Laboratory Assignment 2

PAR - Q2 2019/2020

Marc Monfort Grau
Albert Mercadé Plasencia
Group par2302

March 4th 2020

Index

Open-MP questionnaire

Parallel regions

1.hello.c

1. ***How many times will you see the "Hello world!" message if the program is executed with* `"./1.hello"`*?***
   We will see as many times as the number of threads executing the code. In our case we see it 24 times "Hello world!”.

2. ***Without changing the program, how to make it to print 4 times the* `"Hello World!"` *message?***
   Executing the program with the environment variable OMP_NUM_THREADS=4. Now the parallel code will be executed only 4 times, because we set the number of threads to be used to 4 instead of 24.

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?***
   The execution is not correct because the variable **id** is shared across the threads, so when a thread prints its own number, it may happen that another thread had change the value of the variable **id**.
   To solve it we have to declare the variable **`id`** as private so it won’t be shared, to do it we add **`private(id)`** to `#pragma omp parallel`.

2. ***Are the lines always printed in the same order? Why the messages sometimes appear inter- mixed? (Execute several times in order to see this).***
   They are not always printed in the same order. After being called (in order), threads take different times to execute and hence the order cannot be predicted.
   The messages appear inter-mixed for the same reason. It can happen that one thread prints "Hello" and before printing "world!" another thread prints its own text.

3.how_many.c

1. ***How many* `"Hello world ..."` *lines are printed on the screen?***
   The **first parallel** region prints **8 `"Hello world..."`**, one for each thread that we assigned with OMP_NUM_THREADS = 8.

The **second parallel** region prints **2** "**`Hello world..`**" in the first iteration, and **3** in the second iteration. That's because it assign the number of threads to be equal to **i** ([2,3]).
The **third parallel** region prints **4** "**`Hello world...`**" because that parallel command its called with the `num_threads(4)` option.
The **fourth parallel** region prints **3** "**`Hello world...`**". That is because we had set `omp_set_num_threads()` to 3 in the last iteration of the `for` loop (second parallel region).

2. ***What does*** `omp_get_num_threads` ***return when invoked outside and inside a parallel region?***
Outside the parallel region it returns 1, as only one thread is working. Inside however, it returns the number of threads that are executing that parallel region.

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`***,*** `firstprivate` ***and*** `reduction`***)? Is that the value you would expect? (Execute several times if necessary)***
After the **first parallel region** (**`shared`**) the value of `x` is not determined. The expected value would be (0+1+2+...+15) = 120 , but in each execution the value can be different because of Data Race. The variable `x` is shared across the threads, so it can happen that while one thread is reading the value of `x`, another thread updates the value of `x`, so the value read by the first thread becomes incorrect, and therefore the result of the final value will be erroneous.
After the **second parallel region** (**`private`**), the variable will keep the value 5 from before entering the parallel region. When we assign `private(x)`, the variable inside the construct is a new variable, the value of `x` from outside that region won't change.
After the **third parallel region** (**`firstprivate`**), the variable, as before, will keep the value 5 form before entering the parallel region. In this case we are also creating a new variable without modifying the value of `x` from outside the region. The only difference is that with `firstprivate` the variable inside the region is initialized with the original value.
After the **fourth parallel region** (**reduction**) the value of `x` will be 5 + (0+1+2+...+15) = 125. That's because we use the option `reduction(+:x)`. So after each thread executing the code with a local variable `x`, all the values of `x` are summed to the original `x`.

Loop parallelism

1.schedule.c

1. ***Which iterations of the loops are executed by each thread for each schedule kind?***
   The first loop uses **`schedule(static)`** and therefore the iterations are assigned in chunks of (**num_iterations / num_threads)** in Round-Robin fashion
   In the second loop, the parallel execution is run with **`schedule(static,2)`**. That means that the iterations will be given to the threads in chunks of two in Round-Robin style.
   Thirdly, the loop will be executed using **`schedule(dynamic,2)`**, which means that blocks of two iterations will be dynamically assigned during the execution. This option helps solve load imbalance problems that may arise using `static` scheduling.
   Finally, we execute the loop with **`schedule(guided,2)`** which is similar to dynamic in that the chunks are assigned dynamically. However, the size of the blocks of iterations is reducing over time depending of the number of iterations left to execute 8 (with a minimum chunk of 2).

2.nowait.c

1. ***Which could be a possible sequence of printf when executing the program?***
   The sequence could be printing 2 messages from the first loop (with 2 different threads, any of them). And at the same time (thank to nowait) 2 more messages from the second loop (with 2 different threads).
   All threads would be different because of the call to sleep. That makes the iteration to be too slow that no thread have time to finish and execute another iteration.

2. ***How does the sequence of*** `printf` ***change if the*** `nowait` ***clause is removed from the first*** `for` ***directive?***
   In this case we will see how the first 2 messages from the first loop will be printed immediately, and after 2 seconds (sleep call), another 2 messages from the second loop will be printed. The second loop won't be executed until the first loop had finished (implicit barrier on #pragma omp parallel).
   Now it's possible that one thread that had executed a parallel part in the first loop can execute another parallel part in the second loop. That's because that thread would already had ended executing the iteration from first loop.

3. ***What would happen if dynamic is changed to static in the schedule in both loops? (keeping the nowait clause)***
   The nowait won't have any effect. That's because the static schedule will assign the divided iterations of the second loop to the first 2 threads, however the first loop

already made the same. So to execute the second loop will be necessary to end the first loop to liberate the first two threads.

3.collapse.c

1. ***Which iterations of the loop are executed by each thread when the* `collapse` *clause is used?***
   When the collapse clause is used, it allows to distribute the work from both loops. So with 8 threads and 25 iterations, each thread will execute 3 sequential iterations, except one (in this case the thread 0) that will execute 4 sequential iterations, for a total for 25 iterations.

2. ***Is the execution correct if the* `collapse` *clause is removed? Which clause (different than* `collapse`*) should be added to make it correct?***
   In this case only the first loop will be parallelized. It has 5 iterations, so each thread form 0 to 4 will execute one of the iterations. The problem is that the variable **j** now is not private and it will cause and error on the execution. To solve it we should assign that variable **j** to be private.

Synchronization

1.datarace.c

1. ***Is the program always executing correctly?***
   No, we have a data race problem because the variable **x** is shared between the threads.

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

```
#pragma omp parallel for schedule(dynamic,1) private(i)
   for (i=0; i < N; i++) {
       #pragma omp atomic
       x++;
   }
```

   In the first alternative we use the atomic synchronization mechanism that avoids the possibility of multiple threads reading and writhing on the variable x at the same time.

```
#pragma omp parallel for schedule(dynamic,1) private(i)
reduction(+:x)
   for (i=0; i < N; i++) {
       x++;
   }
```

Now we use the reduction clause for variable x. That makes all threads to accumulate values into a private variable x, and after all threads had finished its execution, the global variable x is updated with the sum of all local variables from each thread.

2.barrier.c

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

   The first 4 messages wont have any specific order. The 4 next messages will be in order because of the sleep time that have each thread (higher for threads with higher id).

   The last 4 messages wont have any specific order, because before printing the messages all threads will have to wait until all of them had finish all previous parallel executions.

3.ordered.c

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

   Te "Outside" message are executed in parallel and can appear in any order. In contrary, the "Inside" messages will appear in order (as if it was a sequential execution).

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?***

   With schedule(dynamic, 2) we ensure that a thread will execute two consecutive iterations. And that iterations will be in order.

Tasks

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?***

   The *single* construct indicates that only one thread can execute the code. However, because of the *nowait*, the other threads continue the execution and therefore can contribute to its completion. Furthermore, it seems to be executed in bursts as a consequence of the `sleep(1)`, because the task creation and `printf` take way less than one second to execute.

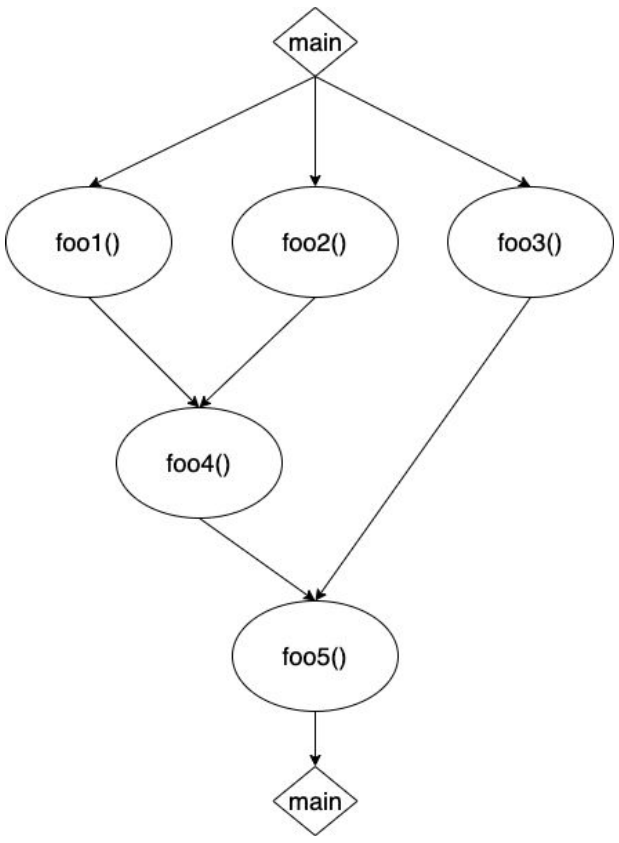
2.fibtasks.c

1. ***Why all tasks are created and executed by the same thread? In other words, why the program is not executing in parallel?***
   Because the code is missing *#pragma omp parallel* in order to activate all the threads and exploit parallelism.

2. ***Modify the code so that the program correctly executes in parallel, returning the same answer that the sequential execution would return.***
   We need to add the following to line 65 before entering the while `loop`:
   ```
   #pragma omp parallel
   #pragma omp single
   ```
   We also need to add `firstprivate(p)` to the line where we declare `#pragma omp task`.
   Finally we need to insert `#pragma omp taskwait` in line 73 between the two `printfs`.

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)***

   The rewritten main of `3.synchtasks.c` looks like this:

```
int main(int argc, char *argv[]) {

    #pragma omp parallel
    #pragma omp single
    {

        printf("Creating task foo3\n");
        #pragma omp task
        foo3();

        printf("Creating task foo1\n");
        #pragma omp task
        foo1();

        printf("Creating task foo2\n");
        #pragma omp task
        foo2();

        #pragma omp taskwait
        printf("Creating task foo4\n");
        #pragma omp task
        foo4();

        #pragma omp taskwait
        printf("Creating task foo5\n");
        #pragma omp task
        foo5();
    }
    return 0;
}
```

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.***

   When executing the loop with `grainsize` each task has a minimum of n iterations assigned to it and at most num_iterations/n tasks.

   However, when using `num_tasks` we have n tasks and each task executes num_iterations/n iterations.

2. ***What does occur if the*** `nogroup` ***clause in the first*** `taskloop` ***is uncommented?***

   What happens is that in the first `taskloop` the implicit `taskgroup` synchronisation is no longer applicable and therefore we don't need to wait for the tasks in the first `taskloop` to finish executing before entering the second `taskloop`.

Observing overheads

Synchronisation overheads

The critical and atomic programs execute one synchronisation operation for each iteration of the for loop. In contrary, the reduction and sumlocal programs only execute one synchronisation operation for each thread.

Version	critical	atomic	reduction	sumlocal
overhead (s)	2,557	0.003	0.041	0.040

In the table above we see the overheads from the different versions of Pi program. The critical version has the higher overhead because of the number of synchronisation operations. The reduction and sumlocal versions has basically the same overhead, as the reduction call is acting like a macro for the sumlocal version
Atomic version have the lowest overhead, even with the same number of synchronisation operations as critical version. That's because atomic clause is acting at very low level reducing the overall time of the synchronisation. And also because we are only using one thread, and there is no major problems with the synchronisation (across threads).

Version	1 thread (sec)	4 thread (sec)	8 thread (sec)
sequential	1.793		
critical	4.351	37.916	33.833
atomic	1.797	6.013	6.777
reduction	1.834	0.478	0.260
sumlocal	1.833	0.4792	0.260

This table show the execution time of the different versions when executing the program with [1, 4, 8] threads, and 100.000.000 iterations.
We see how the critical and atomic version increase the execution time when using more threads. That's because the synchronisation operations are having much more confilcts with more threads, and the benefit of parallelising the code is lost with the synchronisation overhead.
In contrast, the reduction and sumlocal version are having better execution time with more threads. The synchronisation overhead is not affecting that much because in this two cases we

have only one synchronisation operation for each thread (much less than the previous two versions).

Thread creation and termination

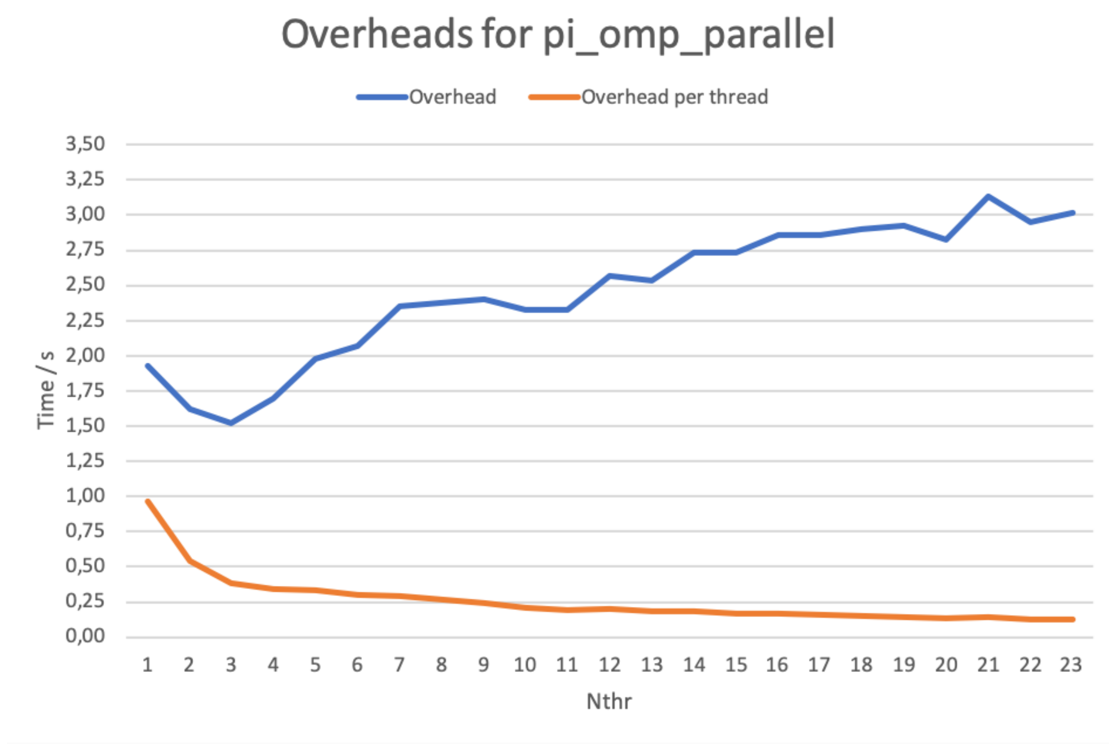


Figure 1. Overheads for pi_omp_parallel

As we see in Figure 1, the overhead incurred as the number of threads (Nthr) used increases shows a clear upwards trend after a small dip at around 3 threads. It is also important to note that the overhead of using just one thread is very large, probably due to reasons that we are no interested in here, and thus the actual cost of increasing the Nthr from 1 to 24 is around one second. If we look at the overhead per thread we see very clearly that it actually decreases as Nthr increases. The overhead per task starts at close to one second and progressively decreases up to around 0.12 seconds for 24 threads. Hence, we conclude that while exploiting parallelism has its costs, in this case if the program is very parallelizable the reduction in execution time won't be greatly offset by the overhead due to the increase in threads.

Task creation and synchronisation

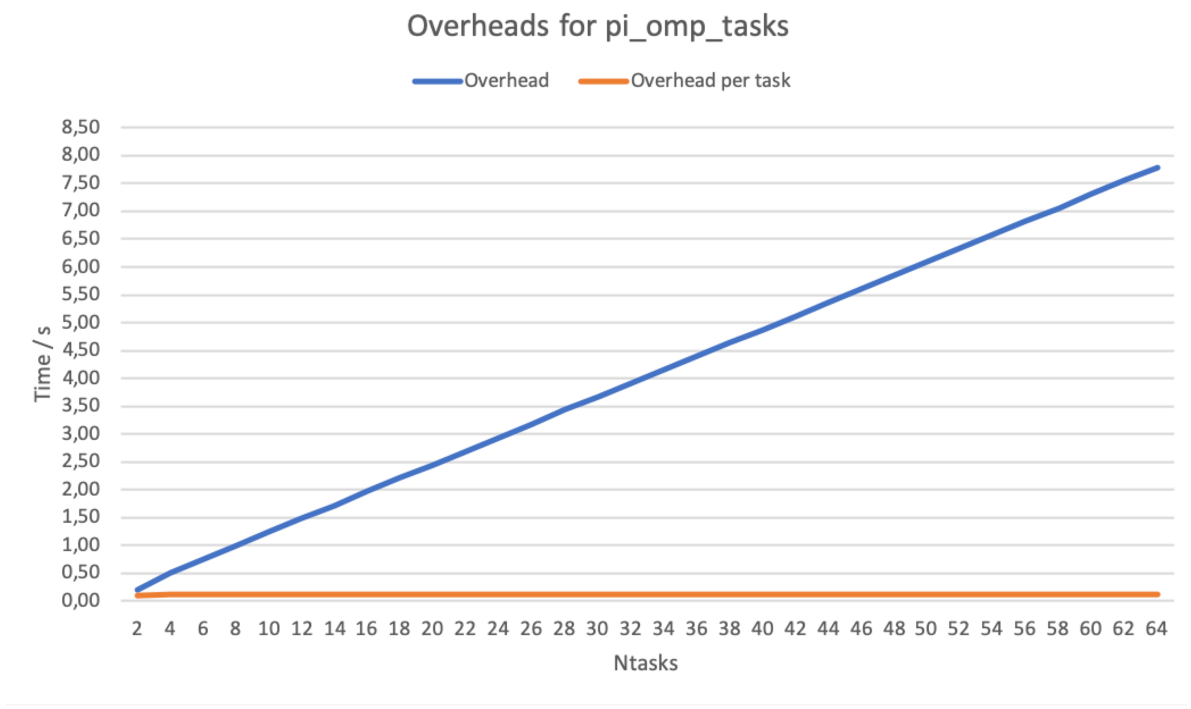


Figure 2. Overheads for pi_omp_tasks

The above Figure 2 resulting from the execution of `pi_omp_tasks.c` conveys to us the evolution of overhead time against the number of tasks (Ntasks). We observe that the overhead increases linearly with Ntasks from almost 0 seconds with 2 tasks to over 7.5 seconds when using 64. As a result, the overhead per task is effectively constant at around 0.12 seconds and we don't see a reduction like we did for the overhead per thread in the previous section.