PAR — Deliverable for Laboratory Session 2

Víctor Jiménez Arador, Miguel Moreno Gómez

Group 12_{02} — Fall semester 2017/2018

Contents

1 OpenMP questionnaire

1.1 Basics

1.1.1 hello.c

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

24 times (as many as cores in our machine).

Without changing the program, how to make it to print 4 times the `"Hello world!"` message? Writing `export OMP_NUM_THREADS = 4` on the terminal prior to the execution of the program.

1.1.2 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 while it prints `"(Thid1) Hello (Thid2) world!"`, we find that sometimes Thid1 $\neq$ Thid2. We add a `# pragma omp critical` clause inside the `# pragma omp parallel` one to execute the program 1 thread at a time, but because of that it runs the code sequentially.

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

As stated before, lines aren't always printed in the same order because all threads will output these lines the moment they finish (and that happens almost at the same time). They won't appear intermixed with the data sharing modifications; the default code can print the messages intermixed, for the same reasons as before.

1.1.3 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 `"Hello world!"` divided by the parallel regions:

- First: 8 prints, one per thread
- Second: 2 prints, one per thread; we set the number of threads to 2
- Third: 3 prints, one per thread; we set the number of threads to 3 BUT just for this region of code.

- Fourth: 2 prints, one per thread; we set the number of threads to 2 before the second region.
- Fifth: 1 print. The possible results of $rand()\%4$ can be $\{0, 1, 2, 3\}$. If we add 1 to it, then these possible results can be $\{1, 2, 3, 4\}$. The `if (0)` will check if any of the results is 0, otherwise the region will be executed sequentially (i.e. one thread). Because the results of the expression $rand()\%4 + 1$ will never be 0, it will always be executed sequentially, thus providing just 1 print.

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 times up to 20, depending on the result of $rand()\%4 + 1$ (the probability for each case is ideally $\frac{1}{4}$).

1.1.4 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: 8, but rarely 7 (this can happen using shared without a synchronization mechanism)
- private: 0 (because each thread has its own x, in the end they won't merge, and the value of x will be the original one)
- firstprivate: 0 (same as above)

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

As foreshadowed previously, we need to add a synchronization mechanism. In this case, we can add a `atomic` or `critical` mechanism.

1.1.5 parallel.c

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

This program prints between 8 and 20 lines. Ideally, each thread should execute the thread ID modulo 4, but due to the fact that all threads are sharing its i, they can read it, and anomalies may happen depending in which order are variables read.

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

We concatenate a `private(i)` to the `#pragma omp parallel num_threads(NUM_THREADS)` directive.

1.1.6 data_race.c

1. Is the program always executing correctly?
No; while it runs correctly most of the times, sometimes it doesn't happen that $X == n$.

2. Add two alternative directives to make it correct. Which are these directives?
Adding either a `atomic` or a `critical` directive between the for loop and the `++x;` instruction.

1.1.7 barrier.c

1. Can you predict the sequence of messages in this program? Do threads exit from the barrier in any specific order?
The only fixed part of the program is that the order the threads wake up is ordered increasingly, using the expression $3 * myid + 2$.

Thread #3 seems to be the first to exit, but others have no fixed pattern.

1.2 Worksharing

1.2.1 1. for.c

1. How many iterations from the first loop are executed by each thread?
2 iterations per thread, because N = 16 and omp_num_threads = 8, 16/8 = 2

2. How many iterations from the second loop are executed by each thread? 3 iterations for the first 3 threads, 2 for the other (same assumptions as before)

3. Which directive should be added so that the first `printf` is executed only once by the first thread that finds it? `#pragma single` printf("Going to distribute iterations in first loop ...
n");

1.2.2 2. schedule.c

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

- First: $\frac{N=12}{omp\_num\_threads=3} = 4$; Iteration space is divided in for chunks and threads take 4 iterations each (`schedule(static)`).
- Second: $\frac{12}{3} = 4$; Iteration space is divided in chunks of two iterations per thread and each thread keeps getting pairs (`schedule(static, 2)`).

- Third: $\frac{12}{3}$ (ideal case); Iteration space is divided in chunks of two iterations per thread, but this time the first thread to take control gets the pair (`schedule(dynamic, 2)`).
- Fourth: Like the third one, but as the execution advances, the chunk size may decrease. In this case, each thread grabs at least two iterations per thread (becaus ethe minimum chunk size we just defined is 2), but sometimes they grab more than two iterations per thread!

1.2.3 3. nowait.c

1. How does the sequence of `printf` change if the `nowait` clause is removed from the first for directive?
No iteration of the 2nd loop will execute until the last one of the 1st loop finishes.

2. If the `nowait` clause is removed in the second `for` directive, will you observe any difference? Because there are no other executions to wait after the 2nd loop, the execution won't differ.

1.2.4 4. collapse.c

1. Which iterations of the loop are executed by each thread when the `collapse` clause is used? The first iteration will get the first 4 iterations, and the next ones will get them in triplets (de tres en tres)

2. Is the execution correct if the `collapse` clause is removed? Which clause (different than `collapse`) should be added to make it correct?. No, due to the fact that i, j are not private anymore; everyone can touch them and once $i = j = 4$, execution comes to an end. To fix it, we just add `#pragma omp private(i, j)`.

1.2.5 5. ordered.c

1. How can you avoid the intermixing of `printf` messages from the two loops? You can avoid the intermixing by removing the `nowait` clause.

2. How can you ensure that a thread always executes two consecutive iterations in order during the execution of the first loop? Change the (dynamic) of the first loop for (dynamic, 2)

1.2.6 6. doacross.c

1. In which order are the "Outside" and "Inside" messages printed? Outside has no specific order, Inside executes once the iteration i-2, for i = (1..n) has executed, odds depend on odds and evens depend on evens, but there-s no relation between both groups.

In which order are the iterations in the second loop nest executed? There are 7 diagonals (from top-right to bottom-left). The execution from each diagonal runs in the same order (due to dependencies), but the execution inside each diagonal has no strict order.

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

Instead of iterating through each diagonal, the matrix is iterated column by column.

1.3 Tasks

1.3.1 1. serial.c

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

Yes, it's printing the Fibonacci numbers from fib(1) to fib(25). It's not executing in parallel because all numbers are computed by a single thread. Reason: there's no parallel directive anywhere in the code.

1.3.2 2. parallel.c

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

No, it's printing each Fibonacci number multiplied by the number of threads (in this case, 4 times).

2. Which directive should be added to make its execution correct?.
`#pragma omp single` after the first `#pragma omp parallel ...` directive.

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?

Removing the one inside the task directive produces the same result (without point 2's modifications). Removing both produces a Segmentation fault. It's redundant because `firstprivate` is implicitly set once we define a task.

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

Because it's either accessing a null pointer in memory or going beyond boundaries.

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

Because the code is serial, executing in multiples threads would be redundant and there's no need to protect its data.

1.3.3 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`.

The program executes the matrix by iterating anything above its diagonal and has dependencies from left to right, and from up to down.

2 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.
It's around $0,25 - (0,4 * num_{threads})$ (in μs).

Executing `./pi_omp_parallel 6 20`, it prints this result:

Num. thread	Overhead	Overhead per thread
2	0.8960	0.4480
3	0.8696	0.2899
4	1.0480	0.2620
5	1.1993	0.2399
6	1.3350	0.2225
7	2.5633	0.3662
8	2.8847	0.3606
9	3.1778	0.3531
10	3.4629	0.3463
11	3.8966	0.3542
12	3.8387	0.3199
13	3.9493	0.3038
14	4.2808	0.3058
15	4.1046	0.2736
16	4.2489	0.2656
17	4.7767	0.2810
18	4.3730	0.2429
19	4.7494	0.2500
20	4.4406	0.2220

(All overheads expressed in μs)

It's not constant. Although it tends to rise with the number of threads, making it a linear relation, sometimes doesn't follow that trend.

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 It's around $0,25 - (0,4 * num_{threads})$ (in μs).

Executing `./pi_omp_parallel 6 20`, it prints this result:

Num_{tasks}	**Overhead per task**
2	0.1268
4	0.1169
6	0.1159
8	0.1152
10	0.1223
12	0.1242
14	0.1238
16	0.1242
18	0.1238
20	0.1229
22	0.1218
24	0.1212
26	0.1208
28	0.1196
30	0.1197
32	0.1194
34	0.1204
36	0.1197
38	0.1193
40	0.1190
42	0.1187
44	0.1185

Num_{tasks}	**Overhead per task**
46	0.1183
48	0.1181
50	0.1186
52	0.1184
54	0.1184
56	0.1182
58	0.1179
60	0.1177
62	0.1177
64	0.1178

(All overheads expressed in μs)

It's not constant, yet the task overhead difference is not big at all. Although it tends to decrease too little with the number of threads, making it a linear relation (yet really flat), sometimes doesn't follow that trend. For example, thread # 34 has a little bump compared to threads #32 and #36.

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. It's around 1,5 μs per thread.

```
par1202@boada-1:~/lab2/overheads\$ ./pi_omp 100 4
Total execution time: 0.004735s
Number pi after 100 iterations = 3.141600986923125

par1202@boada-1:~/lab2/overheads\$ ./pi_omp 100 8
Total execution time: 0.010634s
Number pi after 100 iterations = 3.141600986923125

par1202@boada-1:~/lab2/overheads\$ ./pi_omp_critical 100 4
Total execution time: 0.004984s
Number pi after 100 iterations = 3.141600986923125

par1202@boada-1:~/lab2/overheads\$ ./pi_omp_critical 100 8
Total execution time: 0.007757s
```

```
Number pi after 100 iterations = 3.141600986923126
```

This overhead is produced by the need of every thread adding his sumlocal result to sum alone. So as many threads we add, the much the time will be increased, as the rest of the cores have to wait to update their results. The three reasons could be:

- Adding their results separately (so waiting for the rest).
- Fork and join overheads.
- The #pragma omp for has to divide the loop for the threads.

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. Less than 1 $\mu s per thread$.

```
par1202@boada-1:~/lab2/overheads\$ ./pi_omp_atomic 100 8
Total execution time: 0.005623s
Number pi after 100 iterations = 3.141600986923125

par1202@boada-1:~/lab2/overheads\$ ./pi_omp_atomic 100 16
Total execution time: 0.046216s
Number pi after 100 iterations = 3.141600986923125
```

It increases for the same reason that critical does, as the add operation has to be done serially, but generally the overhead is smaller.

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`. Around 1 μs per thread.

```
par1202@boada-1:~/lab2/overheads\$ ./pi_omp_sumvector 100 8
Total execution time: 0.005929s
Number pi after 100 iterations = 3.141600986923125
```

```
par1202@boada-1:~/lab2/overheads\$ ./pi_omp_sumvector 100 16
Total execution time: 0.016399s
Number pi after 100 iterations = 3.141600986923125
```

This increase is due to the threads are accessing the same vector and, although they are not accessing the same address, there's a proximity that is respected. In the padding one, the overhead is execution time is reduced because the sumvector vector is transformed into a matrix, which enlarges the space between accesses so that the threads don't have to wait.

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? We did the average of three execution tiems per version, and then taken the average time:

Version	**Avg. T_8**	**Speedup (vs seq)**
`pi_seq`	0.792261	1
`pi_omp`	0.184145	4.3024
`pi_omp_atomic`	6.304443	0.1257
`pi_omp_critical`	47.556841	0.0166
`pi_omp_padding`	0.197987	17.374
`pi_omp_subvector`	0.460361	8.1929

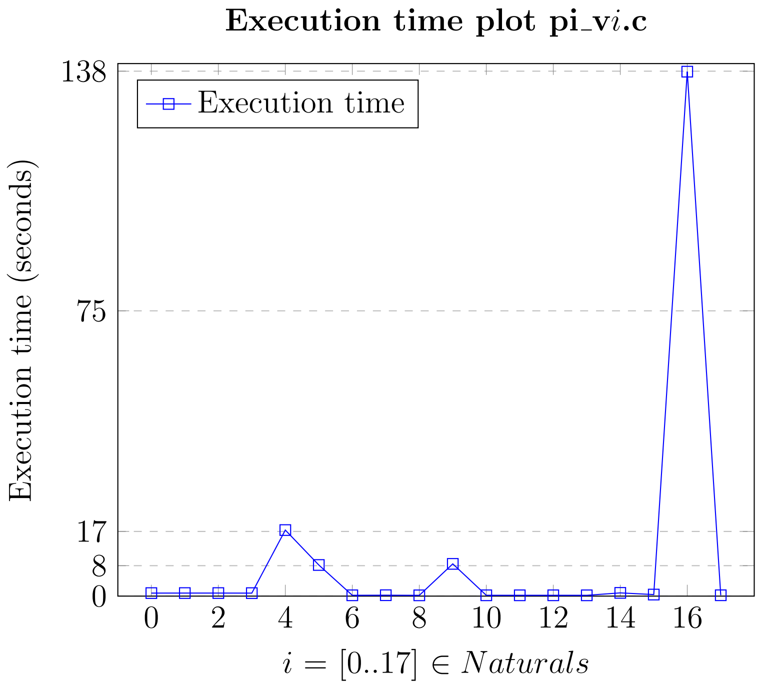
The execution was done using 8 threads and N = 100000000 steps.

3 Optional: execution time for the multiple versions of Pi

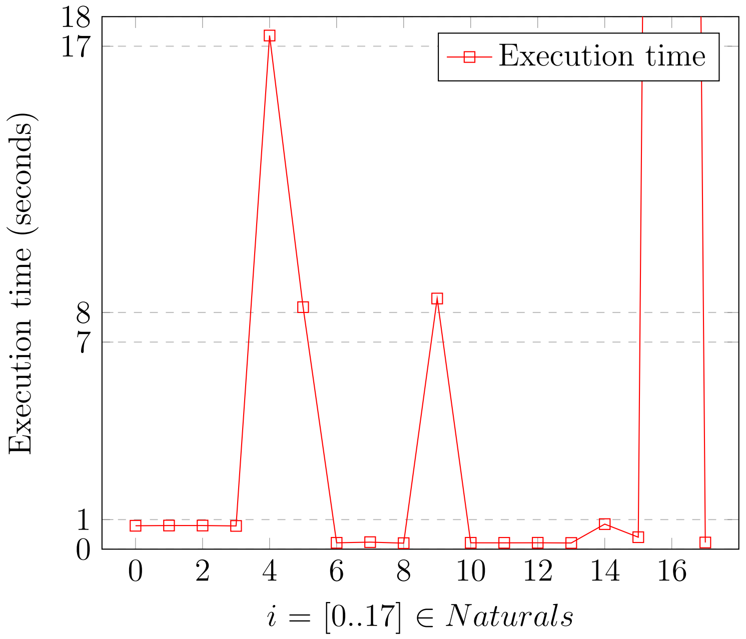
As an optional part for this laboratory assignment, we ask to fill in a table (or draw a graph) with the execution time of the different versions of Pi explored in section 1.2 and the achieved speed-up S4 with respect to the sequential version pi-v0. Which are the most relevant conclusions you extract?

Version	**Time**	**Speedup**	**Version**	**Time**	**Speedup**
V0	0.79134	1	V9	8.47210	0.0934
V1*	0.79886	0.99	V10	0.21516	3.6779
V2*	0.79849	0.99	V11	0.21389	3.6997
V3*	0.78539	1.007	V12*	0.21586	3.6659
V4	17.3609	0.0456	V13	0.20964	3.7748
V5	8.17999	0.0967	V14*	0.84684	0.9344
V6	0.21462	3.6872	V15	0.40473	1.9552
V7	0.23740	3.3333	V16	137.875	0.0057
V8	0.20369	3.885	V17	0.22541	3.5107

(Note that times are represented in *seconds*. An asterisk indicates that the version does *not* return a correct pi output).



Detailed execution time plot pi_vi.c (for times between 0 and 18 seconds)



From V0 up to V3 execution time seems normal for a sequential program. Problems arise in V4 and V5: the critical and atomic directives seem to be quite costly in time, respectively. From V6 until V17 execution time is *almost* four times faster than the previous ones: it fits because it's parallelizing the code using four threads, and the *almost* part is due to overheads and synchronisation. The exceptions are briefly commented:

- **V9:** It uses a dynamic scheduler that grabs one iteration per thread. Considering that there are 100.000.000 iterations, each thread has to fetch it and then run it, causing a huge time expense.
- **V14:** not only its output is not correct, but it also runs slower than the sequential counterpart. Runs slower because of the use of `taskwait` directives.
- **V15:** instead of running almost 2 times as fast as V0, it just runs almost 2 times as fast. The `single` directive is the bottleneck, because instructions bound by it can only be executed by one thread at a time.
- **V16:** This code has an abnormally large execution time. This is because the `#pragma` directives are found inside the `for(...)` loop, and that produces a fork + join at the beginning and the end of each iteration of the loop, respectively. And because those iterations are run by every thread, which causes a huge execution time.