PAR Laboratory Assignment

Lab 2: OpenMP programming model and analysis of overheads

PAR 2312
Fèlix Arribas Pardo
Ana Mestre Borges

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"?

As many as processors/threads working. In that case, 24.

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

You have to change the number of threads. There are several ways to do that:

- **Changing the code**: At the end of `#pragma omp parallel` add `num_threads(<number_of_threads>)` in this case 4.
- **Setting the OpenMP environment variable**: Execute
  `$ set OMP_NUM_THREADS=<number_of_threads>` or
  `$ export OMP_NUM_THREADS=<number_of_threads>`.

Since we cannot touch the source code of `1.hello.c` we are doing:

```
$ export OMP_NUM_THREADS=4
```

2.hello.c: Assuming the OMP NUM THREADS variable is set to 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?

It's not correct, we need to initialize `id` to avoid the warning and add a `critical`.

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

No, the order depends on which processor takes the task and when.

Sometimes the message is intermixed because one task is slow and other tasks print between the slow task's print function execution.

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 lines:

- **Eight** for the **first** printf
- Then **two** for the **second** because it is changed the number of OpenMP threads to 2 in `omp_set_num_threads(2);`
- **Three** for the **third** `printf`. In the `#pragma` clause the number of threads is 3: `#pragma omp parallel num_threads(3)`
- **Two** again for the **fourth** because the 3 threads configuration was only for the third `printf`.
- **One** for the **fifth**. The `printf` is executed with more than one thread if the `if` clause is true, which never happens because 0 is not true.

8 + 2 + 3 + 2 + 1 = 16

2. If the if(0) clause is commented in the last parallel directive, how many "Hello world ..." lines are printed on the screen?
From 16 to 19. Depending on the random number. The **fifth** `printf` have from 0 to 3 threads. The operation to set the threads that will execute the last `printf` is:

```
rand() % 4 + 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)?
Shared: Usually it is 8
Private and firstprivate: Always 0

2. What needs to be changed/added/removed in the first directive to ensure that the value after the first parallel is always 8?.
Add `#pragma omp barrier` inside the parallel region after `x++`.

5.parallel.c
1. How many messages the program prints? Which iterations is each thread executing?
Not 20. The execution ends before the rest of the programs can finish the loop. There are 4 threads. The thread with id = X executes iterations where i%4 equals X.

2. What needs to be changed in the directive to ensure that each thread executes the appropriate iterations?.
Make *i* private: `#pragma omp parallel num_threads(NUM_THREADS) private(i)`

6.datarace.c (execute several times before answering the questions)
1. Is the program always executing correctly?
No, some tasks finish the parallel region before other tasks finish the loop.

2. Add two alternative directives to make it correct. Which are these directives?
`#pragma omp barrier`

Go single
`#pragma omp single`
Create tasks
`#pragma omp task`
Wait the rest of the tasks
`#pragma omp taskwait`

7.barrier.c
1. Can you predict the sequence of messages in this program? Do threads exit from the barrier in any specific order?
You cannot predict the first four messages, but you can predict when they will wake up. Depends on the thread id: 3 x thread_id + 2.

B. Worksharing

1.for.c

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

2 iterations per task. There are 8 threads and N = 16.

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

With N+3 we have 19 iterations. This is two (2.375) iterations per task and three the first 3 tasks.

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

2.schedule.c

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

3 Threads in `#pragma omp parallel for`

Static: The iterations are given to the different threads in order. First thread get from 0 to 3, second from 4 to 7 and the last one from 8 to 11.

Loop 1: (0) gets iteration 0
Loop 1: (0) gets iteration 1
Loop 1: (0) gets iteration 2
Loop 1: (0) gets iteration 3
Loop 1: (2) gets iteration 8
Loop 1: (2) gets iteration 9
Loop 1: (2) gets iteration 10
Loop 1: (2) gets iteration 11
Loop 1: (1) gets iteration 4
Loop 1: (1) gets iteration 5
Loop 1: (1) gets iteration 6
Loop 1: (1) gets iteration 7

Static 2: The iterations are given in order but in group of 2, so iterations 0 and 1 are fro the first thread, 2 and 3 for the second, 4 and 5 for the third and then we start again, 6 and 7 for the first, 8 and 9 for the second and finally 10 and 11 for the third thread.

Loop 2: (0) gets iteration 0
Loop 2: (0) gets iteration 1
Loop 2: (0) gets iteration 6
Loop 2: (0) gets iteration 7
Loop 2: (1) gets iteration 2
Loop 2: (1) gets iteration 3
Loop 2: (1) gets iteration 8
Loop 2: (1) gets iteration 9

Loop 2: (2) gets iteration 4
Loop 2: (2) gets iteration 5
Loop 2: (2) gets iteration 10
Loop 2: (2) gets iteration 11

Dynamic 2: The iterations are given in group of two to the first thread avialable. As soon as it have a free thread, OpenMP gives two iterations to that thread.
Loop 3: (0) gets iteration 0
Loop 3: (0) gets iteration 1
Loop 3: (2) gets iteration 2
Loop 3: (2) gets iteration 3
Loop 3: (2) gets iteration 8
Loop 3: (2) gets iteration 9
Loop 3: (2) gets iteration 10
Loop 3: (2) gets iteration 11
Loop 3: (1) gets iteration 4
Loop 3: (1) gets iteration 5
Loop 3: (0) gets iteration 6
Loop 3: (0) gets iteration 7

Guided 2: Distributes the tasks in groups, increasingly small. The minimum is 2.
Loop 4: (2) gets iteration 0
Loop 4: (2) gets iteration 1
Loop 4: (2) gets iteration 2
Loop 4: (2) gets iteration 3
Loop 4: (0) gets iteration 4
Loop 4: (0) gets iteration 5
Loop 4: (0) gets iteration 6
Loop 4: (0) gets iteration 11
Loop 4: (1) gets iteration 7
Loop 4: (1) gets iteration 8
Loop 4: (2) gets iteration 9
Loop 4: (2) gets iteration 10

3.nowait.c

1. How does the sequence of printf change if the nowait clause is removed from the first for directive?
The loops are executed in order.

2. If the nowait clause is removed in the second for directive, will you observe any difference?
No, because there is no parallel execution after the second loop.

4.collapse.c

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

(3) Iter (2 0)
(3) Iter (2 1)
(3) Iter (2 2)
(4) Iter (2 3)
(4) Iter (2 4)
(4) Iter (3 0)
(1) Iter (0 4)
(1) Iter (1 0)
(1) Iter (1 1)
(0) Iter (0 0)
(0) Iter (0 1)
(0) Iter (0 2)
(0) Iter (0 3)
(5) Iter (3 1)
(5) Iter (3 2)
(5) Iter (3 3)
(2) Iter (1 2)
(2) Iter (1 3)
(2) Iter (1 4)
(7) Iter (4 2)
(7) Iter (4 3)
(7) Iter (4 4)
(6) Iter (3 4)
(6) Iter (4 0)
(6) Iter (4 1)

The iterations are in groups like it was only one loop.

2. Is the execution correct if the collapse clause is removed?
No, only the i is protected. The j is shared.
Which clause (different than collapse) should be added to make it correct?.
`private(j)`

5.ordered.c

1. How can you avoid the intermixing of printf messages from the two loops?
Removing the `nowait` from the first loop

2. How can you ensure that a thread always executes two consecutive iterations in order during the execution of the first loop?
Changing from `schedule(dynamic)` to `schedule(dynamic, 2)`

6.doacross.c

1. In which order are the ”Outside” and ”Inside” messages printed?

The outside is executed whenever the thread is available. The inside is executed after the i-2 iteration is done.

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

The iteration is executed when the left matrix element and the above one are computed.

3. What would happen if you remove the invocation of sleep(1). Execute several times to answer in the general case.

It is faster.

C. Tasks

1.serial.c

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

Yes, fibonacci sequence. It is NOT parallel, there is not even any #pragma omp parallel sentence.

2.parallel.c

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

No, there is task generation in parallel

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

`#pragma omp single` before the 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?

Because each thread uses a different part of the list of results.

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

The program does not break.

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

Because is serial and there is no data sharing.

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.

Thread 2 (T3) creates Thread 0 and Thread 1 (T1 & T2). Then it executes some loop iterations before the taskwait. When the T2 gets out of “siesta” T3 can continue with the execution with T2. Then happens the same with T1.

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.

The overhead is expressed in microseconds and between 1.0 ms and 0.3 ms. It is not constant, when the executions start being parallel there is an overhead to initialize the parallel execution and an overhead for each new thread.
The first one is constant (t_1), the second one depends on the number of threads (t_2):

Overhead = t_1 + num_threads · t_2

Overhead per thread = $\frac{t_1 + num\_threads \cdot t_2}{num\_threads}$

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.

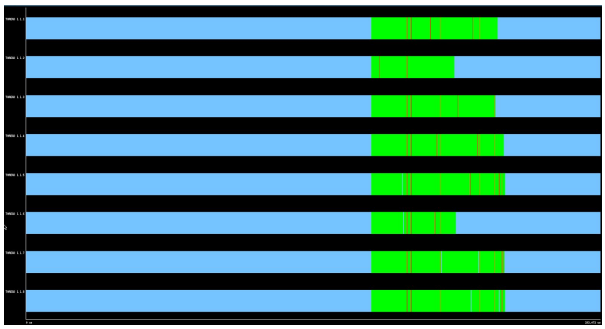
Goes from 0.16 ms to 0.11 ms. Now the overhead of synchronizing the tasks depends on the number of tasks created because of the taskwait. Still not constant.

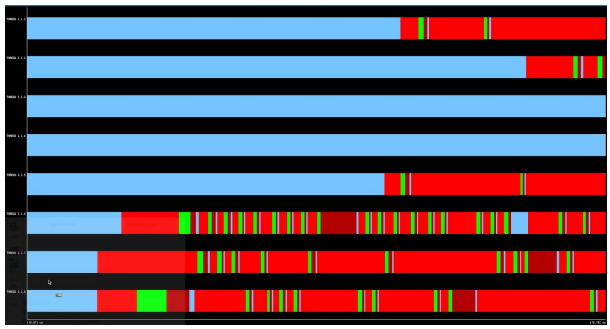
3. Which is the order of magnitude for the overhead associated with the execution of critical regions in OpenMP? How is this overhead decomposed? How and why does the overhead associated with critical increase with the number of processors? Identify at least three reasons that justify the observed performance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

There is a lot of overhead because the threads have to synchronize for the critical part and the tasks wait there for a while. Also the overhead of the creation of the parallel tasks and the overhead of waiting all the tasks to finish.

1. Overhead on task creation and syncronization
2. Overhead on `critical` syncronization
3. The other tasks wait to execute the `critical` section.

Paraver configuration: OpenMP/in_lock.
The first one is the whole execution and in the second one we made zoom to appreciate the critical execution.





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.

The `atomic` also have overhead because of the syncronization and the rest of the tasks waiting to execute the `atomic` code.
On the one hand, `atomic` it better than `critical` because it have in mind the architecture of the computer and is optimized to execute only one statement.
On the other hand, `atomic` it slower than `omp` because of the overhead explained above.

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.

What `sumvector` tries to do is to have no dependencies between tasks but then apears **falsesharing**.
When a processor gets an element of the vector it actually gets all the cache line invalidating the rest of the elements. Then the rest of the tasks cannot access them.

D0	D1	D2	D3	D4	D5	D6	D7
D8	D9	D10	D11	D12	D13	D14	D15

Imagine the first tasks gets D0, then it will block the whole first row. When, for example, task two tries to get D1 it will have to wait until the first task free D1.
But then the rest of task will have to wait for D2, D3, etc.

In `padding` each element of the vector have daft data for the rest of the cache line so any task would block any other element of the vector.

D0							
D1							
D2							
D3							
D4							
D5							
D6							

6. Complete the following table with the execution times of the different versions for the computation of Pi that we provide to you in this 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 pi seq.c. For each version and number of threads, how many executions have you performed?

100 each: We have modified the `submit-omp.sh` script:

```
#!/bin/csh
set i = 0
while ($i < 100)
	./$1 $2 $3 >> $1_$2_$3.txt
	@ i = $i + 1
end
```

Version	1 processor	8 processors	speed-up
pi_seq.c	0.79063352	-	1
pi_omp.c (sumlocal)	0.79072223	0.14150251	5.58742
pi_omp_critical.c	1.80773215	18.01430024	0.04389
pi_omp_atomic.c	1.56609525	8.78051111	0.09004
pi_omp_sunvector.c	0.79006764	0.60589884	1.30489
pi_omp_padding.c	0.79381018	0.13912249	5.68300