**Lab 2: OpenMP programming model and analysis of overheads**

Oscar Faixat Llanas

Xavier Algarra Torelló

Grup: PAR1212

PAR 2017Q2

29/03/2017

**PART 1: 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"?**
      1. The message is printed a total of 24 times as the printf that outputs it is in a parallel region and since there a total of 24 cores its printed once for every core.
   2. **Without changing the program, how to make it to print 4 times the "Hello World!" message?**
      1. In order to do this we execute the program overriding the global variable OMP\_NUM\_THREADS: 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.** 
   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?.**
      1. No, in this case as the calls to printf are in a parallel region the order in which they are executed is random and as such we can’t guarantee that they are done in order. To make it print it in order, we could add the pragma omp critical to ensure it only gets executed by a single thread.
   2. **Are the lines always printed in the same order? Could the messages appear intermixed?**
      1. No the lines are not always printed in the same order as the first thread to get to the code region will be the first to execute it. They will no longer appear intermixed as we are using the omp critical pragma.
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?**
      1. 16 lines are printed on the screen.
   2. **If the if(0) clause is commented in the last parallel directive, how many "Hello world ..." lines are printed on the screen?**
      1. The amount of Hello World lines that will be printed is completely random as the number of threads to which it is assigned to is randomized using srand.
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)?**

Doing various executions, we have found that the value of the variable x with the shared attribute is random as it depends on how many data race conditions have occurred. For the private and firstprivate clauses the value obtained is 0 as both have been initialized at 0 and the parallel threads modify a different region of memory than the used by x at that time.

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

In the first directive we need to add the pragma omp atomic right before the x++ statement to ensure that no data races occur by guaranteeing that the access to the x variable is done atomically.

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

Usually we see only 8 messages however on a rare occasion we have seen it prints 12 messages. Due to this, we think there might be a data race condition involving the variables id and i as they are shared between threads.

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

To ensure that the each unique threads executes the appropriate iterations it’s necessary to declare the variables id and i as private.

1. **datarace.c (execute several times before answering the questions)** 
   1. **Is the program always executing correctly?**

Sometimes the program doesn’t execute correctly as there seems to be a data race condition with variable x as it is shared between threads.

* 1. **Add two alternative directives to make it correct. Which are these directives?**

In order to make the execution correct the first option we have thought of is to make use of the pragma omp atomic. This will make sure that each access by the threads to the variable x is done atomically. The second option we have found is to use the reduction attribute on the parallel pragma (reduction (+:x)). The reduction clause allows each thread to calculate its own variable x and then using the operator we have defined combine all the individual values into one. In this specific case we will have each thread calculate its own x variable and then at the end each value obtained from the threads is added to the shared variable that the compiler has created.

1. **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 only predict the part where the threads wake up and enter the barrier. This is due to the fact that the threads waiting period is relative to their id (the bigger the id, the longer they wait). Therefore the first to enter the barrier will be the lowest id and the last the highest id. However we can predict that the first to exit the barrier will be the last one to reach it as it will immediately detect that all others are done and get unlocked.

**B)**

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

In the first loop each thread executes 2 iterations. This happens because the pragma omp for distributes the iterations evenly between the threads. As there are 8 threads and 16 iterations, each thread gets 2 iterations.

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

In the second loop, there are a total of 19 iterations. As this number is not equally divisible by 8, some threads will receive more iterations than others. We have observed that the threads that will obtain the extra iterations are the first 3 threads (threads 0,1,2). Those first will therefore execute 3 iterations while the rest will only execute 2.

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

In order to achieve this, we could use pragma omp single to make sure that the instruction is only executed by the first thread that tries to execute it.

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

In the **first loop** (static scheduling and no parameter) the iterations are distributed equally among all the threads in one chunk made of consecutive iterations. Therefore thread 0 will execute iterations 0-3, thread 1 will execute iterations 4-7 and thread 2 will execute 8-11.

In the **second loop** (static scheduling and parameter 2) the iterations are distributed in chunks of 2 (or the same value as the parameter) iterations. Therefore thread 0 will execute iterations 0,1,6,7, thread 1 will execute 2,3,8,9 and thread 2 4,5,10,11.

As for the **third loop** (dynamic scheduling and parameter 2), the iterations are theoretically distributed in chunks of 2 and without any order. As such we can’t be certain which chunks will be executed first. We can only guarantee that the first thread will execute a chunk of 2 iterations. It can happen where a thread will execute a chunk with only 1 iteration or that the majority of the workload is absorbed by only one thread.

Finally, for **fourth loop** (guided scheduling and parameter 2) the iterations are distributed in order of request and in chunks of 2 iterations. Therefore the first thread to make a request will obtain: num\_iteration/num\_threads with the following requests using the same method. What this means is that the first request will take 12/3= 4 iteration, the second 8/3 = 3 iterations, the third 5/3 = 2 iterations and so forth. However the last two will need to be 2 as that is minimum chunk size and the last one will be 1 as there are no more remaining. As such we can not determine the timeline of execution for each thread.

1. **nowait.c** 
   1. **How does the sequence of printf change if the nowait clause is removed from the first for directive?**

Currently the loops alternate, however if we remove the nowait clause then until the first loop is finished the second loop will not be executed. The nowait clause eliminates the “barrier” that is at the end of a loop.

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

In this program we won’t notice any difference in the output of the program. Since there is no relevant code after the second for, there will be no visible difference in the output generated by the program.

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

The collapse clause indicates the number of nested for loops that are to be processed by the pragma omp parallel for. As we have N=5, we are dealing with 25 iterations which will try to be distributed evenly by thread id among all threads. In our case, as we cannot evenly distribute all of the iterations between threads, the first thread to start execution will make 1 extra iteration. Therefore thread 0 will execute 4 iterations and the other 7 threads will execute 3 iterations.

* 1. **Is the execution correct if the collapse clause is removed? Which clause (different than collapse) should be added to make it correct?.**

If we eliminate the collapse clause the execution would not be correct. That’s because the variable j is shared and therefore there would be a data race condition. To solve this, we should use the clause private(j) to make the variable “j” private and prevent the data race condition.

1. **ordered.c**
   1. **How can you avoid the intermixing of printf messages from the two loops?**

We can solve this by putting the clause ordered in the pragma for declaration that is located before the first loop. We can also put pragma omp ordered before the printf to also organize the output for the first loop.

* 1. **How can you ensure that a thread always executes two consecutive iterations in order during the execution of the first loop?**

To do this we need to use dynamic scheduling and parameter 2, this way we make sure each thread executes chunks of two consecutive iterations.

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

Inside messages are printed once the 2 previous iterations of the outside messages is printed. It works like this because the inside is inside the depend (sink: i-2) block. Therefore, it has to wait until the completion of the iteration i-2, which will be known when that iteration reaches the pragma omp ordered depend(source).

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

The code which is located inside the depend construct is executed when the iterations j-1 and i-1 have finished executing. As before the end of execution is detected when the ordered depend(source) is reached.

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

Printing is performed in such a way that the external loop is spread throughout all of the threads. This in turn doesn’t allow enough time for the internal loop to wait for the previous iteration.

**C) Tasks**

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

Yes, the output is what we expected as the execution is sequential and so there will be no parallelization problems. Therefore, it is not executed in parallel.

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

No, the code is not outputting fibonacci's sequence correctly as it is being multiplied by 4. We are getting this problem because each thread is generating task creations and as such we are getting more tasks than we need.

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

We should simply add the pragma omp single before the creation of tasks, so that we only get the number of tasks we actually need.

* 1. **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 firstprivate clause from the task directive it would not affect the execution of the program as the variable p is used by a private clause and as such it is never modified and always retains the correct value. However, if we removed the firstprivate clause from the parallel directive, the execution would no longer be correct as a data race condition would now occur on p.

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

It is possible that the program returns a segmentation fault error as p becomes a shared variable leading to a data race condition. We would be getting a segmentation fault error because we can have the case where a thread is executing p = p -> next and is then taken over by another thread which also modifies p. At that point p is the equal to NULL. Therefore, when the thread that was taken over tries to access p we would get a segmentation fault error.

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

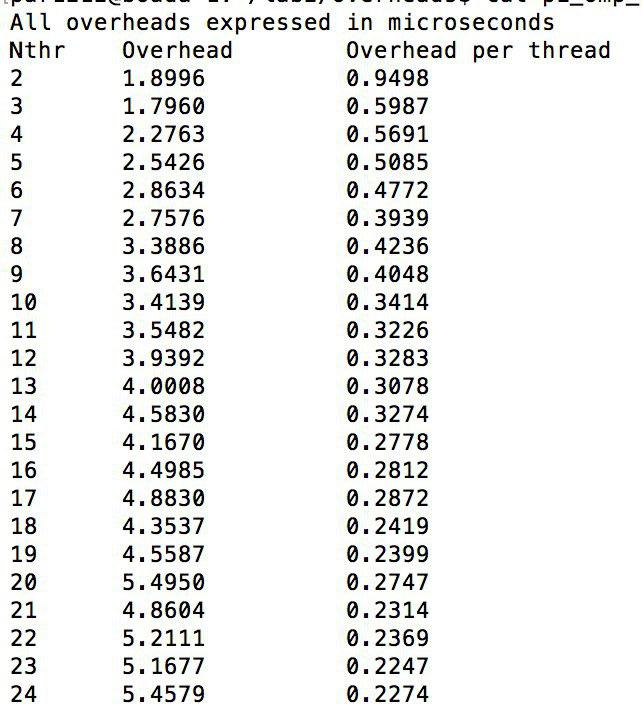
It is not needed because the code is executed sequentially and therefore it would be impossible for a data race to occur as only 1 thread is executing the program.

1. **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.**

A random thread creates T1 and T2 and immediately after enters a taskwait. The 2 following threads then go to sleep for 5 and 10 seconds respectively. The remaining thread then finalizes T3 and T4 and will then start executing the loop body. Immediately after, the 5secs sleeping thread wakes and also starts executing the loop body. Once, the taskwait is over the initial thread will also start executing the loop body.Later, the 10 sec sleep thread wakes up and executes loop body. Finally all of the threads finish executing the iterations that remain of the loop body.

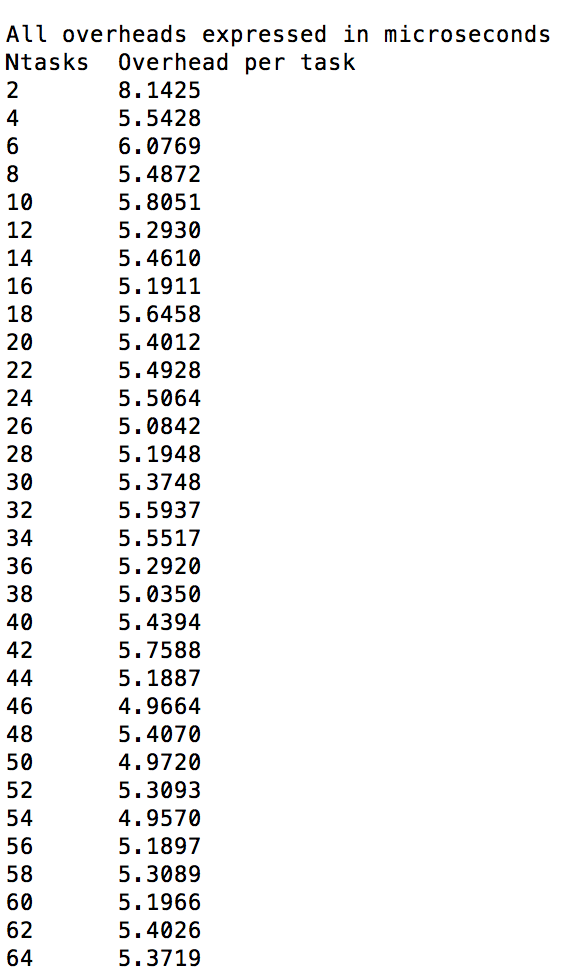
**PART 2: Parallelization overheads**

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 overhead.c code.**



The order of magnitude of the overhead caused by the creation of an individual thread is more or less 10^-7 seconds. The output of the program above is executed with num\_steps=1. We are assuming that the cost of this iteration will tend to zero so most if not all of the execution time consists of the creation and termination of threads. As such we can then calculate the cost of creating and ending each thread by dividing the execution time by the number of threads involved. It is certainly not constant and decreases when the number of threads increase. However the execution time does increase because we have more threads.

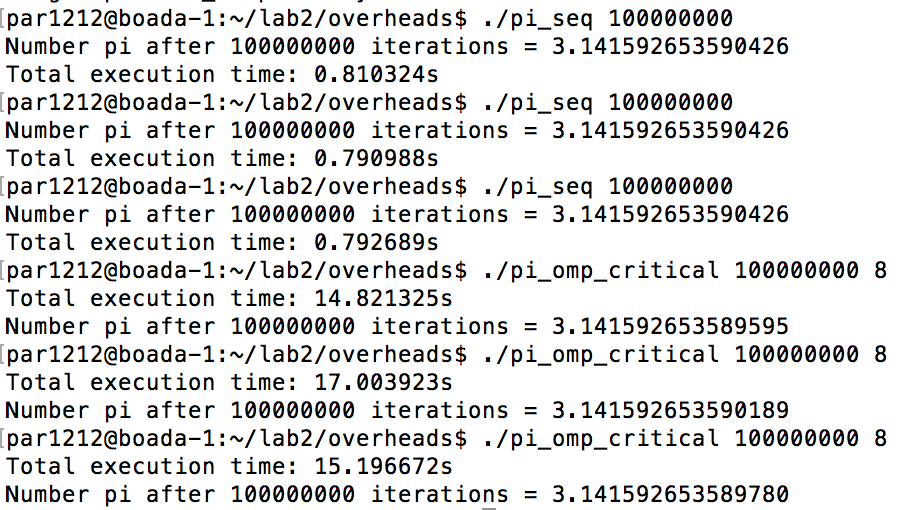
1. **Which is the order of magnitude for the overhead associated with the creation of a task and its synchronization at taskwait in OpenMP ? Is it constant? Reason the answer based on the results reported by the pi omp tasks.c code**



The order of magnitude of the overhead caused by the creation of a task is more or less 10^-6 seconds. The output of the program above is executed with num\_steps=1. We believe that the overhead is somewhat constant seeing as it sways between the 5-6 microseconds mark throughout the majority of cases.

1. **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.**

As we can see the number of critical sections executed in the pi\_omp\_critical program is proportional to num\_steps, because it’s done inside the for loop, whereas in the pi\_omp program it is only proportional to the number of threads, because it is done outside the loop. The order of magnitude of the overhead is of approximately 10^-7 second. This is calculated by executing the program with 100000000 iterations, subtracting the sequential execution time with 100000000 iterations and dividing it by 100000000. In our case, (15.54-0.791)/100000000 ~ 1.5\*10^-7s.



The time of execution for the omp critical is significantly high when compared to the sequential version with num\_steps= 100000000.

The decomposition of the overhead can be done in the lock and unlock operations that have to be necessarily performed to allow the protection of the execution of the line of code. Only one thread can be executed at the same time as it is locked, so the other threads need to wait until it is unlocked. The time is increased when increasing the number of threads as it causes more conflicts. Which increase the time spent in synchronization. Threads are mostly in locked state, waiting for the thread which is currently executing the region that causes conflicts to finish, and reducing the proportion of the unlocked status time.

The number of critical zones executed is in direct relation to the number of iterations that are executed in the code, which end up making the total overhead proportional to it. This is not observed in the pi\_omp.c version because the critical zone is executed exclusively outside of the loop, and is no way proportional to the number of steps.

1. **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.**

To calculate the overhead, we are going to use the same method we used in the previous section. The calculations end up giving us an order of magnitude of 10^-8 seconds.The overhead ends up increasing because of the same reasons as in the previous exercise.

1. **In the presence of false sharing (as it happens in pi omp sumvector.c), which is the additional average memory access time 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 sumvector is used during the execution to store individually for each thread the sum that it ends up calculating. Therefore each thread has its own position saved up in the vector and it doesn’t access any of the other sums.

We calculate the additional memory access time by subtracting the timeof execution of the padding program to the execution time of the original program. We are doing 100000 iterations and using 16 threads. As it can be seen in the screenshots the execution time of the sumvector is 59ms and the execution time of the padding is 13ms, so the time lost doing accesses to memory is approximately of 59 - 13 = 46ms. Which means 0.46 microseconds per iteration.

pi\_omp\_padding:



pi\_omp\_sumvector:



To increase the padding we end up creating a matrix of which we only end up using the first column to save data and a padding (size of cache) is added to the remaining cells of the row. The padding ends up filling the rest of cache line and the next available position maps to the next cache line. Which makes it so that there are no cache conflicts.

1. **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?**

In all executions we have performed 5 executions. We can see that on “critical”, “lock” and “atomic” executions with 8 threads, the execution time is variable as there are some conflicts within the threads exclusion zones and the speedup is only positive in the rest of the programs (sumlocal, sumvector and padding).

