Lab 2: Brief tutorial on OpenMP programming model

par4111
Adrià Cabeza, Xavier Lacasa
Departament d' Arquitectura de Computadors

March 16, 2019
2018 - 19 PRIMAVERA

Contents

1 Introduction **3**

2 OpenMP questionnaire **3**

2.1 Parallel regions . . . 3

2.1.1 1.hello.c . . . 3

2.1.2 2.hello.c . . . 3

2.1.3 3.how_many.c . . . 4

2.1.4 4.data_sharing.c . . . 5

2.2 Loop parallelism . . . 6

2.2.1 1.schedule.c . . . 6

2.2.2 2.nowait.c . . . 6

2.2.3 3.collapse.c . . . 6

2.3 Synchronization . . . 6

2.3.1 1.datarace.c . . . 6

2.3.2 2.barrier.c . . . 7

2.3.3 3.ordered.c . . . 7

2.4 Tasks . . . 7

2.4.1 1.single.c . . . 7

2.4.2 2.fibtask.c . . . 7

2.4.3 3.synchtasks.c . . . 7

2.4.4 4.taskloop.c . . . 7

3 Observing overheads **7**

4 Conclusion **7**

1 Introduction

POSAR QUELCOM MÉS ELABORAT AQUÍ

2 OpenMP questionnaire

2.1 Parallel regions

2.1.1 1.hello.c

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

We see the message 24 times. This is due to the *#pragma omp parallel* call before *printf("Hello world! \n");*, which makes every available thread execute the *printf*. In Boada 1, they happen to be 24 threads, and so the message is printed 24 times.

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

By setting the number of threads available to 4, only 4 threads would execute the *printf("Hello world! \n");* line and so the message would be displayed only 4 times. We can accomplish this by adding *num_threads(4)* after *#pragma omp parallel*. Another way would be by using *export OMP_NUM_THREADS=4* before the execution of *1.hola*, so that only 4 threads are available.

2.1.2 2.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). If not, add a data sharing clause to make it correct?

It is not correct, sometimes errors like
(2) Hello (1) Hello (1) world! (1) world!
occur. These happen because the variable *id* is declared before *#pragma omp parallel*, meaning all threads share it. Then what can happen is that for example thread 2 reads the variable *id*, prints the *"(2) Hello"* line, then thread 1 changes the variable, and when it is time to print the *"(2) world!"*, thread 2 prints *"(1) world!"* instead, because thread 1 assigned a new value before thread 2 was done.
To make it correct, we can add the *private(id)* tag to the *#pragma omp parallel num_threads(8)* line, so that every thread has its own local value of the variable *id*. This way, when they assign their own id, they do not change the value other threads are reading from the var *id*.

2. Are the lines always printed in the same order? Why the messages sometimes appear intermixed? (Execute several times in order to see this). No, the lines are not always printed in the same order. This happens because threads do not get tasks assigned by #id order, so whichever thread gets the task earlier in that execution will print earlier.

Messages appear intermixed because even though we added the *private(id)* tag, threads do not wait for other threads to finish their execution before printing their lines (that would be sequential instead of parallel). This way, one thread might print *(id) Hello* and right then another thread's execution might start, printing *(id) Hello* again with a different *id*. After that, they both will finish with *(id) world!* and messages will have been intermixed. To avoid this, we should make the execution of the other threads stop when one thread starts to print until it finishes (we could use the *critical* construct), but that would make the execution sequential (with the added overhead of parallelization, even worse).

2.1.3 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?

20 *Hello world ...* lines are printed on the screen. Each line is printed as many times as the number of threads available for the region of that line.

- Hello world from the first parallel (8)!: 8 times
- Hello world from the second parallel (2)!: 2 times
- Hello world from the second parallel (3)!: 3 times
- Hello world from the third parallel (4)!: 4 times
- Hello world from the fourth parallel (3)!: 3 times

2. What does omp get num threads return when invoked outside and inside a parallel region?

Outside a parallel region it returns the number of threads during the sequential execution, which is always just one.
Inside a parallel region however, it returns the number of threads available (busy or not) for that region:

1. For the first parallel region it returns all threads available for the program which are 8 (because of the *export OMP_NUM_THREADS=8* restriction).
2. The second parallel region is inside a for loop which completes 2 iterations with i = *2 and 3*. Since the number of threads is set to i at each iteration by calling *omp_set_num_threads(i)*, in the first iteration *omp_get_num_threads()* returns 2 and in the second and final one it returns 3.
   Note that *omp_set_num_threads()* overrides the number of threads set by *export OMP_NUM_THREADS=8*, which means that for the rest of the execution, unless otherwise specified, the number of available threads will be 3 (because the last call to *omp_set_num_threads(i)* was made in the second iteration, with $i = 3$.)

3. Before the third parallel region, the *#pragma omp parallel* directive is extended by adding *num_threads(4)*, which overrides the number of available threads just for this entire parallel region to 4. This way *omp_get_num_threads()* returns 4.

4. For this fourth parallel region the number of threads is not overriden, so it takes the value 3 (which is what *omp_get_num_threads()* returns) from the *omp_set_num_threads(3)* call in the for loop.

2.1.4 4.data_sharing.c

1. Which is the value of variable x after the execution of each parallel region with different datasharing attribute (shared, private, firstprivate and reduction)? Is that the value you would expect? (Execute several times if necessary)

1. After first parallel (shared) x's value keeps varying depending on the execution. It should be 120 as intended by the code because there are 16 threads (as set by *omp_set_num_threads(16)*) and $\sum_{i=0}^{15} i = 120$, but its value after the execution is usually between 100 and 120. This variation occurs because all threads are sharing the same variable x, and to execute *x+=omp_get_thread_num();* they must first read the value of x and then write the result of the sum to it. When they do this, however, it can happen that for example thread 3 reads the value 10 in x, then thread 1 also reads the value 10, after that thread 3 updates x with 10+3=13 and thread 1, that has the old value of x, finally updates x with the value it knows: 10+1=11. And so x instead of being 10+3+1=14, is 11 because thread 1 did not notice thread 3's update of x. This happens quite randomly depending on the order of tasks assigned and threads occupation, so the final value of x keeps changing execution after execution.

2. After second parallel (private) x's value is always 5. This is expected as x is assigned the value of 5 right before the second parallel region. Then at the time of declaration of the parallel region, x is declared private (*#pragma omp parallel private(x)*), which means that each thread will modify a local copy (to each thread) of x, but will in no case modify the shared value of the variable (private(x) also makes the local copies of x start without an initialised value instead of the value of x previous to the parallel region (5)). This way threads can't modify the value other threads have of x, but this also means that after the parallel region, unless strictly specified, the shared value of x is still the same as before executing the region, because none of the threads has written the value of their local copy of x to the shared variable x. This is why in the end of the second parallel region x is always equal to 5.

3. After third parallel (firstprivate) x's value is also always 5. This is expected as well, since firstprivate(x) has the same behaviour as private(x) with the

exception that firstprivate(x) initializes each local copy of x for each thread with the value that x had before the start of the parallel region. In this case, that value is 5. Still, this does not affect the value of x after the execution of the parallel region, but it changes the behaviour inside.

4. After fourth parallel (reduction) x's value is always 125. This is correct because *pragma omp parallel reduction(+:x)* is supposed to make local copies of x for each thread, just like private, but initializing them to the neutral element (for addition, this would be 0). Then each thread executes its instructions inside the parallel region and after that all local copies' values are added together with the initial value of x (which was 5). This is therefore equivalent to computing the following (remember there are 16 threads):

$$5 + \sum_{i=0}^{15} i = 125 \tag{1}$$

And 125 is indeed what is printed after each execution for the fourth parallel region, so it is safe to assume it works as intended.

2.2 Loop parallelism

2.2.1 1.schedule.c

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

2.2.2 2.nowait.c

1. Which could be a possible sequence of printf when executing the program? 2. How does the sequence of printf change if the nowait clause is removed from the first for directive? 3. What would happen if dynamic is changed to static in the schedule in both loops? (keeping the nowait clause)

2.2.3 3.collapse.c

1. Which iterations of the loop are executed by each thread when the collapse clause is used? 2. Is the execution correct if the collapse clause is removed? Which clause (different than collapse) should be added to make it correct?

2.3 Synchronization

2.3.1 1.datarace.c

1. Is the program always executing correctly? 2. Add two alternative directives to make it correct. Explain why they make the execution correct.

2.3.2 2.barrier.c

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

2.3.3 3.ordered.c

1.Can you explain the order in which the *Outside* and *Inside* messages are printed? 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?

2.4 Tasks

2.4.1 1.single.c

1. Can you explain why all threads contribute to the execution of instances of the single worksharing construct? Why are those instances appear to be executed in bursts?

2.4.2 2.fibtask.c

1. Why all tasks are created and executed by the same thread? In other words, why the program is not executing in parallel? 2. Modify the code so that the program correctly executes in parallel, returning the same answer that the sequential execution would return.

2.4.3 3.synchtasks.c

1. Draw the task dependence graph that is specified in this program 2. Rewrite the program using only taskwait as task synchronisation mechanism (no depend clauses allowed)

2.4.4 4.taskloop.c

1. Find out how many tasks and how many iterations each task execute when using the grainsize and num tasks clause in a taskloop. You will probably have to execute the program several times in order to have a clear answer to this question. 2. What does occur if the nogroup clause in the first taskloop is uncommented?

3 Observing overheads

4 Conclusion