PAR Laboratory Assignment
Lab 2: Brief tutorial on OpenMP programming model

PAR2102
Alejandro Alarcón Torres
Guillem Reig Gaset

1- Parallel regions

1.hello

1. How many times will you see the "Hello world!" message if the program is executed with "./1.hello"?
24 times, this happens because the parallel construct spreads the execution to all the threads available (in this case, 24).

2. Without changing the program, how to make it to print 4 times the "Hello World!" message?
We assign 4 threads to the execution by typing the command “OMP_NUM_THREADS=4”.

2.hello

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 id’s can be repeated as the variable is public, so any thread can print the id of another thread (the execution is incorrect). If we add the private(id) clause then the execution will be correct.

```
#include <stdio.h>
#include <omp.h>

/* Q1: Is the execution of the program correct? Add a   */
/*     data sharing clause to make it correct           */
/* Q2: Are the lines always printed in the same order?  */
/*     Why the messages sometimes appear intermixed?    */

int main ()
{
    int id;
    #pragma omp parallel num_threads(8) private(id)
    {
        id =omp_get_thread_num();
        printf("(%d) Hello ",id);
        printf("(%d) world!\n",id);
    }
    return 0;
}
```

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).
No, they aren’t. As each thread is performing the task unsynchronized, the order is completely random, it depends on the cpu scheduling.

3.how_many

1. How many"Hello world ..."lines are printed on the screen?
20 times in total, 8 for the pragma omp parallel, because it uses all the available threads (in this case, 8), then 5 for the loop, because it prints “Hello World” 2+3 times (the omp_set_num_threads changes the threads available). Finally, we get 4 “Hello world” due to the pragma omp parallel num_threads(4) and 3 more in the last parallel region, because the last omp_set_num_threads in the loop set the threads available to 3.

2. What does omp get num threads return when invoked outside and inside a parallel region?
When we are inside the parallel region, the omp_get_num_threads() returns the number of threads that are available at that time, whereas if we are outside that region we get 1.

4.data_sharing

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)
1- shared - we get a random number, as there is a data-race in the variable x.
2- private - we get a 5. As each x variable is private for each thread, we never overwrite it’s value.
3- firstprivate - we get a 5. Each thread has its own instance of the variable x, initialized at 5, therefore, it only modifies that instance, and the global non-parallel instance remains unchanged.
4- reduction - we get the right answer, as an instance of the variable is created for each thread, but at the end of the execution all the parallel instances get added, as we would have expected.

2- Loop parallelism

1.schedule

1. Which iterations of the loops are executed by each thread for each schedule kind
1- static: Each thread gets a block of consecutive iterations (in this case, 3 consecutive iterations for each thread).
2- static(2): Each thread gets at least a block of two consecutive iterations.
3- dynamic: The iterations are assigned dynamically as the threads become idle. In each execution the threads can get assigned different iterations.
4- guided: In this case the guided scheduling does the same as the dynamic one, we would see a difference if each iteration had a different workload.

2.nowait

1. Which could be a possible sequence of printf when executing the program?
0 gets iteration 0, 1 gets iteration 1, 2 gets iteration 2, 3 gets iteration 3

2. How does the sequence of printf change if the nowait clause is removed from the first for directive?
Without the nowait we don't get all the printf at the same time (as we did before). This happens because without the nowait clause the first two threads wait to synchronize before executing the second loop.

3. What would happen if dynamic is changed to static in the schedule in both loops? (keeping the nowait clause)
We get the printf separated between the loops, because due to the static scheduling only the threads 0 and 1 get to execute the loops.

3.collapse

1. Which iterations of the loop are executed by each thread when the collapse clause is used?
Each thread does a group of all the iterations of the inner loop. The inner loop is being interpreted as an iteration of the outer loop.

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 j is shared among all threads, we need to use private(j) to get the same effect as the collapse clause.

3- Synchronization

1.datarace

1. Is the program always executing correctly?
No. The program appears to be generating different outputs as there is a data race with the variable "x".

2. Add two alternative directives to make it correct. Explain why they make the execution correct.
The easiest way to solve that would be using the reduction attribute in the pragma clause. That allows each thread to create a variable that will be later summed up with the rest of the threads'

variables. We can also add the atomic clause before the ++x to avoid overwriting x during the execution of another thread.

2.barrier

1. Can you predict the sequence of messages in this program? Do threads exit from the barrier in any specific order?
We cannot predict the sequence of messages that will be outputted as each thread is executing the program separately. However, we know that the thread with the lowest id will get first to the barrier, and the highest id one will be the last to get there. All the threads exit from the barrier at the same time, as the barrier is a tool to keep the threads synchronized in certain instants of the program, but because of this, we can't predict who will output the last message first.

3.ordered

1. Can you explain the order in which the "Outside" and "Inside" messages are printed?
We cannot predict the outside's output, it only depends on which thread gets first to the output. However, we can predict the inside's output, as due to the ordered clause all the outputs are printed sequentially.

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?
We should change the schedule(dynamic) to schedule(dynamic,2).

4- Tasks

1.single

1. Can you explain why all threads contribute to the execution of instances of the single work-sharing construct? Why are those instances appear to be executed in bursts?
A single clause is generated in every iteration of the loop, and all those instances are executed parallely, one thread for each iteration of i.
As we have a sleep(1) at the end of each iteration, every instance has to pause its execution and as all the tasks are similarly weighted they all execute the sleep at the same time, approximately, so we can see a burst of 4 iterations at a time.

2.fibtasks

1. Why all tasks are created and executed by the same thread? In other words, why the program is not executing in parallel?

Because we don't have a #pragma omp parallel clause in the program, therefore the program is executed in a single thread. We need to use the pragma omp parallel to use all the assigned threads.

2. Modify the code so that the program correctly executes in parallel, returning the same answer that the sequential execution would return

We should add a #pragma omp parallel firstprivate(p) clause and a #pragma omp single clause to make the program execute in parallel and make a single thread create all the tasks. If we don't use the firstprivate(p) we get a segmentation fault because p is not initialized.

3.synchtasks

1. Draw the task dependence graph that is specified in this program

foo4 depends of foo1 and foo2
foo5 depends of foo3 and foo4
foo3, foo1 and foo2 don't have dependencies.

2. Rewrite the program using only taskwait as task synchronisation mechanism (no depend clauses allowed)

```
#pragma omp task
{                   //task1
    printf("Creating task foo1\n");
    foo1();
}

#pragma omp task
{                   //task2
    printf("Creating task foo2\n");
    foo2();
}

#pragma omp task
{                   //task3
    printf("Creating task foo3\n");
    foo3();
}
#pragma omp taskwait

#pragma omp task
{                   //task4
    printf("Creating task foo4\n");
    foo5();
}
#pragma omp taskwait

{                   //task5
    printf("Creating task foo5\n");
    foo5();
}
```

This works as a parallel code but foo4 has to wait until foo3 has finished, using the depend clause we can avoid this.

4.taskloop

1. Find out how many tasks and how many iterations each task executed 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.
Using grainsize it distributes 6 iterations to two different threads. Using num_tasks it creates 5 tasks we only have 4 threads available, so one thread does more work than the others.
2. What does occur if the nogroup clause in the first taskloop is uncommented?
The two loop execute simultaneously, with the nogroup clause we execute the loops in order.

5- Observing overheads

1. Synchronization overheads

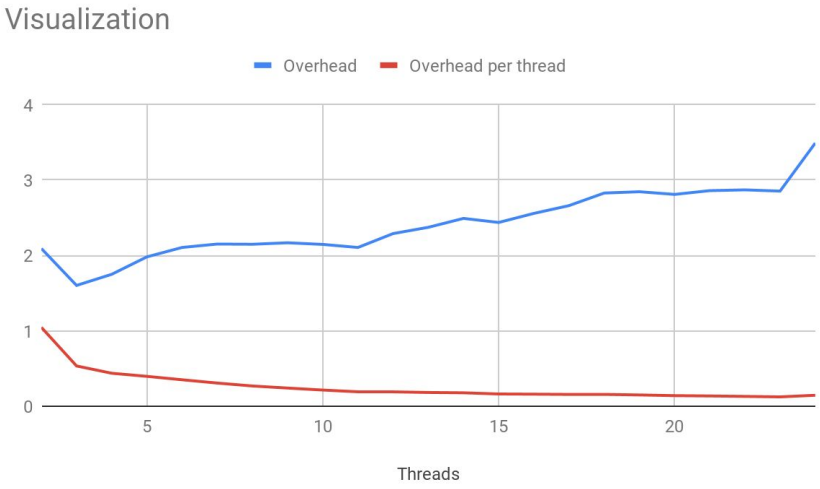
1. If executed with only 1 thread and 100.000.000 iterations, do you observe any major overhead in the execution time caused by the use of the different synchronisation mechanisms? You can compare with the baseline execution time of the sequential version in pi sequential.c.
All the execution times are relatively similar, there are no major variations neither in the execution times nor in the overheads. The overhead benefits and disadvantages tend to appear when we are dealing with more than one thread, so we cannot appreciate major differences given a single-threaded execution.

2. If executed with 4 and 8 threads and the same number of iterations, do the 4 programs benefit from the use of several processors in the same way? Can you guess the reason for this behaviour?
Not at all. The sumlocal and the reduction versions do actually benefit, achieving much lower execution times (up to 2.46 seconds for the reduction with 8 threads, whereas the sequential took about 18 seconds). However, the atomic and critical versions do suffer a severe slowdown, due to the increase in the amount and length of the overheads (62 seconds for the atomic with 4 threads and much more for the critical).

2. Thread creation and termination

1. How does the overhead of creating/terminating threads varies with the number of threads used? Which is the order of magnitude for the overhead of creating/terminating each individual thread in the parallel region?

The overhead per thread decreases and the general overhead increases logarithmically. The approximate coefficient for the increase in 0.6 seconds per thread created. We can see this in the graph below.



3. Thread creation and synchronization

1. How does the overhead of creating/synchronising tasks varies with the number of tasks created? Which is the order of magnitude for the overhead of creating/synchronising each individual task?

We can appreciate a nearly linear relation between these two variables, although the overhead per thread is constant. The approximate coefficient for the increase is 0.06 seconds per task. We can see this in the graph below.