PAR Laboratory Deliverable

Course 2019/20 (Fall semester)

Lab 2: Brief tutorial on OpenMP programming model

Rafel-Albert Bros Esqueu, Joan Vinyals Ylla-Català

User: Par2201

October, 2019

Contents

1 OpenMP questionnaire

1.1 Parallel regions

1.1.1 1.hello.c

- How many times will you see the *Hello world!* message if the program is executed with *./1.hello*?
  The message is printed 24 times.

- Without changing the program, how to make it to print 4 times the *Hello world!* message?
  We set the environment variable `OMP_NUM_THREADS` to 4.

1.1.2 2.hello.c

- 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 is not. In order to correct it we made the variable `id` private.

- Are the lines always printed in the same order?
  No, they are not.
  Why the messages sometimes appear inter-mixed?
  That is because the thread scheduling is done by the operating system following its own policies.

1.1.3 3.how_many.c

Assuming the `OMP_NUM_THREADS` variable is set to 8 with `export OMP_NUM_THREADS=8`.

- How many *Hello world ...* lines are printed on the screen?
  There are 20 lines printed.

- What does `omp_get_num_threads` return when invoked outside and inside a parallel region?
  It returns the number of threads inside its own parallel region. When it is outside of any parallel region it returns number 1.

1.1.4 4.data_sharing.c

Which is the value of variable `x` after the execution of each parallel region with different data-sharing attribute (`shared, private, firstprivate` and `reduction`)?

At the Table 1, we can see the values of x after the execution of 4.datasharing.c, in the first case for the `shared` clause x took 112 as value but this was a random outcome since all threads read and write the same value at once. For the `private` and `firstprivate` we got the same value because inside the

parallel region all the threads are writing at a local copy they have so that the final value is the same as outside the parallel region. Finally, the `reduction` construct using the given function (in this case +) and the target variable (in this case x), it uses the function to reduct all the local values in each thread into the global one.

Schedule type	**Value of x**
`shared`	112
`private`	5
`firstprivate`	5
`reduction`	125

Table 1: Values of `x` per every scheduling type.

1.2 Loop parallelism

1.2.1 1.shcedule.c

- Which iterations are executed by each thread for each `schedule` kind?

```
Going to distribute 12 iterations with schedule(static) ...
Loop 1: (0) gets iteration 0
Loop 1: (0) gets iteration 1
Loop 1: (0) gets iteration 2
Loop 1: (1) gets iteration 3
Loop 1: (1) gets iteration 4
Loop 1: (1) gets iteration 5
Loop 1: (2) gets iteration 6
Loop 1: (2) gets iteration 7
Loop 1: (2) gets iteration 8
Loop 1: (3) gets iteration 9
Loop 1: (3) gets iteration 10
Loop 1: (3) gets iteration 11
Going to distribute 12 iterations with schedule(static, 2) ...
Loop 2: (3) gets iteration 6
Loop 2: (3) gets iteration 7
Loop 2: (0) gets iteration 0
Loop 2: (0) gets iteration 1
Loop 2: (0) gets iteration 8
Loop 2: (0) gets iteration 9
Loop 2: (2) gets iteration 4
Loop 2: (2) gets iteration 5
Loop 2: (1) gets iteration 2
Loop 2: (1) gets iteration 3
Loop 2: (1) gets iteration 10
Loop 2: (1) gets iteration 11
Going to distribute 12 iterations with schedule(dynamic, 2) ...
Loop 3: (0) gets iteration 0
Loop 3: (2) gets iteration 4
Loop 3: (2) gets iteration 5
Loop 3: (1) gets iteration 6
Loop 3: (2) gets iteration 8
```

```
Loop 3: (3) gets iteration 2
Loop 3: (1) gets iteration 7
Loop 3: (0) gets iteration 1
Loop 3: (3) gets iteration 3
Loop 3: (2) gets iteration 9
Loop 3: (1) gets iteration 10
Loop 3: (1) gets iteration 11
Going to distribute 12 iterations with schedule(guided, 2) ...
Loop 4: (3) gets iteration 0
Loop 4: (0) gets iteration 2
Loop 4: (0) gets iteration 3
Loop 4: (0) gets iteration 8
Loop 4: (0) gets iteration 9
Loop 4: (0) gets iteration 10
Loop 4: (0) gets iteration 11
Loop 4: (1) gets iteration 4
Loop 4: (1) gets iteration 5
Loop 4: (3) gets iteration 1
Loop 4: (2) gets iteration 6
Loop 4: (2) gets iteration 7
```

1.2.2 2.nowait.c

- Which could be a possible sequence of `printf` when executing the program?

```
par2201@boada-1:worksharing$ ./2.nowait
Loop 1: thread (0) gets iteration 0
Loop 1: thread (2) gets iteration 1
Loop 2: thread (1) gets iteration 2
Loop 2: thread (3) gets iteration 3
par2201@boada-1:worksharing$ ./2.nowait
Loop 1: thread (0) gets iteration 0
Loop 1: thread (1) gets iteration 1
Loop 2: thread (2) gets iteration 2
Loop 2: thread (3) gets iteration 3
```

- How does the sequence of `printf` change if the `nowait` clause is removed from the first `for` directive?

```
par2201@boada-1:worksharing$ ./2.nowait
Loop 1: thread (0) gets iteration 0
Loop 1: thread (1) gets iteration 1
Loop 2: thread (1) gets iteration 2
Loop 2: thread (0) gets iteration 3
par2201@boada-1:worksharing$ ./2.nowait
Loop 1: thread (0) gets iteration 0
Loop 1: thread (1) gets iteration 1
Loop 2: thread (0) gets iteration 2
Loop 2: thread (1) gets iteration 3
```

- What would happen if dynamic is changed to static in the schedule in both loops (keeping the `nowait` clause)?

```
par2201@boada-1:worksharing$ ./2.nowait
Loop 1: thread (0) gets iteration 0
Loop 1: thread (1) gets iteration 1
Loop 2: thread (0) gets iteration 2
Loop 2: thread (1) gets iteration 3
par2201@boada-1:worksharing$ ./2.nowait
Loop 1: thread (0) gets iteration 0
Loop 1: thread (1) gets iteration 1
Loop 2: thread (0) gets iteration 2
Loop 2: thread (1) gets iteration 3
```

1.2.3 3.collapse.c

- Which iterations of the loop are executed by each thread when the `collapse` clause is used?

```
par2201@boada-1:worksharing$ ./3.collapse
(0) Iter (0 0)
(0) Iter (0 1)
(0) Iter (0 2)
(0) Iter (0 3)
(2) Iter (1 2)
(2) Iter (1 3)
(2) Iter (1 4)
(3) Iter (2 0)
(3) Iter (2 1)
(7) Iter (4 2)
(7) Iter (4 3)
(7) Iter (4 4)
(5) Iter (3 1)
(5) Iter (3 2)
(5) Iter (3 3)
(4) Iter (2 3)
(4) Iter (2 4)
(4) Iter (3 0)
(3) Iter (2 2)
(1) Iter (0 4)
(1) Iter (1 0)
(1) Iter (1 1)
(6) Iter (3 4)
(6) Iter (4 0)
(6) Iter (4 1)
par2201@boada-1:worksharing$ ./3.collapse
(0) Iter (0 0)
(0) Iter (0 1)
(0) Iter (0 2)
(0) Iter (0 3)
(3) Iter (2 0)
(1) Iter (0 4)
(1) Iter (1 0)
(1) Iter (1 1)
(2) Iter (1 2)
(5) Iter (3 1)
(5) Iter (3 2)
(5) Iter (3 3)
(4) Iter (2 3)
```

```
(4) Iter (2 4)
(4) Iter (3 0)
(3) Iter (2 1)
(3) Iter (2 2)
(2) Iter (1 3)
(2) Iter (1 4)
(6) Iter (3 4)
(6) Iter (4 0)
(6) Iter (4 1)
(7) Iter (4 2)
(7) Iter (4 3)
(7) Iter (4 4)
```

- Is the execution correct if the `collapse` clause is removed? Which clause (different than `collapse`) should be added to make it correct?
  Without the `collapse` clause the execution is not correct. With the clause `private(j)` the program executes as expected.

```
par2201@boada-1:worksharing$ ./3.collapse
(0) Iter (0 0)
(0) Iter (0 1)
(0) Iter (0 2)
(0) Iter (0 3)
(2) Iter (1 2)
(2) Iter (1 3)
(2) Iter (1 4)
(3) Iter (2 0)
(3) Iter (2 1)
(7) Iter (4 2)
(7) Iter (4 3)
(7) Iter (4 4)
(5) Iter (3 1)
(5) Iter (3 2)
(5) Iter (3 3)
(4) Iter (2 3)
(4) Iter (2 4)
(4) Iter (3 0)
(3) Iter (2 2)
(1) Iter (0 4)
(1) Iter (1 0)
(1) Iter (1 1)
(6) Iter (3 4)
(6) Iter (4 0)
(6) Iter (4 1)
par2201@boada-1:worksharing$ ./3.collapse
(0) Iter (0 0)
(0) Iter (0 1)
(0) Iter (0 2)
(0) Iter (0 3)
(3) Iter (2 0)
(1) Iter (0 4)
(1) Iter (1 0)
(1) Iter (1 1)
(2) Iter (1 2)
(5) Iter (3 1)
(5) Iter (3 2)
(5) Iter (3 3)
```

```
(4) Iter (2 3)
(4) Iter (2 4)
(4) Iter (3 0)
(3) Iter (2 1)
(3) Iter (2 2)
(2) Iter (1 3)
(2) Iter (1 4)
(6) Iter (3 4)
(6) Iter (4 0)
(6) Iter (4 1)
(7) Iter (4 2)
(7) Iter (4 3)
(7) Iter (4 4)
```

1.3 Synchronization

1.3.1 1.datarace.c

- Is the program always executing correctly?
  It does not. All the observations were incorrect. More than a thousand samples were taken.
- Add two alternative directives to make it correct. Explain why they make the execution correct.
  On the one hand we can use `atomic`. On the other hand we can use `critical`. Alternativeley we can use `reduction(+:x)`.

1.3.2 2.barrier.c

- Can you predict the sequence of messages in this program?
  We can only predict the wake up messages (*(`Thid`) wakes up and enters barrier ...*) ,because the sleeping time depends on the `Thid`.
  Do threads exit from the `barrier` in any specific order?
  No, there is no evidencie of such beavoiur.

```
par2201@boada-1:synchronization$ ./2.barrier
(0) going to sleep for 2000 milliseconds ...
(1) going to sleep for 5000 milliseconds ...
(2) going to sleep for 8000 milliseconds ...
(3) going to sleep for 11000 milliseconds ...
(0) wakes up and enters barrier ...
(1) wakes up and enters barrier ...
(2) wakes up and enters barrier ...
(3) wakes up and enters barrier ...
(0) We are all awake!
(3) We are all awake!
(2) We are all awake!
(1) We are all awake!
```

1.3.3 3.ordered.c

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

The message *Outside* is printed randomly, while the message *Inside* depends of the value of `i`.

- How can you ensure that a thread always executes two consecutives iterations in order during the execution of the ordered part of the loop body?
  Increasing the chunck size of the dynamic scheduling type to two.

```
par2201@boada-1:synchronization$ ./3.ordered
Before ordered - (0) gets iteration 0
Inside ordered - (0) gets iteration 0
Before ordered - (0) gets iteration 1
Inside ordered - (0) gets iteration 1
Before ordered - (0) gets iteration 2
Inside ordered - (0) gets iteration 2
Before ordered - (0) gets iteration 3
Inside ordered - (0) gets iteration 3
Before ordered - (0) gets iteration 4
Inside ordered - (0) gets iteration 4
Before ordered - (2) gets iteration 6
Before ordered - (3) gets iteration 5
```

1.4 Tasks

1.4.1 1.single.c

- Can you explain why all threads contribute to the execution of instances of the `single` work-sharing construct?
  Because the clause `nowait` allows the iteration to end so the next iteration can be executed and assigned to a free thread.
  Why are those instances to appear to be executed in bursts?
  They appear to be executed in bursts because the number of threads is set to 4, and the call to sleep is blocking the thread.

1.4.2 2.fibtasks.c

- Why all tasks are created and executed by the same thread? In other words, why the program is not executing in parallel?
  Because the `pragma omp parallel` is not used so the threads are not created.

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

```
#pragma omp parallel
#pragma omp single
while (p != NULL) {
   ...
  #pragma omp task firstprivate(p)
   processwork(p);
   ...
}
```

1.4.3 3.synchtasks.c

- Draw the task dependence graph that is specified in this program. We used the `depend` clauses to determine if a task has a dependency with another one, and the outcome was the TDG (Task Dependency Graph) in Figure 1

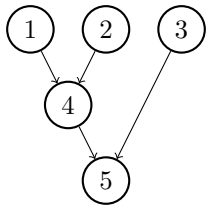

Figure 1: TDG for the program *3.synctasks.c* (the task name is X for foo$X$)

- Rewrite the program using only *taskwait* as task synchronisation mechanism (no `depend` clauses allowed). We used the `taskwait` clause before any class that has a dependency with a previous task.

```
#pragma omp parallel
#pragma omp single
{
    ... task foo[1-3] ...
    #pragma omp taskwait
    ... task foo4 ...
    #pragma omp taskwait
    ... task foo5 ...
}
```

1.4.4 4.taskloop.c

- 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.
  On the one hand, if we use the clause `grainsize`, considering that we have N iterations, $N/grainsize$ tasks are created, and each task executes $grainsize$ iterations. On the other hand, when using the clause `num_tasks`, the number of tasks is obviously num_tasks and as the number of iterations is distributed evenly its value develops from the expression N/num_tasks.

- What does occur if the `nogroup` clause in the first `taskloop` is uncommented?
  The `nogroup` clause is used to remove the implicit `taskgroup` in the `taskloop`, so when it is uncommented the tasks of the second loop start

creating before the ending of the first loop. This is useful if there are no dependencies across loops.

2 Observing overheads

Concerning overheads the main OpenMP functionalities that cause an overhead are both thread creation and task creation. On the one hand, the `parallel` construct, which creates the threads, has a nearly constant overhead regardless of the number of threads created. On the other hand, the cost on task creation is constant for the task, so the overhead increases linearly with the number of tasks created.

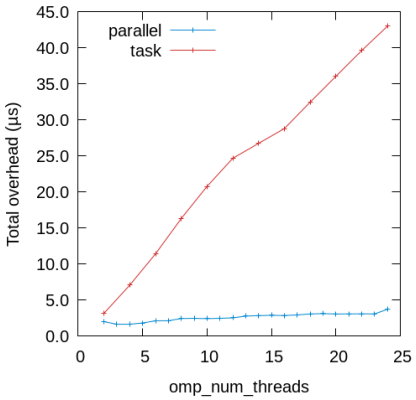

Figure 2: Overhead of invoking the `parallel` construct

However, there are some well known techniques to parallelize the task creation unlike with thread creation.