Albert Bernal (2205) i Sergi Bertran (2206) 22-10-2020

Parallelism - Lab 2

3.1 OpenMP questionnaire

Day 1: Parallel regions and implicit tasks

1.hello.c

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

Two, because of the default number of threads.

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

Adding "OMP_NUM_THREADS=4" before the ./1.hello command.

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?

No, it isn't, because sometimes the execution results in a repeated appearance of an "(x) Hello" or an "(x) world!" due to the id variable having been modified by a thread that isn't the one printing the text. We added the "private(id)" option to #pragma omp parallel to solve the problem.

2. Are the lines always printed in the same order? Why do the messages sometimes appear inter-mixed? (Execute several times in order to see this).

No. Because each thread executes two printf and can occur that another thread executes one of its printf during the other thread's execution.

3.howmany.c: Assuming OMP_NUM_THREADS variable is set to 8 with "OMPNUMTHREADS=8 ./3.howmany"

1. What does omp_num_threads return when invoked outside and inside a parallel region?

Outside the parallel region, omp_num_threads returns 1, inside, it returns 8 threads by default (since OMP_NUM_THREADS variable is set to 8*), otherwise the number specified by num_threads(x) or omp_set_num_threads(x) .*

2. Indicate the two alternatives to supersede the number of threads that is specified by OMP_NUM_THREADS environment variable.
As mentioned in the previous question, adding the num_threads(x) option to #pragma omp parallel overrides the environment variable. Also, calling omp_set_num_threads(x) supersedes it.

3. Which is the lifespan for each way of defining the number of threads to be used?
num_threads(x) lasts until the #pragma omp parallel lasts, in this case until the next ";".
omp_set_num_threads(x) lasts indefinitely, until the value is rewritten.

4.datasharing.c

1. Which is the value of variable x after the execution of each parallel region with different data-sharing attributes (shared, private, firstprivate and reduction)? Is that the value you would expect? (Execute several times if necessary)
We can't ensure the value of x because of data race, the value added to x may be modified by other threads.

5.datarace.c

1. Is the program executing correctly? Why?
No, most of the time the output is "Program executed correctly - maxvalue=15 found", but sometimes it is "Sorry, something went wrong - incorrect maxvalue=9 found".
It is not correct because maxvalue is a shared variable and can be modified by other threads.

2. Propose two alternative solutions to make it correct, without changing the structure of the code (just add directives or clauses). Explain why they make the execution correct.
A solution would be using the reduction clause, finding the max value between the max of every thread.
Another way could be using #pragma omp atomic, which would limit to 1 thread at a time the accesses to the shared variable.

3. Write an alternative distribution of iterations to implicit tasks (threads) so that each of them executes only one block of consecutive iterations (i.e. N divided by the number of threads).
#pragma omp taskloop grainsize(N/howmany)

6.datarace.c

1. Is the program executing correctly? Why?

It is not correct because countmax is a shared variable and can be modified by other threads.

2. Propose two alternative solutions to make it correct, without changing the structure of the program (just using directives or clauses) and never making use of critical. Explain why they make the execution correct.
Adding "#pragma omp atomic" before "countmax++" in order to guarantee the indivisible execution of this read-operate-write operation.

7.datarace.c

1. Is the program executing correctly? If not, explain why it is not providing the correct result for one or the two variables (countmax and maxvalue).
No, countmax's value should be 3 and is 9. Count_max's value is being resolved while maxvalue isn't.
Also, countmax's reduction adds its values instead of choosing the max.

2. Write a correct way to synchronise the execution of implicit tasks (threads) for this program.
It can be corrected by adding the critical or atomic clause to the shared variable.

8.barrier.c

1. Can you predict the sequence of printf in this program? Do threads exit from the #pragmaomp barrier construct in any specific order?
We only know that first it will execute the first printf of all threads. Then all threads will execute the second one printf and once it is finished it will execute the last printf.
We don't know any specific order in which threads exit from the #pragmaomp barrier construct.

Day 2: explicit tasks

1.single.c

1. What is the nowait clause doing when associated to single?
It results in the implicit barrier of single to be avoided during execution.

2. Then, can you explain why all threads contribute to the execution of the multiple instances single? Why do those instances appear to be executed in bursts?
Since there is the nowait clause, other threads start executing the next iteration because they are free. There are no repeated instances due to the single, but the work is shared.
They appear to be executed in bursts because of the function "sleep(1)".

2.fibtasks.c

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

It doesn't have the #pragma omp parallel clause in it. There's only a task definition, but no order to execute the tasks in parallel.

2. Modify the code so that tasks are executed in parallel and each iteration of the while loop is executed only once.

We need to add #pragma omp parallel to make the execution parallelizable and #pragma omp single so that not all the threads do the same operations and overwrite the fibonacci value.

3. What is the firstprivate(p) clause doing? Comment it and execute again. What is happening with the execution? Why?

Theoretically, the firstprivate(p) clause declares p to be private to a task, and initializes it with the value that it has when the construct is encountered. In this case though, we can't see a difference in the execution despite repeating it many times.

3.taskloop.c

1. Which iterations of the loops are executed by each thread for each task grainsize or num_tasks specified?

grainsize: a thread executes 0-3, another or the same executes 4-7, and another or the same executes 8-11, since each task defined is of granularity 4 ("size" 4).

num_tasks: same as above but with groups of 3, since now the total amount of iterations is divided by the number of threads.

2. Change the value for grainsize and num_tasks to 5. How many iterations is now each thread executing? How is the number of iterations decided in each case?

grainsize: a minimum of 5 iterations are executed by each thread. Being 12 the number of iterations, the remaining 2 are assigned to a random thread, one that's free.

num_tasks: in this case the division is in the threads, so the size of the tasks is reduced to around 2 or 3 (12/5 = 2.4).

3. Can grainsize and num_tasks be used at the same time in the same loop?

It can't be used, since it would be dividing the tasks in 2 different ways. Technically it could be correct if, for example, in this case we used grainsize(3) and num_tasks(4) at the same time, since both obtain the same result in task size, but the compiler gives out an error when the clauses are used in the same segment so it is not possible.

4. What is happening with the execution of tasks if the nogroup clause is uncommented in the first loop? Why?

The task division is the same, but the execution of the threads is parallel and there is no waiting for the other tasks to end. The output gets mixed up but is correct.

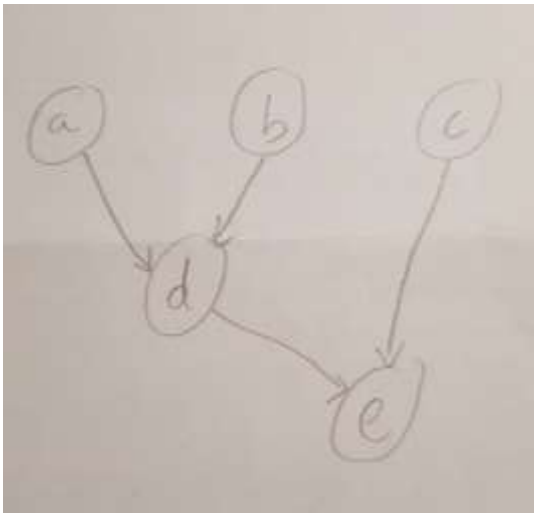
4.reduction.c

1. Complete the parallelisation of the program so that the correct value for variable sum returned in each print statement. Note: in each part of the 3 parts of the program, all tasks generated should potentially execute in parallel.

```
#define SIZE 8192
#define BS   16
int X[SIZE], sum;

int main()
{
    int i;

    for (i=0; i<SIZE; i++)
    X[i] = i;

    omp_set_num_threads(4);
    #pragma omp parallel
    #pragma omp single
    {
        // Part I
    #pragma omp taskgroup task_reduction(+: sum)
        {
        for (i=0; i< SIZE; i++)
        #pragma omp task firstprivate(i) in_reduction(+: sum)
            sum += X[i];
    }

        printf("Value of sum after reduction in tasks = %d\n", sum);

        // Part II
    #pragma omp taskloop grainsize(BS) reduction(+: sum)
    for (i=0; i< SIZE; i++)
        sum += X[i];

        printf("Value of sum after reduction in taskloop = %d\n", sum);

        // Part III
    #pragma omp taskgroup task_reduction(+: sum)
    {
    for (i=0; i< SIZE/2; i++)
        #pragma omp task firstprivate(i) in_reduction(+: sum)
        sum += X[i];
    }
    #pragma omp taskloop grainsize(BS) reduction(+: sum)
    for (i=SIZE/2; i< SIZE; i++)
        sum += X[i];

        printf("Value of sum after reduction in combined task and taskloop = %d\n", sum);
    }

    return 0;
}
```

5.synchtasks.c

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



Where:

- *a = foo1*
- *b = foo2*
- *c = foo3*
- *d = foo4*
- *e = foo5*

2. Rewrite the program using only taskwait as task synchronisation mechanism (no depend clauses allowed), trying to achieve the same potential parallelism that was obtained when using depend.

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

    #pragma omp parallel
    #pragma omp single
    {
    printf("Creating task foo1\n");
    #pragma omp task
    foo1();
    printf("Creating task foo2\n");
    #pragma omp task
    foo2();
    printf("Creating task foo3\n");
    #pragma omp task
    foo3();
    printf("Creating task foo4\n");
    #pragma omp taskwait
    foo4();
    printf("Creating task foo5\n");
    #pragma omp taskwait
    foo5();
    }
    return 0;
}
```

We tried to achieve the same dependence graph but the best we could come up with was the same graph as above but with foo4 depending on foo3.

3. Rewrite the program using only taskgroup as task synchronisation mechanism (no depend clauses allowed), again trying to achieve the same potential parallelism that was obtained when using depend.

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

    #pragma omp parallel
    #pragma omp single
    {
    printf("Creating task foo1\n");
    #pragma omp taskgroup
    {
    foo1();
    printf("Creating task foo2\n");
    #pragma omp task
    foo2();
    printf("Creating task foo3\n");
    #pragma omp task
    foo3();
    }
    printf("Creating task foo4\n");
    #pragma omp taskgroup
    foo4();
    printf("Creating task foo5\n");
    #pragma omp task
    foo5();
    }
    return 0;
}
```

Again, the same dependence graph as the previous without using depends.

3.2 Observing overheads

Thread creation and termination

For this task, we ran the execution without sbatch because boada had too many requests and after waiting for minutes for the output it didn't respond. Executing by using "./pi_omp_parallel" we obtained the following output:

```
par2205@boada-1:~/lab2/overheads$ ./pi_omp_parallel 1 24
All overheads expressed in microseconds
Nthr    Overhead        Overhead per thread
2       1.0423          0.5212
3       4.6695          1.5565
4       5.0834          1.2709
5       8.7476          1.7495
6       10.8357         1.8059
7       15.4255         2.2036
8       16.0759         2.0095
9       18.7780         2.0864
10      20.5262         2.0526
11      24.0042         2.1822
12      25.8717         2.1560
13      28.7492         2.2115
14      31.2154         2.2297
15      34.9310         2.3287
16      36.9639         2.3102
17      41.0808         2.4165
18      43.7793         2.4322
19      47.1860         2.4835
20      50.6119         2.5306
21      53.7529         2.5597
22      67.0339         3.0470
23      70.7304         3.0752
24      77.2063         3.2169
```

It shows that the order of magnitude for the overhead of creating/terminating each individual thread in the parallel region is around 2.3 (in other executions the results were similar).

After around 15 minutes the .txt files were generated with the new results. We think leaving the previous data in the deliverable since it is interesting seeing both executions side by side. This is the new output obtained:

```
All overheads expressed in microseconds
Nthr    Overhead    Overhead per thread
2    2.0359    »    1.0179
3    1.5024    »    0.5008
4    1.7023    »    0.4256
5    1.8437    »    0.3687
6    1.9600    »    0.3267
7    2.0146    »    0.2878
8    2.2474    »    0.2809
9    2.2245    »    0.2472
10   2.3851    »    0.2385
11   2.3639    »    0.2149
12   2.4938    »    0.2078
13   2.7892    »    0.2146
14   3.1034    »    0.2217
15   2.8184    »    0.1879
16   3.4691    »    0.2168
17   3.0205    »    0.1777
18   3.5469    »    0.1971
19   3.0156    »    0.1587
20   3.7221    »    0.1861
21   3.1359    »    0.1493
22   4.0558    »    0.1844
23   3.1978    »    0.1390
24   4.1360    »    0.1723
```

This time, the results allow us to be vastly more optimistic, since, even though overhead increases, overhead per thread is decreasing slowly. From this, we conclude that the amount of overhead added in from thread addition is a bit less than the speedup that we get because of it.

The order of magnitude seems to be around 0.2 for this execution.

Task creation and synchronisation

For this task we will try with the sbatch command again and wait for the correct output. These are the results:

```
All overheads expressed in microseconds
Ntasks  Overhead    Overhead per task
2   0.2473      0.1236
4   0.4385      0.1096
6   0.6047      0.1008
8   0.7852      0.0982
10  1.0563      0.1056
12  1.2254      0.1021
14  1.5562      0.1112
16  1.6048      0.1003
18  1.8083      0.1005
20  2.1097      0.1055
22  2.2445      0.1020
24  2.4318      0.1013
26  2.6179      0.1007
28  2.8238      0.1008
30  3.0136      0.1005
32  3.2412      0.1013
34  3.6879      0.1085
36  3.9498      0.1097
38  3.8670      0.1018
40  4.1427      0.1036
42  4.3017      0.1024
44  4.5661      0.1038
46  4.7055      0.1023
48  4.9842      0.1038
50  5.1092      0.1022
52  5.4553      0.1049
54  5.7486      0.1065
56  5.7649      0.1029
58  6.1897      0.1067
60  6.3434      0.1057
62  6.3411      0.1023
64  6.5552      0.1024
```

This execution shows a somewhat constant overhead per task, with all the values except for three in the 0.10 range, and the variation is, in our opinion, due to the program being executed in a non-deterministic machine. This means that the amount of time of creating a task is independent of the amount of tasks.