2nd Deliverable

par1103
Gallego, Victor
Iqbal, Omair
2015-2016
March 2016

Part I: OpenMP questionnaire

When answering to the questions in this questionnaire, please DO NOT simply answer with yes, no or a number; try to minimally justify all your answers. Sometimes you may need to execute several times in order to see the effect of data races in the parallel execution.

A) Basics

1. hello.c:

1. How many times will you see the "Hello world!" message if the program is executed with "./1.hello" ?
   We can see the message “Hello world” printed 24 times, once per line.

2. Without changing the program, how to make it to print 4 times the "Hello World!" Message?
   We will modify the variable OMP_NUM_THREADS and execute the program as follows:
   par1103@boada-1:~/lab2/openmp/basics$ 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? (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, 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?
   No, because we can’t guarantee which thread is the first to get executed.
   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?
   32 lines with “Hello world... “ get printed.
2. If the if(0) clause is commented in the last parallel directive, how many "Hello world ..." lines are printed on the screen?
   Between 32 and 35 lines are being printed in every execution because the commented line (if(0)) disappears and that allows to execute another task and print the

“Hello word from fifth parallel!” mesage from the fifth parallel execution.

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: 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?

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?

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?

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?
   No, there are no errors in any executions.
2. Add two alternative directives to make it correct. Which are these directives?
   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?
   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.
   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?
   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?
   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?
   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?

   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:

   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?

   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?

   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?

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)

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 isn't correct because the induction variables (i and j) aren't private anymore and that creates a race condition.

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?

Its 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?

No is not printing the expected result.

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?

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?

Nothing, the execution will be the same as it was with the clause because we still have the first pragma.

Segmentation Fault the reason is explained in the question 4

They are redundant because both firstprivates do the same

4. Why the program breaks when variable p is not firstprivate to the task? 5. Why the firstprivate clause was not needed in 1.serial.c?

Because we try to acces p when its value is NULL (due to the fact that p is in a race condition zone).

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

The order of magnitude of each thread we add is approximately 250 nanoseconds, when we reach 24 threads the overhead becomes close to being linear.

The reason behind this is that there is a constant cost which does not depends on the number of threads we create, and there is a proportional cost additional for each thread. When we create around 2024 threads at once, the cost becomes nearly negligible and only the proportional cost remains (250 nanoseconds), which would be the real cost of adding a thread.

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.

Version	1 Processor	8 Processors [1]

[1] Table 1: Execution time of pi_omp.c and pi_omp_critical.c with 100.000.000 iterations. Overhead is obtained by subtracting pi_omp time from pi_omp_critical time

pi_omp.c	0.802267 s	0.161852 s
pi_omp_critical.c	1.851882 s	18.832908 s
Overhead	1.049615 s	18.671056

If we take the execution with 1 processor as a reference, we see that critical region cause a overhead of 1,049615 seconds. We know that the critical program accesses one critical region in each iteration, therefore the order of magnitude for one critical region is 1.049615/100000000 = 1.049615 * 10^-8 seconds with 100000000 iterations.
The overhead is decomposed into 4 possibles statues: lock, locked, unlock and unlocked, according to the paraver trace for pi_omp_critical with 1 thread.

The lock status corresponds to the time in which the processor waits to acquire the lock, the locked status corresponds to the time which the processor executes the critical part of the code, the unlock status corresponds to the time in which the processor releases the lock and the the unlocked status corresponds to the time in which the processor executes code which has nothing to do with the critical region.
The overhead associated with critical increases due to the massive amount of time that each threads spends waiting to acquire the lock of the critical region when the number of processors increases.
There is no clear pattern to define the increase, but we guess that the use of more processors on this critical program will make a higher execution time. Three reasons to explain this are the increase of fork/join region creations, the unbalanced load between the processors and the overhead of the critical regions.

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.

Version	1 Processor	2 Processors	8 Processors
pi_omp.c	0.78 s	0.40 s	0.09 s
pi_omp_atomic.c	1.41 s	6.00 s	8.39 s
Overhead	0.63 s	5.60 s	8.30 s

Table 2. Execution time of pi_omp.c and pi_omp_atomic.c with 100000000 iterations.

Having an atomic section in the code with X number of iterations ends up adding an overhead, which we can see in the single processor version. In the 2 processor versions we can see that the overhead is much worse, this is because in this versions the coherence protocol to manage the atomic regions in several processors comes into play.

That's why the difference between 1 and 2 is much more noticeable than the difference from 2 to 8. The order of magnitude of the overhead of an atomic region (looking at the single processor version) is 0.63s/100.000.000 atomic regions = 0.63 * 10^-8 s.

4. 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.

Version	1 Processor	8 Processor
pi_omp_sumvector.c	0.79 s	0.54 s
pi_omp_padding.c	1.22 s	0.08 s
pi_omp.c	0.78 s	0.07 s

Table 3. Execution time of pi_omp.c, pi_omp_padding.c and pi_omp_sumvector.c with 100000000 iterations in 1 and 8 processors.

We can see that pi_omp_sumvector.c execution time with 1 processors is almost the same as in pi_omp.c, because false sharing does not appear when we use a single processor, since the false sharing conflicts begin to occur when we use 2 processors or more.
In 8 processor executions, the execution time for pi_omp_sumvector.c is not T1/8 as expected, that is because of the additional memory access time caused by processors having to update or invalidate other processors cache line when they access the sumvector.
For the pi_omp_padding.c, we can see that the T8 is approximately T1/8 and that is because the padding (allocate each sum variable in different cache lines to avoid false sharing) is well done, so each processor works with its own cache and no memory exchange between processors is required.
For the sequential version we can see that it takes a little bit longer.

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?
We perform 5 executions of each version and we pick up the average of the 3 intermediate values

Version	1 Processor	8 Processors	Speed-up

pi_seq.c	0.800439 s	----	1
pi_omp.c (sumlocal)	0.806098 s	0.196737 s	4.097338
pi_omp_critical.c	1.856649 s	15.329345 s	0.121117
pi_omp_lock.c	1.821951 s	51.924370 s	0.035088
pi_omp_atomic.c	1.463832 s	8.593183 s	0.170367
pi_omp_sumvector.c	0.802320 s	0.701268 s	1.144098
pi_omp_padding.c	0.805474 s	0.196723 s	4.094457