also, for each thread that it's added to the program, it usually goes up, while this overhead time goes up, we can also observe that the time needed by each thread decreases.

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.

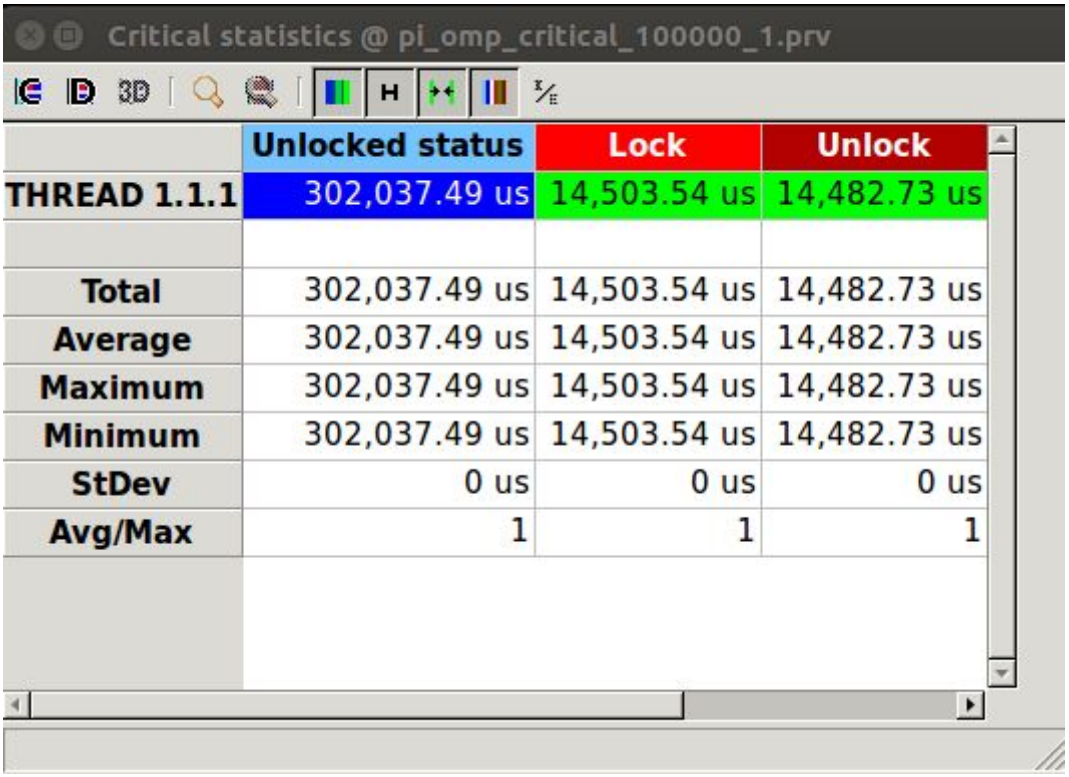
All overheads expressed in microseconds

Ntasks	Overhead per task
2	0.1188
4	0.1148
6	0.1161
8	0.1157
10	0.1216
12	0.1245
14	0.1238
16	0.1255
18	0.1256
20	0.1245
22	0.1237
24	0.1227
26	0.1219
28	0.1207
30	0.1209
32	0.1210
34	0.1212
36	0.1208
38	0.1203
40	0.1207
42	0.1198
44	0.1195
46	0.1193
48	0.1194
50	0.1208
52	0.1203
54	0.1193
56	0.1192
58	0.1190
60	0.1185
62	0.1189
64	0.1191

The order of the magnitude is microseconds and aside some minimal difference, we can say that the overhead per task is constant.

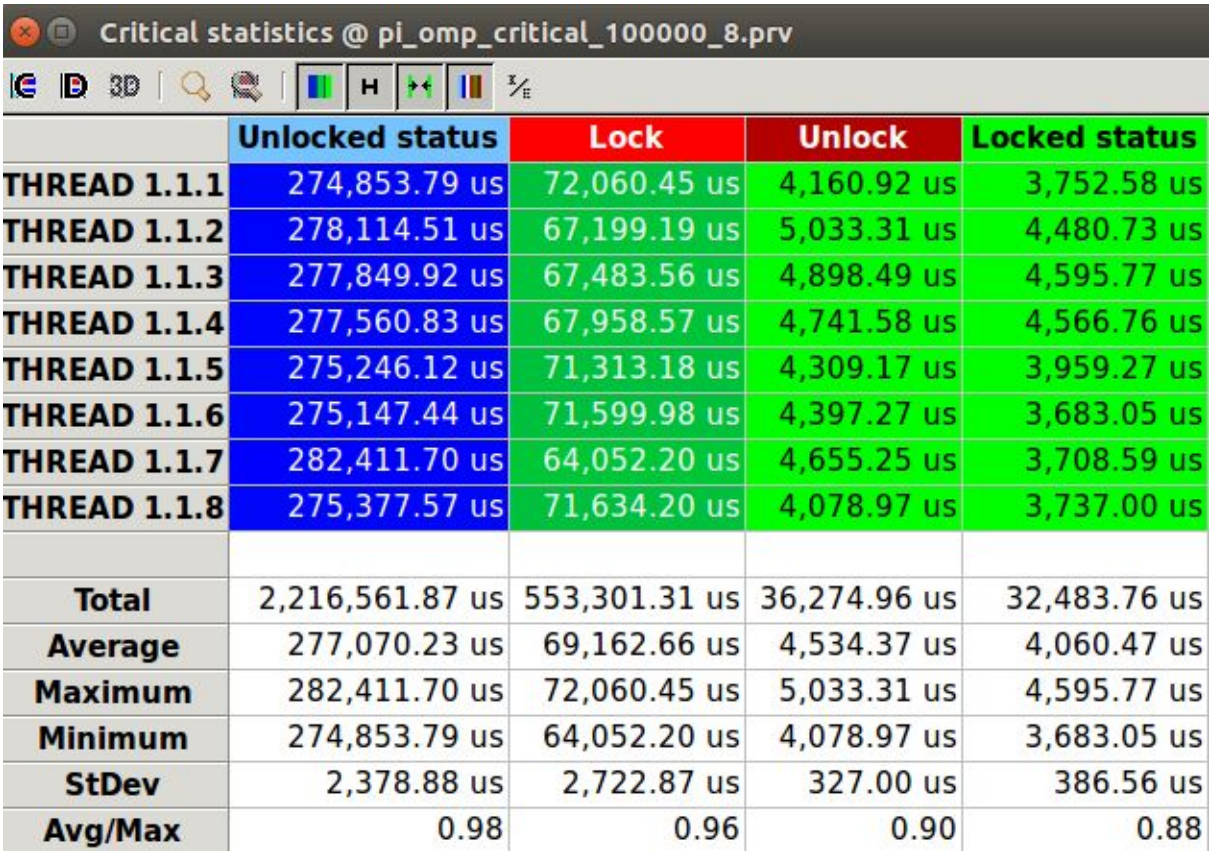
3. Which is the order of magnitude for the overhead associated with the execution of critical regions in OpenMP? How is this overhead decomposed? How and why does the overhead associated with critical increase with the number of processors? Identify at least three reasons that justify the observed performance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

Critical execution with 1 thread



	Unlocked status	Lock	Unlock
THREAD 1.1.1	302,037.49 us	14,503.54 us	14,482.73 us
Total	302,037.49 us	14,503.54 us	14,482.73 us
Average	302,037.49 us	14,503.54 us	14,482.73 us
Maximum	302,037.49 us	14,503.54 us	14,482.73 us
Minimum	302,037.49 us	14,503.54 us	14,482.73 us
StDev	0 us	0 us	0 us
Avg/Max	1	1	1

Critical execution with 8 threads



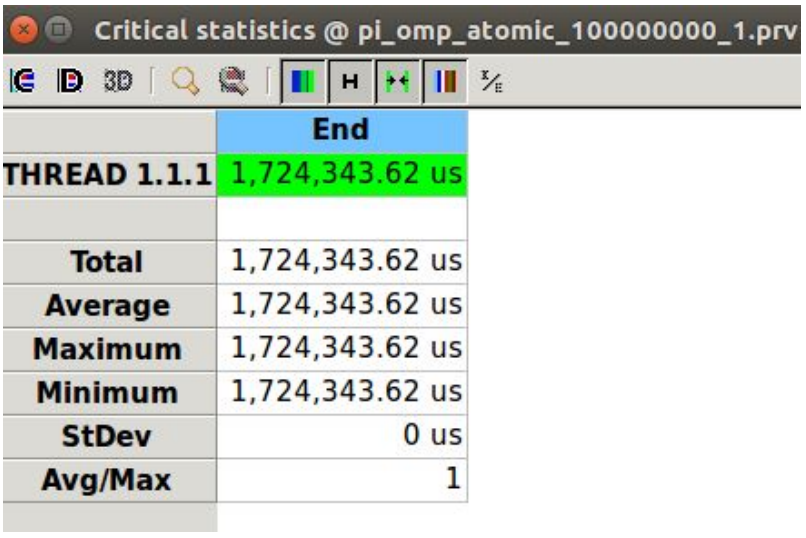
	Unlocked status	Lock	Unlock	Locked status
THREAD 1.1.1	274,853.79 us	72,060.45 us	4,160.92 us	3,752.58 us
THREAD 1.1.2	278,114.51 us	67,199.19 us	5,033.31 us	4,480.73 us
THREAD 1.1.3	277,849.92 us	67,483.56 us	4,898.49 us	4,595.77 us
THREAD 1.1.4	277,560.83 us	67,958.57 us	4,741.58 us	4,566.76 us
THREAD 1.1.5	275,246.12 us	71,313.18 us	4,309.17 us	3,959.27 us
THREAD 1.1.6	275,147.44 us	71,599.98 us	4,397.27 us	3,683.05 us
THREAD 1.1.7	282,411.70 us	64,052.20 us	4,655.25 us	3,708.59 us
THREAD 1.1.8	275,377.57 us	71,634.20 us	4,078.97 us	3,737.00 us
Total	2,216,561.87 us	553,301.31 us	36,274.96 us	32,483.76 us
Average	277,070.23 us	69,162.66 us	4,534.37 us	4,060.47 us
Maximum	282,411.70 us	72,060.45 us	5,033.31 us	4,595.77 us
Minimum	274,853.79 us	64,052.20 us	4,078.97 us	3,683.05 us
StDev	2,378.88 us	2,722.87 us	327.00 us	386.56 us
Avg/Max	0.98	0.96	0.90	0.88

The order of the magnitude is nanoseconds and the overhead is decomposed in unlocked status, lock, unlock and locked status.

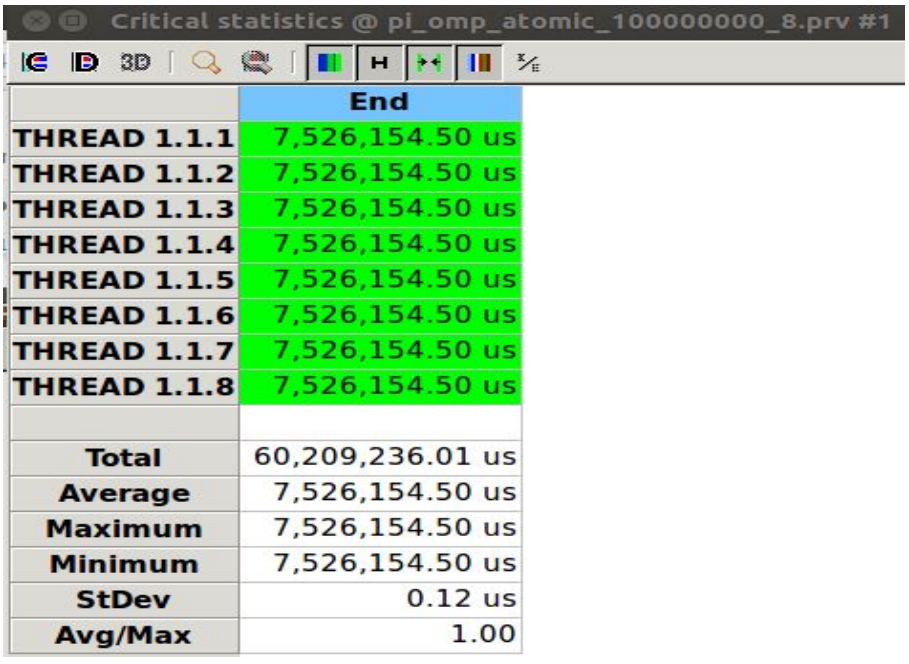
Three causes that justifies the observed performance degradations are:

- Synchronize
- Competition for the sum variable, locking and unlocking it as needed.
- Creation and ending of the different tasks the threads makes.

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.



	End
THREAD 1.1.1	1,724,343.62 us
Total	1,724,343.62 us
Average	1,724,343.62 us
Maximum	1,724,343.62 us
Minimum	1,724,343.62 us
StDev	0 us
Avg/Max	1



	End
THREAD 1.1.1	7,526,154.50 us
THREAD 1.1.2	7,526,154.50 us
THREAD 1.1.3	7,526,154.50 us
THREAD 1.1.4	7,526,154.50 us
THREAD 1.1.5	7,526,154.50 us
THREAD 1.1.6	7,526,154.50 us
THREAD 1.1.7	7,526,154.50 us
THREAD 1.1.8	7,526,154.50 us
Total	60,209,236.01 us
Average	7,526,154.50 us
Maximum	7,526,154.50 us
Minimum	7,526,154.50 us
StDev	0.12 us
Avg/Max	1.00

The order of magnitude for the overhead is nanoseconds.

In this case, we can assume that due to the atomic memory access, the more threads we generate, the more time the program takes to execute. This longer execution time is caused because the atomic parallelism locks de variable and doesn't allow other threads to use it, in this case, the "sum" variable, which is locked and unlocked by every thread, stopping this way the flow of the parallelism.

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.

./run-omp.sh pi_omp_sumvector 1000000000 8
Total execution time: 5.764198s

./run-omp.sh pi_omp 1000000000 8
Total execution time: 1.484433s

./run-omp.sh pi_omp_padding 1000000000 8
Total execution time: 1.094037s

The increase is caused because false sharing, just makes the program share the same memory regions which use the same cache lines, this causes more loading times of data therefore more execution time for the program, with padding, this is fixed because it doesn't make the program share same cache lines, the opposite, the elements accessed by different threads are located in different cache lines.
This is accomplished making a matrix instead of a vector, this way, and using only the first column, we can assure each row of the matrix will fill a cache line.

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?

Pi version	1 thread	8 threads	SpeedUp
pi_seq	0.790829s	-	1
pi _omp	0.792109s	0.139754s	5'658721
pi_omp_atomic	1.470318s	6.332194s	0'12489
pi_omp_critical	1.793339s	47.327983s	0'016710
pi_omp_padding	0.792573s	0.110964s	7'126897
pi_sumvector	0.788311s	0.405553s	1'950000

We performed 5 executions per each version of the pi program. With the results obtained we can observe that the atomic and the critical versions have a speedup lesser than 1 due to the time of overheads. The omp, padding and sumvector versions have a speedup greater than 1. This means that the overhead generated with the parallelization has an important impact to the execution time of a program.

Optional Part: Execution times of pi

Code	Correct?	Execution Time	Speed-up*
v0	yes	0.083403777 seconds	-
v1	no	0.215679043 seconds	0'386703
v2	no	0.201899414 seconds	0'413096
v3	no	0.022110099 seconds	3'772202
v4	yes	3.501163435 seconds	0'023822
v5	yes	0.958090429 seconds	0'087052
v6	yes	0.022646570 seconds	3'682843
v7	yes	0.025684039 seconds	3'247299
v8	yes	0.036857857 seconds	2'262849
v9	yes	0.509560686 seconds	0'163677
v10	yes	0.023208903 seconds	3'593611
v11	yes	0.021988583 seconds	3'793049
v12	no	0.029155128 seconds	2'860689
v13	yes	0.023983076 seconds	3'477609
v14	no	0.252159094 seconds	0'330758
v15	yes	0.058506714 seconds	1'425541
v16	yes	50.354981153 seconds	0'001656
v17	yes	0.035136876 seconds	2'373682

*Speed-up vs the version 0 of the pi program.