PAR Laboratory Assignment

Lab 2: OpenMP programming model and analysis of overheads

Group 13-03

Carlos Bárcenas Holguera

Miquel Gómez i Esteve

### Facultat d’Informàtica de Barcelona - UPC

### March 2018

### Spring 2017-2018

#### **Index**

#### 

#### **Part I: OpenMP questionnaire……………………………....4**

#### **Part II: Parallelization overheads……………………………9**

#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### **Part I: OpenMP questionnaire**

**A) Basics**

**1.hello.c**

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

24 times. One per each thread by default.

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

We can add #pragma omp parallel num\_threads(4). Or we can execute the program with the clause OMP\_NUM\_THREADS : 4

**2.hello.c: Assuming the OMP NUM THREADS variable is set to 8 with "export OMP NUM THREADS=8"**

**1. Is the execution of the program correct? (i.e., prints a sequence of "(Thid) Hello (Thid) world!" being Thid the thread identifier) Which data sharing clause should be added to make it correct?.**

It’s not correct. We must add #pragma omp parallel private(id) clause.

**2. Are the lines always printed in the same order? Could the messages appear intermixed?**

No, the messages appear mixed and not always at the same order.

**3.how many.c: Assuming the OMP NUM THREADS variable is set to 8 with "export OMP NUM THREADS=8"**

**1. How many "Hello world ..." lines are printed on the screen?**

16 messages appear in total.

First -> 8 messages. Because we use the predefined threads omp\_num\_threads=8

Second -> 2 messages. Because we define a new number of threads, 2.

Third -> 3 messages. Because we define a local number of threads, but only in “local”

Fourth -> 2 messages. Because we define a new number of threads, 2.

Fifth -> 2 messages. Because we define a new number of threads, 2.

**2. If the if(0) clause is commented in the last parallel directive, how many "Hello world..." lines are printed on the screen?**

Fifth goes between 1 and 4, depends on the execution random.

**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 and firstprivate)?**

Shared -> The final result is 7 or 8. We share the variable within threads so increment in order.

Private -> The final result is 5. We don’t use the variable during the execution.

Firstprivate -> The fins result is 71. We use the last definition of the variable before the execution (before the parallel region).

**2. What needs to be changed/added/removed in the first directive to ensure that the value after the first parallel is always 8?.**

We must add #pragma omp parallel reduction (+ : x)

**5.parallel.c**

**1. How many messages the program prints? Which iterations is each thread executing?**

Total = 8+7+6+5 = 26. Desde la seva id fins a N-1.

**2. Change the for loop to ensure that its iterations are distributed among all participating threads.**

int main()

{

#pragma omp parallel num\_threads(NUM\_THREADS)

{

int id=omp\_get\_thread\_num();

int num\_threads=omp\_get\_num\_threads();

for (int i=id; i < N; i=i+num\_threads) {

printf("Thread ID %d Iter %d\n",id,i);

}

}

return 0;

}

The highlight part are the changes on the code, we have defined chunks of iterations of the num\_threads size.

**6.datarace.c (execute several times before answering the questions)**

**1. Is the program always executing correctly?**

No, sometimes is wrong. Because the x is a shared variable, that’s why every execution is different.

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

We can use the reduction clause. Or we can use #pragma omp atomic or #pragma omp critical before the ++x

**7.barrier.c**

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

Yes, because the sleep time depends on the id of this one.

No, because barrier only makes threads wait until the rest of threads reach this point of the code, but the threads can exit at different order.

**B) Worksharing**

**1.for.c**

**1. How many and which iterations from the loop are executed by each thread? Which kind of schedule is applied by default?**

Two, the two that correspond to each thread by order. The schedule by default is static.

**2. Which directive should be added so that the first printf is executed only once by the first thread that finds it?.**

We must add #pragma omp single.

**2.schedule.c**

**1. Which iterations of the loops are executed by each thread for each schedule kind?**

Schedule (static) -> The N iterations are divided by the number of threads.

Schedule (static,2) -> The N iterations are divided by the number of threads. En sections of 2.

Schedule (dynamic,2) -> Two iterations are assigned to each thread, when one finishes two more get assigned.

Schedule (guided,2) -> Like dynamic, but in every iteration the work decreases.

**3.nowait.c**

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

The loop1 waits to finish before start the loop2.

**2. If the nowait clause is removed in the second for directive, will you observe any difference?**

There’s no difference because the loop2 executes in order by default.

**4.collapse.c**

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

All the threads execute 3 iterations except the thread 0 witch executes 4.

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

It’s not correct, because the collapse makes i and j private by default. We must add the private(j) clause.

**5.ordered.c**

**1. Can you explain the order in which printf appear?**

The “before” iterations are not in order, but, in the other hand, the “inside” iterations are in order because the ordered clause makes effect.

**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 must change the schedule from dynamic to static,2

**6.doacross.c**

**1. In which order are the ”Outside” and ”Inside” messages printed?**

Outside messages not apply sync policy. Inside messages apply the (i-2) execution after (i) policy.

**2. In which order are the iterations in the second loop nest executed?**

In the second loop we need the execution of (i-1) and (j-1) after (i,j)

**3. What would happen if you remove the invocation of sleep(1). Execute several times to answer in the general case.**

Without sleep the order is different and executes faster, but the sync policy still applies.

**C) Tasks**

**1.serial.c**

**1. Is the code printing what you expect? Is it executing in parallel?**

Yes, the execution is correct. It executes in sequential, only one thread executes.

**2.parallel.c**

**1. Is the code printing what you expect? What is wrong with it?**

No, it’s not correct. The number are incorrect.

**2. Which directive should be added to make its execution correct?**

We must add #pragma omp single before #pragma omp task, to make that only one thread makes the processwork.

**3. What would happen if the firstprivate clause is removed from the task directive? And if the firstprivate clause is ALSO removed from the parallel**

**directive? Why are they redundant?**

It has no effect. Because, by default the variables are firstprivate.

**4. Why the program breaks when variable p is not firstprivate to the task?**

Because if it’s shared with the other threads, all try to access to this variable making a segmentation fault.

**5. Why the firstprivate clause was not needed in 1.serial.c?**

Because it executes in sequential, so only one thread executes.

**3.taskloop.c**

**1. Execute the program several times and make sure you are able to explain when each thread in the threads team is actually contributing to the execution of work (tasks) generated in the taskloop.**

#### **Part II: Parallelization overheads**

**1. Which is the order of magnitude for the overhead associated with a parallel region (fork and join) in OpenMP ? Is it constant? Reason the answer based on the results reported by the pi omp parallel.c code.**

The order of magnitude is microseconds.

Data obtained after executing the code interactively:

Nthr Overhead Overhead per thread

2 0.9613 0.4807

3 1.8581 0.6194

4 2.0398 0.5100

5 2.6422 0.5284

6 2.7757 0.4626

7 2.6623 0.3803

8 3.0212 0.3776

9 3.1447 0.3494

10 3.2425 0.3243

11 3.8412 0.3492

12 3.2856 0.2738

13 3.6398 0.2800

14 4.0950 0.2925

15 3.7270 0.2485

16 4.4439 0.2777

17 4.4990 0.2646

18 4.2406 0.2356

19 4.4945 0.2366

20 4.3488 0.2174

21 4.5959 0.2189

22 4.6534 0.2115

23 5.0225 0.2184

24 4.9318 0.2055

Data obtained when submitting the code for execution

Nthr Overhead Overhead per thread

2 1.7674 0.8837

3 1.7118 0.5706

4 2.2463 0.5616

5 2.7105 0.5421

6 2.8622 0.4770

7 2.6866 0.3838

8 3.0388 0.3799

9 3.4262 0.3807

10 3.1162 0.3116

11 3.9282 0.3571

12 3.4065 0.2839

13 3.8046 0.2927

14 4.3599 0.3114

15 3.9369 0.2625

16 4.7429 0.2964

17 4.7072 0.2769

18 4.3653 0.2425

19 4.5849 0.2413

20 4.3973 0.2199

21 4.7669 0.2270

22 4.9796 0.2263

23 5.0980 0.2217

24 5.2288 0.2179

With these executions we can see that obviously the more threads we have the more overhead the program needs. However, the overhead per thread we need decreases with the more threads we have, which allows us to prove that it is not constant.

**2. Which is the order of magnitude for the overhead associated with the creation of a task and its synchronization at taskwait in OpenMP ? Is it constant? Reason the answer based on the results reported by the pi omp tasks.c code.**

The order of magnitude is microseconds.

Data obtained after executing the code interactively:

Ntasks Overhead per task

2 0.0896

4 0.1245

6 0.1221

8 0.1227

10 0.1210

12 0.1199

14 0.1194

16 0.1184

18 0.1220

20 0.1213

22 0.1207

24 0.1199

26 0.1197

28 0.1183

30 0.1197

32 0.1186

34 0.1188

36 0.1186

38 0.1192

40 0.1192

42 0.1192

44 0.1213

46 0.1192

48 0.1171

50 0.1185

52 0.1179

54 0.1179

56 0.1175

58 0.1176

60 0.1175

62 0.1182

64 0.1193

Data obtained when submitting the code for execution:

Ntasks Overhead per task

2 0.1146

4 0.1172

6 0.1144

8 0.1148

10 0.1222

12 0.1244

14 0.1240

16 0.1246

18 0.1244

20 0.1236

22 0.1229

24 0.1222

26 0.1213

28 0.1203

30 0.1202

32 0.1202

34 0.1205

36 0.1198

38 0.1198

40 0.1196

42 0.1189

44 0.1190

46 0.1189

48 0.1189

50 0.1188

52 0.1186

54 0.1186

56 0.1183

58 0.1184

60 0.1184

62 0.1181

64 0.1180

The more tasks we generate the more overhead the program has. However, the overhead per task decreases when we generate more tasks, which means that it is not constant. Approximately after we generate the 34th tasks the overhead stabilizes, it seems to have reached a limit.

**3. Based on the results reported by the pi omp taskloop.c code, If you have to generate tasks out of a loop, what seems to be better: to use task or taskloop? Try to reason the answer.**

Data obtained after executing the code interactively:

Ntasks Overhead per task

2 0.0620

4 0.0345

6 0.0158

8 0.0133

10 0.0088

12 0.0076

14 0.0057

16 0.0036

18 0.0020

20 -0.0011

22 -0.0010

24 -0.0012

26 -0.0027

28 -0.0012

30 -0.0040

32 -0.0033

34 -0.0046

36 -0.0036

38 -0.0054

40 -0.0026

42 -0.0023

44 -0.0050

46 -0.0043

48 -0.0020

50 -0.0053

52 -0.0043

54 -0.0028

56 -0.0068

58 -0.0046

60 -0.0054

62 -0.0070

64 -0.0059

Data obtained when submitting the code for execution:

Ntasks Overhead per task

2 0.0307

4 0.0140

6 -0.0000

8 -0.0011

10 0.0086

12 0.0057

14 0.0035

16 0.0009

18 -0.0012

20 -0.0013

22 -0.0021

24 -0.0027

26 -0.0033

28 -0.0030

30 -0.0040

32 -0.0043

34 -0.0054

36 -0.0055

38 -0.0061

40 -0.0060

42 -0.0062

44 -0.0045

46 -0.0048

48 -0.0061

50 -0.0067

52 -0.0070

54 -0.0070

56 -0.0070

58 -0.0073

60 -0.0075

62 -0.0071

64 -0.0073

Taskloop generates a overhead per task which is almost negligible. Comparing the results from the two codes, one using tasks and the other one taskloop, we can see that the best option would be to use taskloop instead of task.

**4. Which is the order of magnitude for the overhead associated with the execution of critical regions in OpenMP? How is this overhead decomposed? How and why does the overhead associated with critical increase with the number of processors? Identify at least three reasons that justify the observed performance degradation. Base your answers on the execution times reported by the pi omp.c and pi omp critical.c programs and their Paraver execution traces.**



Fig 1: Trace execution of pi\_omp\_critical with 8 threads.



Fig 2: Trace execution of pi\_omp with 8 threads.

As seen in fig 1, the single part of the code is executed correctly, and afterwards we can see a race between all threads because all of them are trying to access the variable sum (with all the synchronizations it implies). The code is very inefficient as proven on the fig1 with all the red part. With all the threads waiting we see why we have that large overhead.

Comparing it with the fig 2 (trace execution for pi\_omp), we can see that most of the time is spent computing, opposed to all the time lost in the pi\_omp\_critical execution. At the end of the fig 2 trace, we see that there is still some red part, which is due to the critical part again.

The overhead execution it’s decomposed in: lock, unlock, locked status and unlocked status.

Possible performance degradation reasons:

* A lot of synchronizations per thread
* The more threads we have, the more overhead is added, because of its creation, synchronizations, etc.
* The code executed in the critical is important for the whole program to be efficient, so it is much more efficient to use a local variable like sumlocal, and then add it to the global variable instead of doing the whole operation on the critical part.

**5. Which is the order of magnitude for the overhead associated with the execution of atomic memory accesses in OpenMP? How and why does the overhead associated with atomic increase with the number of processors? Reason the answers based on the execution times reported by the pi omp.c and pi omp atomic.c programs.**

The atomic construct serializes the operation, and it is usually more efficient than the critical.

However we have the same problem as the previous exercise and because of all the threads waiting to execute the operation in the atomic construct the overhead is still very large.

|  |  |  |
| --- | --- | --- |
| Execution time for 1000000 iterations | 1 Thread | 8 Threads |
| pi \_omp | 0.011872s | 0.010936s |
| pi\_omp\_critical | 0.021405s | 0.136399s |

**6. In the presence of false sharing (as it happens in pi omp sumvector.c), which is the additional average time for each individual access to memory that you observe? What is causing this increase in the memory access time? Reason the answers based on the execution times reported by the pi omp sumvector.c and pi omp padding.c programs. Explain how padding is done in pi omp padding.c**

We’ve seen the time to access the memory has increased. That’s due to the false sharing. That happens when more than one thread changes the value of various memory positions that are in the same cache line. When a thread updates one of this positions, the other ones updates or invalidates their corresponding cache line depending on the coherence protocol.

In the omp\_padding, we substitute the sumvector for a matrix, and forcing every element to be in a different row.