LAB 0
SECOND DELIVERABLE
PAR-FIB

Members:
Víctor Pérez Martos
Albert Suàrez Molgó
Group: *par2205*
Date: *30/10/2015*
Academic course: *QT-2015/16*

Part I: OpenMP questionnaire

A) Basics

1. hello.c

1. 24 times, because the computer has 2 sockets with 6 cores each one and 2 threads for each core. So, the computer can execute up to 24 threads.
2. Executing the command line “export OMP_NUM_THREADS=4”. Depending on the operating system it may be “setenv” instead of “export”.

2. hello.c

1. No, because it should show two messages for each thread. We have to add “*id*” as a private element in the pragma declaration.
2. No, because the printing of the message is managed by the operating system and it may print the messages in a different order.

3. how_many.c

1. 16 lines: 8 “*first parallel’s*”, 2 “*second parallel’s*”, 3 “*third parallel’s*” and 1 “*five parallel’s*”.
2. It may print 16, 17, 18 or 19 lines. Depends on the random value set to “*five parallel’s*”.

4. data_sharing.c

1. After first parallel with *shared* attribute, the value of *x* is 8 (but it may change depending on the execution). After second parallel with *private* attribute, the value of *x* is 0. And after third parallel with *first private* attribute, the value of *x* is 0.
2. We have to remove *shared(x)* and add *reduction(+:x)*.

5. parallel.c

1. The program prints 8 messages. Each thread executes all the iterations with a number that applying module 4 has the same number as the thread.
2. We have to add *#pragma omp for schedule(static ,N/NUM_THREADS)* before the loop.

6. datarace.c

1. No, because *x* is a shared variable and it produces a data race program where all threads want to access it at the same time.
2. One possible alternative directive would be adding *reduction(+:x)* to *#pragma omp parallel*. Another possible option would be to add a *#pragma omp critical* or *#pragma omp atomic* directive before executing the incrementation of *x*.

7. barrier.c

1. After looking at the code and some executions of it, we can predict that the program will show first the sentences related to *going to sleep…* then the ones about entering the barrier and finally the ones of *we are awake*. Nevertheless, we can’t predict the exact order of the threads execution inside the statements showed before.

B) Worksharing

1. for.c

1. Each thread executes 2 iterations of the loop because *#pragma omp for* distributes the 16 iterations from the loop between the 8 existing threads.
2. The three first threads executes 3 iterations of the loop and the other 5 threads executes 2 iterations due to the *#pragma omp parallel for* distribution method.
3. We have to add *#pragma omp single* directive before the printf command.

2. schedule.c

1. On the first loop we have a static distribution with no chunk specification which distributes equally the number of iterations between the number of threads.
   On the second one we have a static distribution with the value 2 for the chunk clause which distributes equally in packages of 2 iterations the number of iterations between the number of threads.
   On the third loop we have a dynamic distribution with the value 2 for the chunk clause which distributes in packages of 2 iterations the number of iterations between the number of threads assigning them by execution order, so the iterations are given to the first thread who arrives.
   And on the fourth one we have a guided distribution with the value 2 for the chunk clause which distributes the number of iterations between the number of threads in packages until no chunks remains to be assigned.

3. nowait.c

1. The *#pragma omp for* directive has an implicit *omp barrier* directive at the end of for statement. By adding *nowait* we allow threads to skip this barrier and keep working. So, if we take out the *nowait* clause all threads must remain waiting at the end of the first loop until all threads are done with it. In consequence, it prints firstly the messages related to first loop and then all messages of the second one.
2. No, because the program ends after doing the second loop.

4. collapse.c

1. The *collapse* clause specifies how many loops are associated with the loop construction. So, the total number of iterations is 25 (5x5). Once we have this number the *#pragma omp parallel for* directive distributes equally these 25 iterations between the 8 threads.
2. No, because the work is done by only 5 threads. If we add the clause *private(j)*, we don't need to use the *collapse* clause.

C) Tasks

1. serial.c

1. The code prints a list of the first 25 numbers of the Fibonacci sequence, which is exactly what we were expecting after looking at the code. It is executed with no parallel region on it.

2. parallel.c

1. No, the result of the first 25 numbers of Fibonacci sequence is wrong. The problem with it is that it prints the numbers 4 times bigger.
2. We have to add *#pragma omp single* after *#pragma omp parallel firstprivate(p) num_threads(4)*.
3. If the *firstprivate* clause is removed from the task directive it does not change the result. But, if you remove the two *firstprivate* clauses it throws an error. It is redundant because it is only needed to be specified once.
4. Because *firstprivate* variables are captured in creation time while *shared* and *private* are not.
5. Because it is not executed in parallel.

Part II: Parallelization overheads

1.

Nthr	Time (µs)	Time per thread (µs)
2	2.147	1.0735
3	2.0141	0.6714
4	2.4567	0.6142
5	2.4998	0.5
6	2.6487	0.4415
7	2.9159	0.4166
8	3.5949	0.4494
9	3.4068	0.3785
10	3.6501	0.365
11	3.6353	0.3305
12	3.6395	0.3033
13	4.1179	0.3168
14	3.8494	0.275
15	4.1164	0.2744
16	4.4802	0.28
17	4.3227	0.2543
18	4.7996	0.2666
19	4.5374	0.2388
20	5.1559	0.2578
21	4.8085	0.229
22	5.292	0.2405
23	5.4106	0.2352
24	5.4671	0.2278

Table 1: Result of *pi_omp_overhead* execution.

The order of magnitude for the overhead associated for each thread is microseconds with a little variations between different executions. Time is not constant, because if we use more threads the fork and join time is greater but the time for each thread is lower.

By each thread we add, we get an additional overhead time. When the number of threads is really big, we get more stability between time and thread. So we could consider that the time per thread is about *0.2278* as we can see in table 1.

To calculate the cost, we also need to know the fixed cost, which we can approximately know by taking the total time when we use 2 threads and subtracting the time of 2 threads.
Fixed cost = 2.147 - 2*(0.2278) = 1.6914.

Approximately, **the cost is 1.6914 (fixed cost) + 0.2278 * #threads (variable cost)**

2.

#Threads	User	System	Elapsed	%CPU
1	0.79	0.00	0:00.79	99
8	0.89	0.00	0:00.12	730

Table 2: Result of *pi_omp* execution with 100.000.000 iterations.

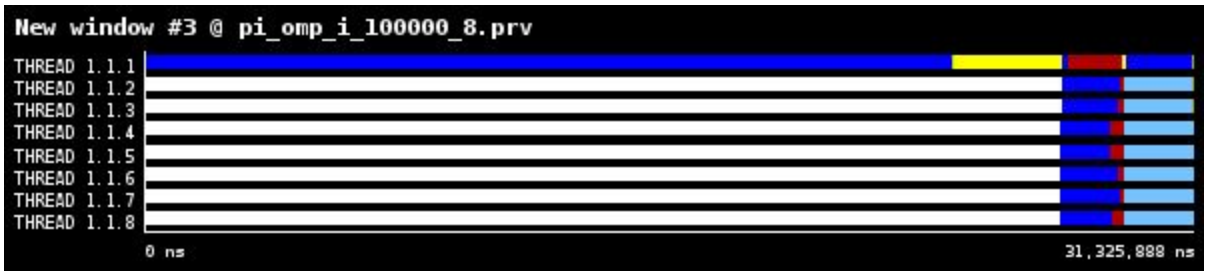

Image 1: Paraver's trace of *pi_omp* executions with 8 threads.

#Threads	User	System	Elapsed	%CPU
1	1.82	0.00	0:01.83	99
8	221.99	7.47	0:30.78	745

Table 3: Result of *pi_omp_critical* execution with 100.000.000 iterations.

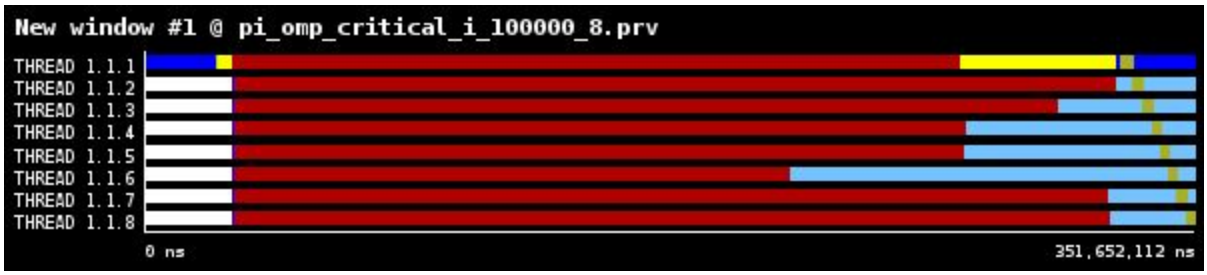

Image 2: Paraver's trace of *pi_omp_critical* executions with 8 threads.

	Unlocked status	Lock	Unlock	Locked status
THREAD 1.1.1	34.94 %	55.14 %	5.37 %	4.55 %
THREAD 1.1.2	20.66 %	69.56 %	4.96 %	4.82 %
THREAD 1.1.3	26.04 %	64.29 %	5.02 %	4.64 %
THREAD 1.1.4	34.87 %	55.44 %	5.18 %	4.51 %
THREAD 1.1.5	35.14 %	55.13 %	5.17 %	4.55 %
THREAD 1.1.6	51.91 %	37.95 %	5.35 %	4.79 %
THREAD 1.1.7	21.36 %	68.89 %	5.10 %	4.65 %
THREAD 1.1.8	21.14 %	68.99 %	5.17 %	4.70 %
Total	246.05 %	475.40 %	41.32 %	37.22 %
Average	30.76 %	59.43 %	5.16 %	4.65 %
Maximum	51.91 %	69.56 %	5.37 %	4.82 %
Minimum	20.66 %	37.95 %	4.96 %	4.51 %
StDev	10.04 %	10.13 %	0.13 %	0.11 %
Avg/Max	0.59	0.85	0.96	0.96

Table 4: *omp_critical_profile*.

To calculate the overhead time due to critical, we take the result time from the *pi_omp_critical* execution and we subtract from it the time obtained with the *pi_omp* execution. If we take this time and we divide it by the number of iterations of the program, we obtain the overhead associated to critical. So, in our case, it would be ($(30.78-0.12)/10^8$) = $3.066*10^{-7}$ seconds.

As we can see on table 4, this overhead is decomposed in three regions: fork, lock and join. Looking at the table, we can observe that the lock region is the one who takes more time from the execution, that is caused by the fact that all processors are constantly accessing to the variable *sum* and blocking it.

Obviously, this overhead associated with the access to *sum* causes that every thread we add increments the *lock* time. It also means time for *fork* and *join* but in lesser measure.

This problem can be solved by giving a local *sum* variable to each thread and, at the end, we should only do one critical access to *sum*. We can see on table 2, which implements this solution, how the time is reduced by applying this change.

3.

#Threads	User	System	Elapsed	%CPU
1	1.44	0.00	0:01.45	99
8	77.22	7.47	0:09.77	790

Table 4: Result of *pi_omp_atomic* execution with 100.000.000 iterations.

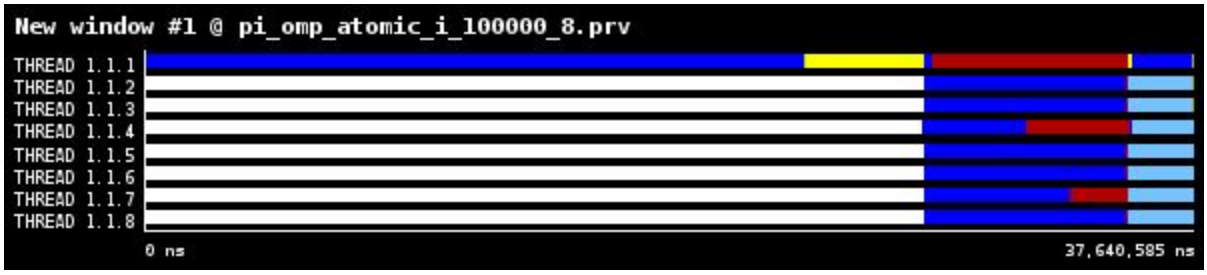

Image 3: Paraver's trace of *pi_omp_atomic* executions with 8 threads.

The atomicity from *pi_omp_atomic* unlike *pi_omp_critical* implies no locks between the different processors during execution. This results in an almost despicable overhead associated. Nevertheless, the execution time of the parallel region doesn't improve due to the lack of locks, so the time obtained is greater than the one from *pi_omp* execution as we can see from tables 2 and 4.

4.

#Threads	User	System	Elapsed	%CPU
1	0.79	0.00	0:00.79	99
8	5.15	0.00	0:00.66	771

Table 5: Result of *pi_omp_sumvector* execution with 100.000.000 iterations.

#Threads	User	System	Elapsed	%CPU
1	0.79	0.00	0:00.79	99
8	1.02	0.00	0:00.16	632

Table 6: Result of *pi_omp_padding* execution with 100.000.000 iterations.

Comparing both codes, we can appreciate that he only difference between them is how they store the values: while *sumvector* uses a vector, *pi_omp_padding* creates a matrix and stores each value in a different row of the matrix.

This change between using a matrix rather than a vector, allows the system to avoid false sharing, which is caused when two different threads try to modify different variables which are allocated in the same cache line. By using a matrix, we are allocating each value in a different cache line than the previous one.

The avoidance of false sharing, means that the system will do less accesses to memory and will save time as we can appreciate in the different times between *pi_omp_sumvector* i *pi_omp_padding* on tables 5 and 6.

5.

version	1 processor (s)	8 processors (s)	speed-up
pi_seq.c	0.9	-	1
pi_omp.c (sumlocal)	0.79	0.12	6.583
pi_omp_critical.c	1.83	30.77	0.059
pi_omp_lock.c	1.82	56.05	0.032
pi_omp_atomic.c	1.44	7.64	0.188
pi_omp_sumvector.c	0.79	0.62	1.274
pi_omp_padding.c	0.79	0.17	4.647

Table 2

On Table 2 we can observe the average of time obtained on different versions by doing several executions. With the results obtained, we can observe that *pi_omp_critical*, *pi_omp_lock* and *pi_omp_atomic* get a Speedup lesser than 1 due to the time of overhead. On the other side, we have *pi_omp_sumvector, pi_omp_padding* and *pi_omp* where we get a positive speedup. This means that the overhead associated with parallelization can determine the execution of a program. We can conclude that **sometimes less is more**.