Universitat Politècnica de Catalunya

Parallelism

Lab 2: OpenMP programming model and analysis of overheads

Carlota Catot Bragós
Ferran Martínez Felipe
PAR1110

Quadrimestre Primavera 2017-2018

Contents

1 Introduction

Learning the basic concepts of the topic in which are we going to work is the first step to success. After that, the next step is knowing all about the platform in wich we are going to work. That was the scope of the 3 first lab sessions of PAR laboratory.

However, to start working with parallelism that two important steps are not enough to start generating real parallelism in real applications. At this moment we know the hardware in which we are going to work, but a first approach to the software that we are going to use is needed.

This is the scope of the next two sessions of PAR laboratory, introduce the students to the OpenMP library, how it works and how we need to structure the code in order to generate real parallelism.

It is for that reason that in the following chapters we are going to introduce ourselves to de OpenMP library and the rules that are needed to be respected in order to work with real parallelism applications, from the simpler to the most complicated pragmas.

2 OpenMP Questionaire

2.1 Basics

2.1.1 hello.c v1

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

24 times. This result is due to the fact that #pragma omp parallel spreads the execution of the program between all the available threads in the cluster (in this case 24) and because all the threads are executing the printf line the hello world is being printed 24 times.

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

Changing the number of threads in which we are executing the programs
(by doing #export OMP_NUM_THREADS=4)

2.1.2 hello.c v2

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?

No, is not correct, because the threads are changing the value of the variable id between them (the variable is in a parallel region, and due to that is shared).

To correct the execution of the code it is enough to make the variable id private (change #pragma omp parallel to #pragma omp parallel private(id)).

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

No, the lines are not always printed in the same order, because we are not deciding the order in which the threads are making the prints (we are not making the execution sequential). In addition, the messages can appear intermixed. Example:

```
(1) Hello (1) world!
(0) Hello (0) world!
(3) Hello (3) world!
(7) Hello (7) world!
(6) Hello (6) world!
(5) Hello (5) world!
(2) Hello (4) Hello (4) world!
(2) world!
```

This is due to the fact that the private(id) is provoking that each thread executes one hello and one world (with its id number). The problem is that we are not controlling the order in which each thread are printing the messages, and we can have situations in which after one hello another thread prints another hello (as in the previous example).

2.1.3 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 times. It prints hello world:

- 8 times for the first region (#export OMP_NUM_THREADS = 8)
- 2 times for the second region (#omp_set_num_threads(2))
- 3 times for the third region (#pragma omp parallel num_threads(3))
- 2 times for the forth region (it is applied the same num threads than in the second region, because the third region num_threads doesn't change the global num_threads)
- 1 time for the fifth region (the if in the pragma is always false, and then there is not parallel region)

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

A random number of times between 1 and 4 (provoked by the following code):

```
srand(time(0));

#pragma omp parallel num_threads(rand()%4+1)
```

2.1.4 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)?

8, 5, 71 (Note: The 8 can change depending of how the threads are executed, because the variable is shared)

The first result is provoked by the sum of the ++x for all the threads that are being executed.

The second result is provoked by the effect of the private: Inside the parallel region the variable is uninitialized, and outside the parallel region the variable has the same value as before the parallel region (because the private creates a new copy of the variable).

The third result is provoked by the effect of the firstprivate: Inside the parallel region the variable has the same value as outside the parallel region, but each thread has his own copy of the variable (this is why the result is 72 inside the parallel region). Outside the parallel region the behaviour is the same as in the second result.

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

It needs to be done the following change:

```
#pragma omp parallel shared(x) {
    #pragma omp critical
    x++;
    printf("Within first parallel (shared) x is: %d\n",x);
}
```

This change is done in order to protect the increment of the variable x, in order to avoid data races.

2.1.5 parallel.c

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

26 messages. It is provoked because each thread executes all the for loop, and because the first value of i for each thread (it's thread id) and the fact that N is 8, the number of loops are 8 + 7 + 6 + 5 = 26 messages.

```
Iterations for thread:
Thread 0: 0,1,2,3,4,5,6,7
Thread 1: 1,2,3,4,5,6,7
Thread 2: 2,3,4,5,6,7
Thread 3: 3,4,5,6,7
```

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

```
#pragma omp parallel num_threads(NUM_THREADS) {
    int id=omp_get_thread_num();
    #pragma omp for schedule(static)
    for (int i=0; i < N; i=i+1) {
        printf("Thread ID %d Iter %d\n",id,i);
    }
}
```

2.1.6 datarace.c

1. Is the program always executing correctly?

No, there is a data race. Example:

CORRECT VALUE: Congratulations!, program executed correctly (x = 1024)

WRONG VALUE: Sorry, something went wrong, value of x = 896

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

Solution 1:

```
#pragma omp parallel private(i) {
        int id=omp_get_thread_num();
        for (i=id; i < N; i+=NUM_THREADS) {
                #pragma omp critical
                x++;
        }
}
```

Solution 2:

```
#pragma omp parallel private(i) {
        int id=omp_get_thread_num();
        for (i=id; i < N; i+=NUM_THREADS) {
                #pragma omp atomic
                x++;
        }
}
```

The two solutions are based on the same, and this is the protection the operation of x++. In the first solution we make the region inside the pragma critical a region that can only been executed for one thread at once. The second solution makes the operation of x++ indivisible, forcing to make all the read and write on one thread before allowing another thread to access to the variable.

2.1.7 barrier.c

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

Not at all. We can predict that the messages of going to sleep are going to be printed before the messages of being awake, and that the messages of being awake are going to be printed before the messages of crossing the barrier. In addition, we can predict the sequence of threads being awake. Anyway, we cannot predict the order to the messages to going to sleep and the order of the threads to cross the barrier.

In addition, we must say that the threads doesn't exit the barrier in any specific order.

2.2 Worksharing

2.2.1 for.c

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

Each thread is executing 2 consecutive iterations of the for loop (thread 0 do iterations 0 and 1, thread 1 do iterations 2 and 3, ...). This is because on one hand by default the pragma for works using the static schedule, and on the other hand the static gives the same iterations to each thread (if no chunk_size is specified)

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

#pragma omp single

2.2.2 schedule.c

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

- **Static**: In this case the iterations are equally distributed between all the threads, that means that each thread executes 1/2 of the total iterations consecutively.
- **Static (chunk_size = 2)**: In this case the iterations are equally distributed between all the threads, but in with the difference that in this case each thread is executing only 2 consecutive iterations of the for loop each time (respecting the thread order).
- **Dynamic (chunk_size = 2)**: In this case each thread is executing two consecutive iterations of the for loop each time, but with the difference that in this case each group of 2 iterations is executed for the first thread that is free.
- **Guided (chunk_size = 2)**: Similar to dynamic, with the difference that in this case the chunk_size only defines a lower bound for the number of consecutive iterations done for a thread (but a thread can do more in order to balance the work between processors).

2.2.3 nowait.c

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

The nowait clause is used to remove the synchronization time between threads. With the clause, the threads doesn't wait to the others at the end of the execution of the parallel region, they continue executing code. That means that without the nowait the sequence of printf shows first all the executions of the Loop 1 and then the executions of the Loop 2. Without the nowait all the Loop printf are printed randomly (but respecting the code sequence in a thread).

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

There is no difference, because after the second parallel regions the code execution ends.

2.2.4 collapse.c

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

A group of iterations of the intern loop are being executed one after another (collapse clause group sets of iterations in the inner loop as a only iteration of the outer loop). That means that each thread executes consecutively a set of consecutive iterations that corresponds to the sequential iterations of the loops combined.

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

The execution is not correct (some iterations are being repeated). A way to solve this problem is make private the variable j (using #pragma omp parallel for private(j)) to avoid that different iterations of the loop executed in different threads override the variable j, changing the correct behavior of the intern for loop.

2.2.5 ordered.c

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

Not at all. We can ensure that inside the ordered pragma all the iterations are going to be printed in order, but before the ordered we cannot be sure about what printf is going to be printed next (the first thread to arrive is the first thread to be executed).

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?

By changing the scheduling to static and adding an ordered. Example:

```
#pragma omp for schedule(static, 2) ordered
for (i=0; i < N; i++) {
    int id=omp_get_thread_num();
    #pragma omp ordered
    printf("Loop 1 - (%d) gets iteration %d\n",id,i);
}
```

2.2.6 doacross.c

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

The Outside message is printed randomly, but the inside message depends of i. This behavior is happening due to the #pragma omp ordered depend(sink: i-2), that sets the wait point for the completion of computation in iteration i-2.

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

Each iteration (i, j) depends of the iterations (i, j - 1) and (i - 1, j). That fact shows that to process an iteration first the following iterations must be processed: (i-1), (j), (1), (j-1).

If the loops starts at (1, 1) (the first loop and the only one without dependencies) the two next that are free of dependencies are (1, 2) and (2, 1). After processing the iteration (2, 1), the iteration (1, 3) and (2, 2) will be free, and so on.

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

The execution order is not respected anymore, due to the fastest thread is going to execute one of the new two possible iterations without dependencies, generating a different result.

2.3 Tasks

2.3.1 serial.c

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

Yes, it is (it's printing all the fibbonacci numbers). Anyway, the execution is not parallel (all the work is done in the thread 0).

2.3.2 parallel.c

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

No, it isn't. It's not calculating correctly the first numbers of the Fibbonacci list, and the error is extended to the rest of the execution due to the way of calculating the Fibonacci numbers.

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

The problem is that we are generating a task for each thread (with the same execution of the processwork function). A way to solve this problem is generating the task only one time, making the following change:

```
while (p != NULL) {
    #pragma omp single
    #pragma omp task firstprivate(p)
    processwork(p);
    p = p->next;
}
```

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?

If we remove the firstprivate clause from the task directive it changes nothing, because the firstprivate clause in the task generation is redundant with the one of the pragma parallel. If we remove also the firstprivate clause of the parallel directive we are going to obtain a segmentation fault. The two firstprivate are redundant because they are defining the same initialized private variable inside the parallel region.

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

By removing the firstprivate clause of the two directives we are going to obtain a segmentation fault. That is because all threads are going to work with the same p pointer, breaking the execution of the for loop by the use of p = p $\rightarrow$ next() for different threads.

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

Because the execution is done sequentially. This provoke that although we are generating tasks this tasks will be executed by a single thread, and because that we don't need the variable p to be firstprivate or not.

2.3.3 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.

What is happening is that one of the threads (the first that arrive to the pragma #omp parallel single) is creating all the tasks sequentially, in order to execute after the creation the tasks between all the threads (the tasks remains in a taskpool waiting to be executed by the first available thread).

The creation of the tasks are done following the same schema:

- The first available thread generates T1 and T2.
- Inside T2 the thread generates T3 and T4.
- Inside T4 the thread generates N tasks.

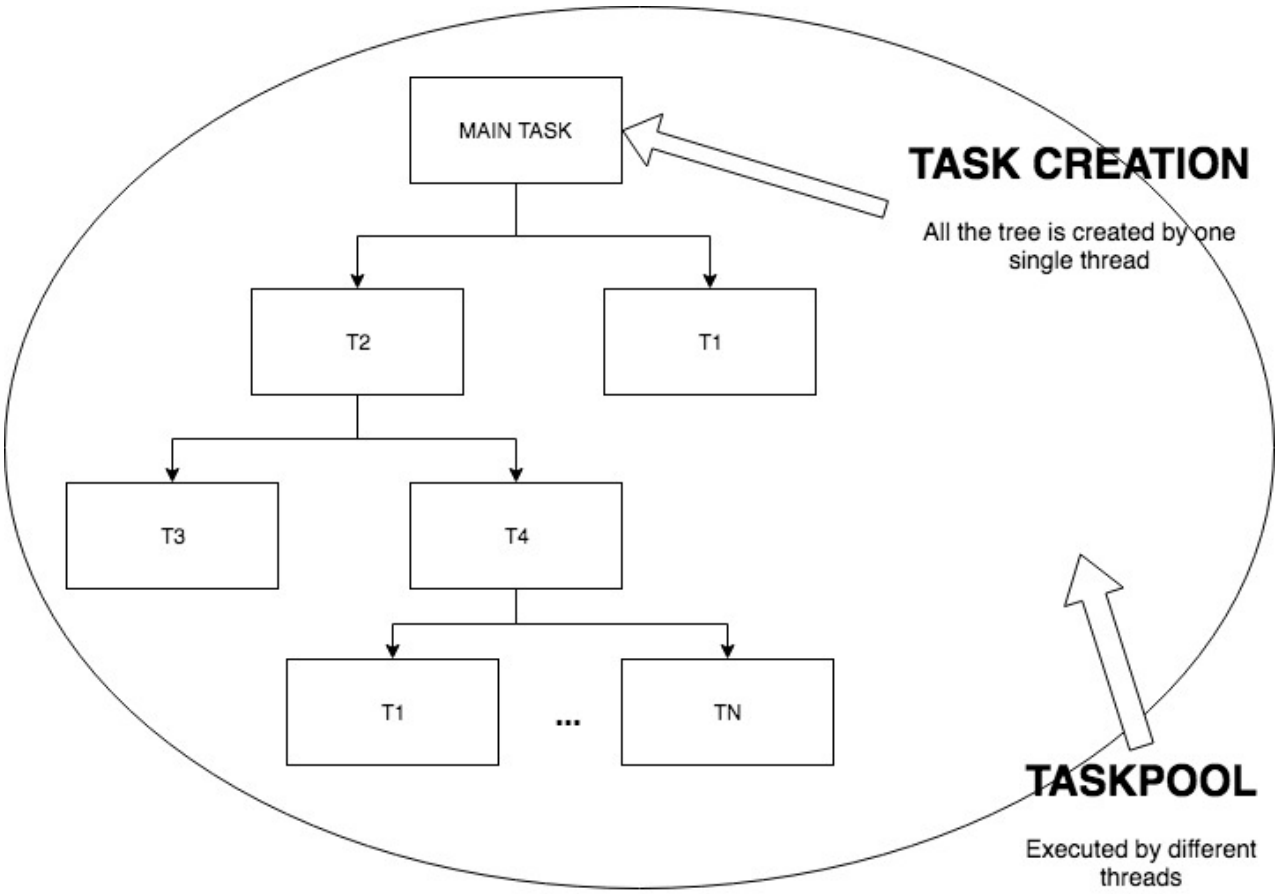

Figure 1: Task Graph

3 Parallelization overheads

3.1 Thread creation and termination

In this section we learned about the thread creation and termination, we see it through the file *pi_omp_parallel.c*.

In the results of the execution we can see that the order of magnitude of the overhead generated in the threads creation is in microseconds, we can see it clearly in figure 2 or 3, when the execution starts, the first message informs you that the overheads are expressed in microseconds.

By watching the results on the execution of *pi_omp_parallel* with 1 iteration and a maximum of 24 threads comparing the two execution versions interactive (figure 2) and queue execution (figure 3) that there is no visible difference in the different executions.

So, seeing one of the executions, we can prove that the time of the overhead is not constant, it increase linearly when the number of threads increase. However, the overhead per thread tends to decrease because when the number of threads increases, every thread has to do do less calculations.

```
boada-1:~/lab2/overheads\$ ./run-omp.sh pi_omp_parallel 1 24

make: 'pi_omp_parallel' is up to date.

All overheads expressed in microseconds

Nthr        Overhead            Overhead per thread
2           1.7756              0.8878
3           2.0646              0.6882
4           2.1585              0.5396
5           2.4041              0.4808
6           2.8677              0.4780
7           2.7454              0.3922
8           3.2307              0.4038
9           3.3295              0.3699
10          3.3915              0.3392
11          3.8651              0.3514
12          3.5965              0.2997
13          3.8513              0.2963
14          4.3382              0.3099
15          4.1014              0.2734
16          4.7833              0.2990
17          4.8757              0.2868
18          4.2886              0.2383
19          4.6536              0.2449
20          4.5649              0.2282
21          4.5456              0.2165
22          4.9912              0.2269
23          5.2452              0.2281
24          5.1591              0.2150

12.69user 0.11system 0:00.87elapsed 1462\%CPU (0avgtext+0avgdata 2056maxresident)k
0inputs+0outputs (0major+131minor)pagefaults 0swaps
```

Figure 2: Execution of **pi_omp_parallel.c** (interactive version)

```
boada-1:~/lab2/overheads\$ qsub -l execution submit-omp.sh pi_omp_parallel 1 24

All overheads expressed in microseconds
Nthr        Overhead            Overhead per thread
2           0.9931              0.4965
3           1.8887              0.6296
4           2.2482              0.5621
5           2.8934              0.5787
6           2.9807              0.4968
7           2.6141              0.3734
8           3.2926              0.4116
9           3.5497              0.3944
10          3.3294              0.3329
11          3.7432              0.3403
12          3.5117              0.2926
13          3.5941              0.2765
14          4.3749              0.3125
15          4.0106              0.2674
16          4.5941              0.2871
17          4.7804              0.2812
18          4.2347              0.2353
19          4.5980              0.2420
20          4.4383              0.2219
21          4.8699              0.2319
22          4.9696              0.2259
23          5.6689              0.2465
24          5.4413              0.2267
```

Figure 3: Execution of **pi_omp_parallel.c** (execution in queue version)

3.2 Task Creation and Synchronization

In this section we lerned about mesuring the overhead related with the creation of **task** and their synchronization at a **taskwait**, we see it in the file *pi_omp_task.c*.

In the results of the execution we can see that the order of magnitude of the overhead generated in the threads creation is in microseconds, we can see it clearly in figure 4 or 5, when the execution starts, the first message informs you that the overheads are expressed in microseconds.

By watching the results on the execution of *pi_omp_tasks* with 10 iterations and one thread, we can see that in figure 5 with the interactive version and in figure 4 with queue execution version, that it doesn't matter the number of tasks, the overhead per task will cost the same (creation and syncronization) independently of the number of tasks, and there is no visible difference between the two different executions.

So to take a conclusion of this part, we can say that the Overhead per task is constant due to the variation is almost null, this is because of the inestability of the maquine.

```
boada:~/lab2/overheads\$ qsub -l execution submit-omp.sh pi_omp_tasks 10 1
All overheads expressed in microseconds
Ntasks  Overhead per task
2           0.1596
4           0.1165
6           0.1152
8           0.1148
10          0.1220
12          0.1239
14          0.1233
16          0.1240
18          0.1238
20          0.1227
22          0.1221
24          0.1212
26          0.1207
28          0.1196
30          0.1196
32          0.1186
34          0.1196
36          0.1194
38          0.1191
40          0.1188
42          0.1187
44          0.1184
46          0.1182
48          0.1179
50          0.1183
52          0.1181
54          0.1180
56          0.1178
58          0.1178
60          0.1177
62          0.1180
64          0.1174
```

Figure 4: Execution of **pi_omp_tasks.c** (queue execution version)

```
boada:~/lab2/overheads\$ ./run-omp.sh pi_omp_tasks 10 1

make: 'pi_omp_tasks' is up to date.

All overheads expressed in microseconds

Ntasks  Overhead per task
2           0.1628
4           0.1230
6           0.1244
8           0.1257
10          0.1233
12          0.1235
14          0.1235
16          0.1241
18          0.1234
20          0.1230
22          0.1217
24          0.1213
26          0.1210
28          0.1206
30          0.1202
32          0.1193
34          0.1205
36          0.1205
38          0.1201
40          0.1199
42          0.1194
44          0.1193
46          0.1188
48          0.1193
50          0.1195
52          0.1193
54          0.1195
56          0.1194
58          0.1191
60          0.1189
62          0.1185
64          0.1187

2.94user 0.00system 0:02.94elapsed 99\%CPU (0avgtext+0avgdata 1664maxresident)k
0inputs+0outputs (0major+101minor)pagefaults 0swaps
```

Figure 5: Execution of **pi_omp_tasks.c** (interactive version)

3.3 Task vs. Tasklopp

In this section we lerned about comparing the overheads of **task** and **taskloop**, we can see it in the file *pi_omp_taskloop.c*.

We can assume that the time difference is constant, although also we see that it decreases in the third decimal. This is due to the calculation that makes the difference between task and taskloop, is visible in the third decimal and is noticeably fastest taskloop than task but being in the third decimal becomes almost insignificant.

Therefore, as the discussed above, for a large number of tasks it's better to use taskloop so it's seen that manages better the task management although it's quite insignificant with the numbers we see in figure 6 or figure 7, we can see that there is a slight trend that would become more significant when performing many tasks. We execute also the code in the two versions, and we can see that it still the same, there is no visible difference between the two different executions, figure 7 for interactive version and figure 6 for queue execution version.

```
boada:~/lab2/overheads\$ qsub -l execution submit-omp.sh pi_omp_taskloop 10 1
All overheads expressed in microseconds
Ntasks  Overhead per task
2       0.0225
4       0.0091
6       -0.0004
8       0.0007
10      0.0102
12      0.0047
14      0.0028
16      0.0000
18      -0.0017
20      -0.0019
22      -0.0027
24      -0.0032
26      -0.0044
28      -0.0039
30      -0.0047
32      -0.0051
34      -0.0059
36      -0.0063
38      -0.0065
40      -0.0066
42      -0.0065
44      -0.0071
46      -0.0073
48      -0.0075
50      -0.0082
52      -0.0082
54      -0.0079
56      -0.0078
58      -0.0082
60      -0.0085
62      -0.0084
64      -0.0083
```

Figure 6: Execution of **pi_omp_taskloop.c** (interactive version)

```
boada:~/lab2/overheads\$ ./run-omp.sh pi_omp_taskloop 10 1
gcc -Wall -O0 -std=c99 -fopenmp pi_omp_taskloop.c -o pi_omp_taskloop
All overheads expressed in microseconds
Ntasks  Overhead per task
2       0.0683
4       0.0313
6       0.0158
8       0.0134
10      0.0089
12      0.0046
14      0.0035
16      0.0011
18      -0.0006
20      -0.0010
22      -0.0025
24      -0.0028
26      -0.0035
28      -0.0043
30      -0.0050
32      -0.0046
34      -0.0052
36      -0.0056
38      -0.0054
40      -0.0055
42      -0.0054
44      -0.0058
46      -0.0068
48      -0.0066
50      -0.0072
52      -0.0068
54      -0.0072
56      -0.0073
58      -0.0065
60      -0.0085
62      -0.0073
64      -0.0083

4.14user 0.00system 0:04.14elapsed 99\%CPU (0avgtext+0avgdata 1812maxresident)k
0inputs+0outputs (0major+121minor)pagefaults 0swaps
```

Figure 7: Execution of **pi_omp_taskloop.c** (interactive version)

3.4 Thread synchronization: *critical*

In this section we lerned about the use of *critical* as one of the mechanisms for synchronization, we will work with the files *pi_omp_critical.c*, *pi_omp.c*, *pi_seq.c*.

```
boada:~/lab2/overheads\$ ./run-omp.sh pi_omp_critical 100000000 1
make: 'pi_omp_critical' is up to date.
Total execution time: 1.795388s
Number pi after 100000000 iterations = 3.141592653590426
1.79user 0.00system 0:01.79elapsed 99\%CPU (0avgtext+0avgdata 1848maxresident)k
0inputs+0outputs (0major+80minor)pagefaults 0swaps
```

Figure 8: Execution of **pi_omp_critical.c**

For one thread:

By analizing the code, we find that in the *pi_omp* code, the critical is outside the loop, and in the case of *pi_omp_critical* the critical is inside the loop. Because of that we can see the reason why the *pi_omp_critical* takes more time than *pi_omp*. In the first case, more accesses to memory will be made and although we only have one thread the memory and locks will be prepared in each step by the #pragma omp critical.

In conclusion, we see that the #pragma omp critial have an overhead that provoke that if we put it inside the loop, generate a lineal overhead with the number of iteration that the loop have, and this makes that take more time.

```
boada:~/lab2/overheads\$ ./run-omp.sh pi_omp_critical 100000000 8
make: 'pi_omp_critical' is up to date.
Total execution time: 46.414796s
Number pi after 100000000 iterations = 3.141592653589781
357.90user 7.79system 0:46.41elapsed 787%CPU (0avgtext+0avgdata 1904maxresident)k
0inputs+0outputs (0major+267minor)pagefaults 0swaps
```

Figure 9: Execution of **pi_omp_critical.c**

For eight threads:

We can see that the behavior is the same for the *pi_omp* case, with the proviso that now the for will be shared among the 8 threads reducing considerably the execution time due to the critical stills executing one time outside the loop.

For the *pi_omp_critical* case, happens just the opposite, now insted of one threads made the critical region, each loop have to be syncronized by the 8 threads, making that now this takes more time in do the calculation.

The order of magnitude we see it in figures 8 and 9 where the total exevution time is mesure in seconds, so for 100.000.000 the order of magnitude is taken in seconds.

The three reasons that justify all the specify over, are:

1. **Creation and termination of tasks**: There are more overheads because there are more critical regions cause the pragma omp critical executes in each iteration of the loop.
2. **Synchronization**: In each iteration of the loop there is a cost of synchronization for the critical of the 8 threads, in the case of one thread (figure 10) it doesn't have to be synchronized with anyone therefore this time is saved. This is seen in the red zone in the traces, we can see that for 1 thread (figure 10) we have less time than for 8 threads (figure 11).
3. **Locks and unlocks**: In each iteration of the loop there is a variable sum that makes a lock and unlock for each iteration, this is independent for the number of threads because it only will be done by one thread each time.

Figure 10: Traces with 1 thread

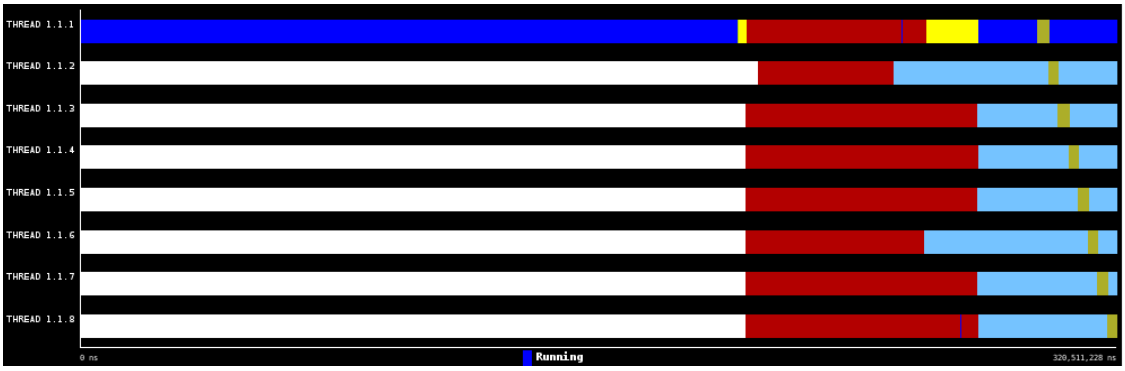

Figure 11: Traces with 8 threads

3.5 Atomic memory access

```
boada:~/lab2/overheads\$ ./run-omp.sh pi_omp_atomic 100000000 1
make: 'pi_omp_atomic' is up to date.
Total execution time: 1.529913s
Number pi after 100000000 iterations = 3.141592653590426
1.52user 0.00system 0:01.53elapsed 99\%CPU (0avgtext+0avgdata 1644maxresident)k
0inputs+0outputs (0major+75minor)pagefaults 0swaps
```

Figure 12: Execution of **pi_omp_atomic.c** with 1 thread

```
boada:~/lab2/overheads\$ ./run-omp.sh pi_omp_atomic 100000000 8
make: 'pi_omp_atomic' is up to date.
Total execution time: 8.303363s
Number pi after 100000000 iterations = 3.141592653589727
64.48user 0.00system 0:08.30elapsed 776\%CPU (0avgtext+0avgdata 1892maxresident)k
0inputs+0outputs (0major+188minor)pagefaults 0swaps
```

Figure 13: Execution of **pi_omp_atomic.c** with 8 threads

The order of magnitude for the overhead is in seconds, we can see it in the execution of the code with 1 thread (figure 12) or 8 threads (figure 13). The overhead is big because all threads are trying to get a little (because only protects writes) than critical, but, the overhead is similar because we have 1 write, doing same overhead.

3.6 False data sharing

As we use caches, we have to maintain a certain memory coherence, so every time a thread modifies memory addresses stored in the same cache line as the rest of threads, every cache must make a flush (what literally means setting all the validity bits to zero).

This will have a big repercussion because a fast access to a cache will result in a slow access to the principal memory.

The *padding.c* solves that problem, as before we were using a vector of doubles where elements of different threads took place in the same cache line (of different caches), now we assume that elements of different threads are in different cache lines, we improve the time to access memory significantly respect to the *pi_omp_sumvector.c* giving up some additional space per thread.

4 Conclusions

It is clear that OpenMP has changed the way in which we focus a parallel application. Nowadays libraries like OpenMP allow the users to generate parallelism without thinking about all the intrinsical problems of the parallelism.

The session 1 shows how different pragmas works in OpenMP. From the simplest ones to understand (like parallel) to the most complicated (like ordered) all the pragmas shown could be usefull in a variety of situations. And that is the objective of this session, allow the students to know the pragmas, their uses and different examples of how all of them work, a little step to us to generate real parallelism in real applications.

The session 2 is a step inside the problems of the parallelism. This session shows that when we go from the theorical world to the real one generating parallelism becomes a difficult task. In the real world overheads have to be taken into account, either for thread generation, task synchronization, task generation, etc.

In conclusion, if the first 3 laboratory sessions shows the surroundings about the parallelism (hardware, simulator, etc), this 2 sessions shows the inside of the parallelism (library usage and problems of generating parallelism). After all this work, we should be ready to start making our own first parallel applications.