Lab 2: Brief tutorial on OpenMP programming model

*2018-2019 Q2*

**par2110**

Maurici Abad Gutierrez

Nicolas Schrack

# OpenMP questionnaire

## A) Parallel regions

### 1. hello.c

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

24 times, because 24 is the default number of threads in the machine.

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

Run the program width the option “OMP\_NUM\_THREADS=4 ./1.hello” and set the thread number to 4.

### 2. hello.c

#### a. 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.

#pragma omp parallel num\_threads(8) private(id)

#### b. Are the lines always printed in the same order? Why the messages sometimes appear intermixed? (Execute several times in order to see this).

No. Because they are executed in parallel and in each execution, the operating system gives different execution time at each thread.

### 3. how\_many.c

#### a. How many "Hello world ..." lines are printed on the screen?

8 + 2 + 3 + 4 + 3 = 20

#### b. What does omp get num threads return when invoked outside and inside a parallel region?

Outside shows 1 and inside shows the thread number (like the process id).

### 4. data\_sharing.c

#### a. Which is the value of variable x after the execution of each parallel region with different datasharing attribute (shared, private, firstprivate and reduction)? Is that the value you would expect? (Execute several times if necessary)

The shared is most of the times 120, but due to data overriding sometimes changes because the variable x is shared between threads and not synchronized.

The private is always 5 because each thread has a copy of the variable x initialized to 0, so the original is not changing.

The firstprivate is always 5 because each thread has a copy of the variable x, so the original is not changing.

The reduction is always 125 because there is no data overriding and the variable x is updated at the end.

## 

## B) Loop parallelism

### 1. schedule.c

#### a. Which iterations of the loops 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: (0) gets iteration 0

Loop 2: (0) gets iteration 1

Loop 2: (0) gets iteration 8

Loop 2: (0) gets iteration 9

Loop 2: (1) gets iteration 2

Loop 2: (1) gets iteration 3

Loop 2: (1) gets iteration 10

Loop 2: (1) gets iteration 11

Loop 2: (2) gets iteration 4

Loop 2: (2) gets iteration 5

Loop 2: (3) gets iteration 6

Loop 2: (3) gets iteration 7

Going to distribute 12 iterations with schedule(dynamic, 2) ...

Loop 3: (0) gets iteration 6

Loop 3: (0) gets iteration 7

Loop 3: (1) gets iteration 2

Loop 3: (1) gets iteration 3

Loop 3: (2) gets iteration 0

Loop 3: (2) gets iteration 1

Loop 3: (2) gets iteration 8

Loop 3: (2) gets iteration 9

Loop 3: (2) gets iteration 10

Loop 3: (2) gets iteration 11

Loop 3: (3) gets iteration 4

Loop 3: (3) gets iteration 5

Going to distribute 12 iterations with schedule(guided, 2) ...

Loop 4: (0) gets iteration 0

Loop 4: (0) gets iteration 1

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 6

Loop 4: (1) gets iteration 7

Loop 4: (2) gets iteration 2

Loop 4: (2) gets iteration 3

Loop 4: (3) gets iteration 4

Loop 4: (3) gets iteration 5

### 2. nowait.c

#### a. Which could be a possible sequence of printf when executing the program?

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

#### b. How does the sequence of printf change if the nowait clause is removed from the first for directive?

The second loop has to wait until the first finishes.

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

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

It uses less threads.

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

### 3. collapse.c

#### a. Which iterations of the loop are executed by each thread when the collapse clause is used?

(0) Iter (0 0)

(0) Iter (0 1)

(0) Iter (0 2)

(0) Iter (0 3)

(1) Iter (0 4)

(1) Iter (1 0)

(1) Iter (1 1)

(2) Iter (1 2)

(2) Iter (1 3)

(2) Iter (1 4)

(3) Iter (2 0)

(3) Iter (2 1)

(3) Iter (2 2)

(4) Iter (2 3)

(4) Iter (2 4)

(4) Iter (3 0)

(5) Iter (3 1)

(5) Iter (3 2)

(5) Iter (3 3)

(6) Iter (3 4)

(6) Iter (4 0)

(6) Iter (4 1)

(7) Iter (4 2)

(7) Iter (4 3)

(7) Iter (4 4)

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

No, it’s wrong, because the *j* variable is overwritten. We should use **private(j)** instead.

## C) Synchronization

### 1. datarace.c

#### a. Is the program always executing correctly?

No.

#### b. Add two alternative directives to make it correct. Explain why they make the execution correct.

#pragma omp parallel for schedule(dynamic,1) reduction(+:x)

for (i=0; i < N; i++) {

x++;

}

#pragma omp parallel for schedule(dynamic,1)

for (i=0; i < N; i++) {

#pragma omp atomic

x++;

}

### 2. barrier.c

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

We can’t predict the first one.

We can predict the second one.

We can’t predict the third one.

### 3. ordered.c

#### a. Can you explain the order in which the ”Outside” and ”Inside” messages are printed?

Outsite messages are printed in different orders each execution, and Inside messages are ordered always.

#### b. 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?

You can write **ordered(2)** instead of just ordered.

## D) Tasks

### 1. single.c

#### a. 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?

Because inside the parallel region the single construct makes that each task runs only once and the nowait clause allows the program to continue, making it parallel.

They are executed in burst because they run simultaneously in groups of 4, due to the omp\_set\_num\_threads(4) and the sleep(1) makes them wait.

### 2. fibtasks.c

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

Because there is no parallel construct, just a task.

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

...

void traverse\_list(struct node\* p){

while (p != NULL) {

printf("Thread %d creating task that will compute %d\n", omp\_get\_thread\_num(), p->data);

#pragma omp task

processwork(p);

p = p->next;

}

}

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

...

//Modification

#pragma omp parallel

#pragma omp single

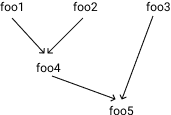
traverse\_list(p);

...

}

### 3. synchtasks.c

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



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

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();

#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

#### a. 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.

Between 5 and 10 iterations per task. Most of the time we get 2 tasks with 6 iterations each.

#### b. What does occur if the nogroup clause in the first taskloop is uncommented?

The second loop doesn’t wait the first to start, so they get mixed.

# Observing overheads

## Synchronisation overheads

This are the execution times we collected for each version:

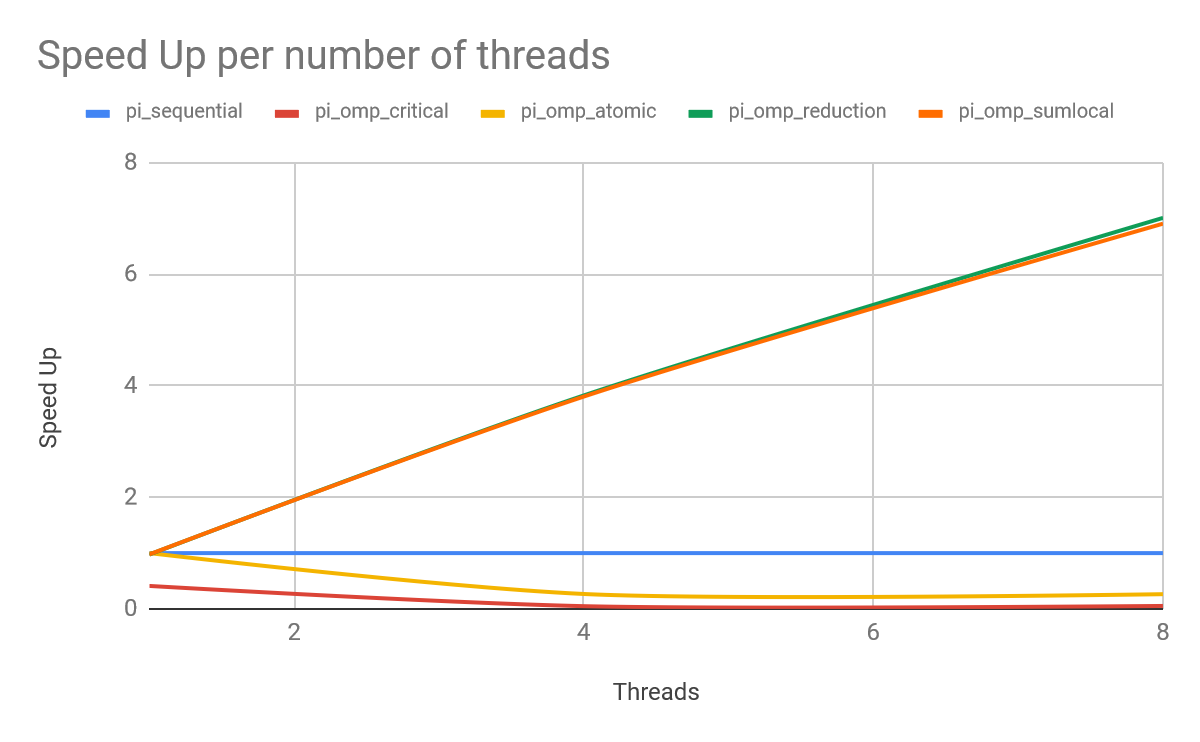
|  |  |  |  |
| --- | --- | --- | --- |
|  | 1 thread | 4 threads | 8 threads |
| pi\_sequential | 1.795932055 | 1.793233156 | 1.795014143 |
| pi\_omp\_critical | 4.364167 | 36.652714 | 33.930714 |
| pi\_omp\_atomic | 1.795376 | 6.708079 | 6.804043 |
| pi\_omp\_reduction | 1.841598 | 0.468692 | 0.256132 |
| pi\_omp\_sumlocal | 1.836557 | 0.471146 | 0.259977 |

### Gráfico

### Speed Ups

This are the calculated speed ups from the execution times:

|  |  |  |  |
| --- | --- | --- | --- |
|  | 1 thread | 4 threads | 8 threads |
| pi\_sequential | 1 | 1 | 1 |
| pi\_omp\_critical | 0.4115177203 | 0.04892497609 | 0.05290233925 |
| pi\_omp\_atomic | 1.000309715 | 0.2673243944 | 0.2638158141 |
| pi\_omp\_reduction | 0.9752030872 | 3.826037475 | 7.008160413 |
| pi\_omp\_sumlocal | 0.9778798344 | 3.806109265 | 6.904511334 |



### Analysis

The *pi\_omp\_critical* version takes a lot of time because the critical part is not executed in parallel and also the implementation of omp critical is slow.

The conclusion is that the best options are *pi\_omp\_reduction* and *pi\_omp\_sumlocal*. The other ones are actually worse than running the program sequentially. Those two are the best because they don’t stop the execution of the other threads.

## Thread creation and termination

A sample of the results of the pi\_omp\_parallel overhead calculations.

All overheads expressed in microseconds

Nthr Overhead Overhead per thread

2 1.9512 0.9756

3 1.6214 0.5405

4 1.6713 0.4178

5 1.7002 0.3400

21 2.9721 0.1415

22 3.0870 0.1403

23 3.0966 0.1346

24 3.1624 0.1318

The overhead grows with the numbers of threads, however, the overhead per thread decreases at the same time. For example the overhead for using two threads is approximately 0.98 μs per thread but the overhead for 24 threads is 0.13 μs per thread.

This means the overhead is not constant. There is an overhead for the parallel execution and an overhead for each thread that is initialized.

The overhead can be calculated with the following formula:

is the constant overhead for initializing parallelisation

is the overhead for adding a thread

is the number of threads

The overhead per thread can be calculated with the following formula:

## Task creation and termination

A sample of the results of the pi\_omp\_task overhead calculations.

All overheads expressed in microseconds

Ntasks Overhead Overhead per task

2 0.2484 0.1242

4 0.5126 0.1281

6 0.7573 0.1262

8 1.0013 0.1252

10 1.2479 0.1248

60 7.3070 0.1218

62 7.5606 0.1219

64 7.8043 0.1219

The overhead grows with the numbers of tasks but this time, the overhead per task stays the same. For example the overhead for using two tasks is approximately 0.12 μs per task and the overhead for 64 tasks is 0.12 μs per task.

This means the overhead grows linear to the number of tasks.

The overhead can be calculated with the following formula:

is the overhead for one task

is the number of threads

The overhead per thread can be calculated with the following formula: