UPC: Facultat de Informàtica de Barcelona

Lab 2: OpenMP programming model and analysis of overheads

PAR Laboratory Assignment

Oriol Moyano Núñez
Martí Ferret Vivó

ID: par2108

10/04/2018

Index

Exercises **3**
A) Basics 3
hello.c 3
hello.c 3
how many.c 3
data sharing.c 3
parallel.c 4
datarace.c 4
barrier.c 5
B) Worksharing 5
for.c 5
schedule.c 5
nowait.c 6
collapse.c 6
ordered.c 7
doacross.c 7
C) Tasks 8
serial.c 8
parallel.c 8
taskloop.c 8
Part II: Parallelization overheads 9

Exercises

A) Basics

hello.c

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

As many as threads are opened by program, 24 in our case executing the code in the boada system in the interactive mode.

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

Using the environment variable OMP_NUM_THREADS when we execute the program. It could be done with the sequence of commands below:

```
export OMP_NUM_THREADS=4
./1.hello
```

hello.c

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, as the variable id is implicitly shared some issues can happen due to overwriting, creating sequences with the same Thid more than twice. This problem could be solved using the data sharing clause private(id), firstprivate(id) or creating the id variable inside the parallel region.

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

No, each thread works asynchrony. Yes, the lines could appear intermixed. The messages don't follow a specific order and each execution could produce different results (orderwise).

how many.c

1. How many "Hello world ..." lines are printed on the screen?

A total of 16 lines are printed on the screen. (8 (*NUM_THREADS=8*) + 2 (*NUM_THREADS=2*) + 3 (*num_threads(3)*) + 2 (*NUM_THREADS=2*) + 1 (*if(0), not parallelized*)) = **16 lines.**

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

A random number between 16 and 19, because the number of threads used is random dependent with a number between 1 and 4, both included. (8 (*NUM_THREADS=8*) + 2 (*NUM_THREADS=2*) + 3 (*num_threads(3)*) + 2 (*NUM_THREADS=2*) + rand[1..4] (*num_threads(rand()%4+1*)).

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 different values in our execution are:

a. Shared: 8 (But it is undefined, could be any value between 1 and 8).
b. Private: 5.
c. Firstprivate: 71.

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

A reduction could be added as in the directive below:

```
#pragma omp parallel reduction(+:x)
```

parallel.c

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

The program prints 26 messages. **26** = 8(*thread id 0*)+7(*thread id 1*)+6(*thread id 2*)+5(*thread id 3*). The first thread executes iterations from 0 to 7, the second one form 1 to 7, the third from 2 to 7 and the fourth from 3 to 7 (the starting iteration correspond to the thread id).

2. Change the for loop to ensure that its iterations are distributed among all participating threads.

The following code distributes 2 iterations to each thread:

```
for (int i=id; i < N; i = i + NUM_THREADS) {
       printf("Thread ID %d Iter %d\n",id,i);
}
```

datarace.c

1. Is the program always executing correctly?

No, because the variable x is shared between all threads.

2. Add two alternative directives to make it correct. Explain why they make the execution correct.

We have found 3 different options:

1. Using the critical directive :

```
#pragma omp parallel private(i)
{
    int id=omp_get_thread_num();
    #pragma omp critical
    for (i=id; i < N; i+=NUM_THREADS) {
        x++;
    }
}
```

2. Using the atom directive (improvement from the last one, because the atom directive locks the other threads for less time):

```
#pragma omp parallel private(i)
{
    int id=omp_get_thread_num();
    #pragma omp atomic
    for (i=id; i < N; i+=NUM_THREADS) {
        x++;
    }
}
```

3. Using a reduction:

```
#pragma omp parallel private(i) reduction(+:x)
{
    int id=omp_get_thread_num();
    for (i=id; i < N; i+=NUM_THREADS) {
        x++;
     }
}
```

barrier.c

1. Can you predict the sequence of messages in this program? Do threads exit from the barrier in any specific order?

We can predict the message blocks, meaning that we know that a block with sleeping messages will be the first, a block with wake ups and barriers will be the second, and a block with awakes will be the third. We can also predict the contents of the middle block as it will always have the same order. Aside from those things, we can only guess the rest of the contents due to parallelization asynchronism. Threads do not exit the barrier in any specific order.

B) Worksharing

for.c

1. How many and which iterations from the loop are executed by each thread? Which kind of schedule is applied by default?

Every thread executes 2 iterations, thread 0 executes i = 0 ad i = 1, thread 1 executes i = 2 and i = 3 ...

The schedule applied by default is static.

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

We could add the single directive so that only one threads executes the print, as shown in the code below:

```
#pragma omp single
printf("Going to distribute iterations in first loop ...\n");
```

schedule.c

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

- **Schedule(static) [default]:** by default the size of chunks is 1 so we know that every thread will execute 4 iterations the 1st thread will execute the first 4 iterations [0..3], the 2nd [4..7] and 3th [8..11].
    - thread 0 executes: 0, 1, 2, 3
    - thread 1 executes: 4, 5, 6, 7
    - thread 2 executes: 8, 9, 10, 11
- **Schedule(static, 2):** each thread gets chunks of 2 iterations in an 'interleaved' way, meaning that the scheduler gives the first bundle to the first thread, the second bundle to the second thread and so on. In our code:
    - thread 0 executes: 0, 1, 6, 7

    - thread 1 executes: 2, 3, 8, 9
    - thread 2 executes: 4, 5, 10 ,11
- **Schedule(dynamic, 2):** we can not know which iteration will be executed by each thread, because the iteration are assigned dynamically in chunks of 2. In our case we got this results:
    - thread 0 executes: 4, 5
    - thread 1 executes: 2, 3, 6, 7
    - thread 2 executes: 0, 1, 8, 9, 10, 11
- **Schedule(guided, 2):** knowing that a guided scheduling is similar to a dynamic one (with a decrease in chunk size when iterations are allocated), we can deduce that it will suffer from the same problems, namely, different allocations for each execution. Our values are the following (but could be different in another execution because of the reasons explained above**:**
    - 1st thread executes: 7, 8, 11
    - 2nd thread executes: 4, 5, 6
    - 3th thread executes: 0, 1, 2, 3, 9, 10

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 no wait directive, the implicit barrier at the end of the for (generated by #pragma omp for) will stop all the threads when they finish the first loop thus making the sequence of prints change from a disordered one to a sequence where the Loop 1 prints go before the Loop 2 ones.

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

No, because the program ends after this loop, so the threads will get to the end of the parallel region not having an effect on the program result.

collapse.c

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

- thread 0 executes: (0 0), (0 1), (0 2), (0 3)
- thread 1 executes: (0 4), (1 0), (1 1)
- thread 2 executes: (1 2), (1 3), (1 4)
- thread 3 executes: (2 0) (2 1) (2 2)
- thread 4 executes: (2 3) (2 4) (3 0)
- thread 5 executes: (3 1) (3 2) (3 3)
- thread 6 executes: (3 4) (4 0) (4 1)
- thread 7 executes: (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?.

No, because the loop variables (i,j) are defined outside the parallel region making them shared by default. That creates a data sharing problem where one iteration can update a false value o get wrong information on those variables, having an effect on the total number of executed iterations (in this case a decrease). We could add the directive private to the for or add the variable definition inside of the parallel region, as displayed below:

```
int i,j;

omp_set_num_threads(8);
#pragma omp parallel for private(i, j)
for (i=0; i < N; i++) {
    for (j=0; j < N; j++) {
        int id=omp_get_thread_num();
        printf("(%d) Iter (%d %d)\n",id,i,j);
    }
}
```

ordered.c

1. Can you explain the order in which printf appear?

In the printf before ordered you cannot predict order because all the threads are running in parallel, however in the printf inside order you can predict order, so all the iterations will be executed in order.

2. How can you ensure that a thread always executes two consecutive iterations in order during the execution of the ordered part of the loop body?

One way to do it is to divide the iterations in chunks of size 2, by adding a 2 to the scheduling clause (schedule(dynamic, 2)). With this change, we can ensure that each thread always executes two consecutive iterations.

doacross.c

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

We can only be sure that the first message will be an “Outside” and the last will be an “Inside” because all the iterations (i) have a dependency to the i - 2, so in order to be able to execute an “Inside” we need an “Outside”. We know that it will finish with an “Inside” for the same reason. Aside from that, we can only say about the order that the “Inside” message of an iteration i, will only be printed after the “Outside” message of the i - 2.

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

The iterations appear in order of the dependencies, for example, first iteration (1 1) do not have a dependence, but iteration (3 3) will be executed only if the iterations (2 3) and (3 2) are executed.

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

Now the threads don’t have to stop to continue with execution of the program, so all messages are more disorders.

C) Tasks

serial.c

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

Yes, because we are expecting a sequential execution due to the lack of pragma omp parallel (and the values of Fibonacci and the thread num is the one expected). No, the code is serial.

parallel.c

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

No, it has some error because it fails in fibonacci. It fails because all threads are doing the same work so the final result is the fibonacci succession multiplied by 4 (there are 4 threads).

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

We could add a single directive on the while to make a single thread create all the tasks (instead of each thread creating their tasks). The while is executed serially but the task keep being parallel. This change can be seen in the following code:

```
#pragma omp parallel firstprivate(p) num_threads(4)
#pragma omp single
   while (p != NULL) {
        #pragma omp task firstprivate(p)
        processwork(p);
        p = p->next;
}
```

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 from the task directive the program will keep working as before, but if the second clause is deleted, we will get a segmentation fault error because of a pointer. They are redundant because if we add a single clause only one thread will be executing the while loop so there won't be any need to not share the variables. (Also because the first first private is only used at the beginning of the loop to avoid a shared p, which is the same but for each iteration in the inner first private).

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

It breaks because at some point, a thread tries to retrieve information from a Null. Let's see an practical example of this situation to understand better out answer: thread 0 reads p and it is not NULL, the thread 1 executes p = p -> next and now next == NULL, so when thread 0 is going to processwork(p), it makes a Segmentation Fault.

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

Because the serial.c is, as its name implies, a serial program so it does not have data sharing problems.

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.

We have four threads, for the sake of an easier explanation we are going to number them 1, 2, 3 and 4 but the numbers are arbitrary. The execution does the following steps:

1. The thread 1 creates the task 1 and 2 and goes to the taskwait until task 1 and 2 are finished.
2. The thread 2 once the task 1 is created executes long sleep for 5 seconds.
3. The thread 3 once the task 2 is created creates the 3th and 4th tasks and leaves the thread open for other tasks [Task 2 finished]
4. One of the free threads at this point (3 or 4, let's suppose it's the 3) starts task 3, a long sleep of ten seconds.
5. Meanwhile the other free thread (4) starts creating the taskloop tasks until it finishes.
6. When the **taskloop tasks are created, the same thread starts processing them**.
7. At some point the thread that was sleeping for 5 second will wake up, finishing its task so **thread 2 is free to do taskloop task.** [Task 1 finished]
8. At the same time, the 1st thread is unlocked because task 1 and 2 are finished, ending its execution and leaving **another free thread that starts doing taskloop tasks.**
9. Some seconds later the thread that was executing task 3 wakes up from the long sleep finishing its task, and the open threads **starts helping with the other taskloop tasks.**

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 magnitude order for the overhead is in microseconds while the overhead per thread gets to a point around 12 cores where it tends to 0.2 microseconds. The general overhead is not constant but due to the tendency of the overhead per thread, has a linear growth.

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.

The magnitude order is tenths of microseconds. It is constant, because the overhead per task tends to 0.11 microseconds. Obviously, more tasks imply a higher total overhead.

3. Based on the results reported by the pi omp taskloop.c code, If you have to generate tasks out of a loop, what seems to be better: to use task or taskloop? Try to reason the answer.

We get the similar results with task or taskloop, but with a smaller number of tasks the task directive works slightly better and with a greater number of tasks, taskloop gives better results. This could happen due to a myriad of reasons, among them: compiler tricks, better optimization of directives, etc. However task and taskloop can be used in most cases equally, as one could "create" a taskloop using tasks and taskgroups.

4. 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 one order of magnitude higher (omp = 0.794204s, omp_critical = 1.789486s, so seconds). The overhead can be decomposed in lock and unlock states (also the transitions in between). The overhead grows the more threads are used, as everyone would expect because more tasks of synchronization, task creation and such parallelization tasks are needed. That happens due to a region being critically blocked, meaning only one thread can have access (which amplifies greatly the synchronization costs).

	nlocked status	Lock	Unlock	Locked status
THREAD 1.1.1	547,902.66 us	15,069.70 us	15,015.34 us	15,651.08 us
THREAD 1.1.2	593,638.78 us	-	-	-
THREAD 1.1.3	593,638.78 us	-	-	-
THREAD 1.1.4	593,638.78 us	-	-	-
THREAD 1.1.5	593,638.78 us	-	-	-
THREAD 1.1.6	593,638.78 us	-	-	-
THREAD 1.1.7	593,638.78 us	-	-	-
THREAD 1.1.8	593,638.78 us	-	-	-

Figure 1. *Paraver trace created using pi_omp_critical.c with 1 threads.*

	Unlocked status	Lock	Unlock	Locked status
THREAD 1.1.1	313,764.78 us	33,573.62 us	4,373.47 us	4,408.04 us
THREAD 1.1.2	314,278.47 us	33,390.46 us	4,374.92 us	4,076.06 us
THREAD 1.1.3	288,859.75 us	60,570.72 us	3,141.07 us	3,548.37 us
THREAD 1.1.4	311,785.45 us	36,203.72 us	4,268.15 us	3,862.59 us
THREAD 1.1.5	286,333.79 us	63,011.42 us	3,014.05 us	3,760.65 us
THREAD 1.1.6	287,053.47 us	62,315.34 us	3,368.70 us	3,382.40 us
THREAD 1.1.7	286,914.93 us	62,113.49 us	3,494.63 us	3,596.86 us
THREAD 1.1.8	289,576.43 us	58,943.15 us	4,035.35 us	3,564.99 us

Figure 2. *Paraver trace created using pi_omp_critical.c with 8 threads.*

5. 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.

Using the script to submit pi_omp_atomic with different parameters and with the visualization provided by the Paraver application we have the date needed to start answering the questions proposed:

- Order of magnitude of atomic overhead: microseconds (5 microseconds in the case of 1 thread and around 7500 microseconds)
- We can observe a considerable growth in the atomic overhead as the number of threads grows. That's because more threads means that each atomic operation will have more cores

waiting without doing useful work (that's why the synchronization values are so different, way higher in the 8 threads trace). If we analyze more specifically our case we see that the memory position affected by our atomic is only sum instead of the whole operation as in the critical case, that's the main reason the numbers from the last exercises are so different and lets us conclude that the atomic clause introduces much less overhead that the critical directive.

```
#pragma omp atomic
 sum += 4.0/(1.0+x*x);
```

	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O
THREAD 1.1.1	223,128,127 ns	-	5,775 ns	21,941 ns	12,620 ns
THREAD 1.1.2	-	223,170,090 ns	-	-	935 ns
THREAD 1.1.3	-	223,170,728 ns	-	-	297 ns
THREAD 1.1.4	-	223,170,808 ns	-	-	217 ns
THREAD 1.1.5	-	223,165,040 ns	-	-	5,985 ns
THREAD 1.1.6	-	223,170,748 ns	-	-	277 ns
THREAD 1.1.7	-	223,170,785 ns	-	-	240 ns
THREAD 1.1.8	-	223,170,778 ns	-	-	247 ns

Figure 3. *Paraver trace created using pi_omp_atomic.c with 1 threads.*

	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O
THREAD 1.1.1	248,365,970 ns	-	1,729,488 ns	5,052,053 ns	12,846 ns
THREAD 1.1.2	5,779,120 ns	230,031,106 ns	80,458 ns	-	12,547 ns
THREAD 1.1.3	7,416,734 ns	228,468,605 ns	5,518 ns	-	8,670 ns
THREAD 1.1.4	7,174,011 ns	228,468,668 ns	248,048 ns	-	6,102 ns
THREAD 1.1.5	7,217,067 ns	228,468,635 ns	205,000 ns	-	7,668 ns
THREAD 1.1.6	7,228,360 ns	228,468,660 ns	193,614 ns	-	6,225 ns
THREAD 1.1.7	4,958,834 ns	228,505,971 ns	2,476,305 ns	-	6,292 ns
THREAD 1.1.8	4,892,394 ns	228,467,235 ns	2,609,035 ns	-	6,077 ns

Figure 4. *Paraver trace created using pi_omp_atomic.c with 8 threads.*

6. 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.

The data retrieval has been done similarly as in the last exercise so it won't be explained again.

Total	450,101,716 ns	10,315,104,537 ns	9,197,081 ns	76,100 ns	2,480 ns
Average	56,262,714.50 ns	448,482,805.96 ns	9,197,081 ns	3,170.83 ns	2,480 ns
Maximum	446,047,042 ns	455,259,126 ns	9,197,081 ns	12,698 ns	2,480 ns
Minimum	451,222 ns	432,917,373 ns	9,197,081 ns	175 ns	2,480 ns
StDev	147,324,636.61 ns	10,243,647.78 ns	0 ns	3,848.26 ns	0 ns
Avg/Max	0.13	0.99	1	0.25	1

Figure 5. *Paraver trace created using pi_omp_sum_vector.c with 8 threads.*

Total	245,319,456 ns	5,714,543,714 ns	9,092,231 ns	77,333 ns	2,427 ns
Average	30,664,932 ns	248,458,422.35 ns	9,092,231 ns	3,222.21 ns	2,427 ns
Maximum	244,151,582 ns	253,259,056 ns	9,092,231 ns	13,006 ns	2,427 ns
Minimum	155,701 ns	237,434,526 ns	9,092,231 ns	190 ns	2,427 ns
StDev	80,690,369.60 ns	7,256,846.23 ns	0 ns	3,897.24 ns	0 ns
Avg/Max	0.13	0.98	1	0.25	1

Figure 6. *Paraver trace created using pi_omp_padding.c with 8 threads.*

We can get the total execution time of sum_vector and padding from figures Figure 5 and Figure 6:

- Execution time sum_vector = 10.51 s (serialized program) [Average of 10 executions].
- Execution time padding = 5.29 s (serialized program) [Average of 10 executions].
- Additional average time for each access to memory = (10.51-5.29)/100008s = 0.000052 s = 52 microseconds.

The increase in the memory access time is caused by de "false sharing". A great description of this effect can be found in the laboratory notes: "*False sharing occurs when multiple threads modify different memory addresses that are stored in the same cache line. When multiple threads update these independent memory locations, the cache coherence protocol forces other threads to update/invalidate their caches.*". We can see that contiguous positions of vector_sum are accessed in different threads, this can create a degree of false sharing and considering that all the vector is accessed this way the overheads is quite relevant.

Padding uses 0 on a the sum_vector (as a matrix) to not let different threads change individual elements on the same cache lines, eliminating the false sharing effect.