

# PARALLELISM

## Laboratory 2: Brief tutorial on OpenMP programming model

## Index

1. OpenMP questionnaire							
1.1 Day 1: Parallel regions and implicit tasks	.	.	.	.	.	.	page 1-5
1.2 Day 2: explicit tasks	.	.	.	.	.	.	page 5-8
2. Overheads	.	.	.	.	.	.	page 8-9

# 1. OpenMP questionnaire

## 1.1 Day 1: Parallel regions and implicit tasks

### 1.hello.c

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

I can see it 2 times. One time for every thread that enters the parallel region. In this case, with `./hello`, executes the parallel region 2 threads (every thread print Hello world! and die).

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

I can do it with the command `"OMP_NUM_THREADS=4 ./1.hello"`. If we do that, every thread will execute one time the parallel region and in it, everyone will print Hello world! Then they will die. Finally we will see in the terminal the 4 messages of Hello world!

### 2.hello.c:

1. 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?

We can see in the picture that the identifier of the threads doesn't correspond sometimes with the next identifier. For example in the 4rd line, the identifier of Hello is 2 and for world! is 6.

```
par3110@boada-1:~/lab2/openmp/Day1$ ./2.hello
(0) Hello (3) Hello (3) world!
(3) world!
(6) Hello (6) world!
(2) Hello (6) world!
(1) Hello (1) world!
(5) Hello (5) world!
(4) Hello (5) world!
(7) Hello (7) world!
```

My solution is to add `private(id)` in the line that have `#pragma omp parallel num_threads(8)` like this: `#pragma omp parallel num_threads(8) private(id)`.

Now all the id are private and they will use the same identifier in the same line because the variable is not shared and it will have the same value.

2. Are the lines always printed in the same order? Why the messages sometimes appear inter-mixed? (Execute several times in order to see this).

```
par3110@boada-1:~/lab2/openmp/Day1$ ./2.hello
(0) Hello (3) Hello (3) world!
(2) Hello (2) world!
(6) Hello (0) world!
(1) Hello (1) world!
(6) world!
(5) Hello (5) world!
(4) Hello (4) world!
(7) Hello (7) world!
```

As we can see in the picture, the execution can be intermixed because all the threads are executing the parallel version without any order and two threads can print at the same time.

### 3.how many.c:

Assuming the OMP NUM THREADS variable is set to 8 with "OMP NUM THREADS=8 ./.3.how many". What does omp\_get\_num\_threads return when invoked outside and inside a parallel region?

It returns the number of threads that is running at the moment.

2. Indicate the two alternatives to supersede the number of threads that is specified by the OMP NUM THREADS environment variable.

Firstly, we can modify the number of threads with the directive "omp\_set\_num\_threads(X);" being X the number of threads.

Secondly, we can define how many threads will be in the directive #pragma omp parallel(num\_threads(X)) being X the number of threads.

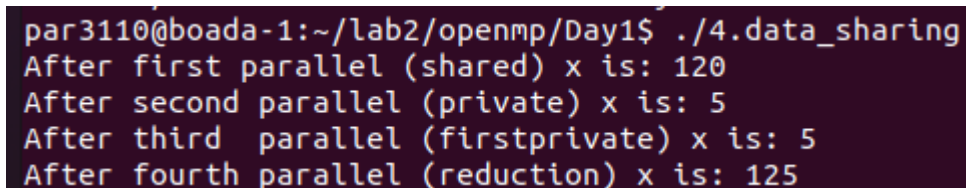
3. Which is the lifespan for each way of defining the number of threads to be used?

The lifespan is the time that is needed to execute the parallel region.

When a thread enters in a parallel region, a single thread executes all the tasks to do and the others run them. When the single thread ends the creation, if there are more tasks to run, it runs one. So, all threads are running when tasks are available.

### 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, firstprivate and reduction)? Is that the value you would expect? (Execute several times if necessary)



```
par3110@boada-1:~/lab2/openmp/Day1$ ./4.data_sharing
After first parallel (shared) x is: 120
After second parallel (private) x is: 5
After third parallel (firstprivate) x is: 5
After fourth parallel (reduction) x is: 125
```

The picture shows the value after the execution of the data-sharing attribute.

In the shared, the value will be modified by every thread, so, the first thread with id = 0 will sum  $x += 0$ , the next will do  $x += 1$  and this to the thread id number 15 so  $x$  will be  $0 + 1 + 2 + 3 + 4 + \dots + 15 = 120$ . So is the expected value.

The second parallel with private, all the variables  $x$  will be privatized and the modifications that will be used won't affect the global result. So the value is the expected because it is the same as before the parallel region.

The third parallel is firstprivate, is the same as in private clause, but is initialized to 5 in every thread, but it won't affect the final result because it is privatized only in every thread.

The fourth parallel is with reduction( $+:x$ ), and it will sum the value in every thread ( $0+1+2+3+\dots+15 = 120$ ). It had an initial value of 5 so  $x$  will be  $120 + 5 = 125$ . This is the expected value.

## 5.datarace.c

1. Should this program always return a correct result? Reason either your positive or negative answer.

It could be wrong because maxvalue is a shared variable and the execution of the threads don't have any order. So, it can be possible that when the instruction enter in the instruction `if (vector[i] > maxvalue) maxvalue = vector[i];` and simultaneously, another thread enters in the if condition, with the last value of maxvalue and rewrites the maxvalue with a wrong value. It will be a data race.

2. Propose two alternative solutions to make it correct, without changing the structure of the code (just add directives or clauses). Explain why they make the execution correct.

One option could be with `#pragma omp critical` before the if inside the for.

Another option that I thinked is to have a vector declared before the parallel and only the size of the vector is the same as the threads and only the thread can access to the vector[id] where id is the identifier. In every position will be the maximum partial and when the parallel region ends, go over the vector to choose which is the maximum.

3. Write an alternative distribution of iterations to implicit tasks (threads) so that each of them executes only one block of consecutive iterations (i.e. N divided by the number of threads).

```
#define N 1 << 20
int vector[N]={0, 0, 0, 1, 2, 3, 4, 5, 6, 7, 15, 14, 13, 12, 11, 10, 9, 8, 15, 15};
int maxvalue[8] = {0,0,0,0,0,0,0,0};

int main()
{
    int i;

    omp_set_num_threads(8);
    #pragma omp parallel private(i)
    {
        int id = omp_get_thread_num();
        int howmany = omp_get_num_threads();
        int inici = id * (N/howmany);
        int fi = (id + 1) * (N/howmany);

        for (i=inici; i < fi; ++i) {
            if (vector[i] > maxvalue[id])
                maxvalue[id] = vector[i];
        }
    }

    int max = 0;
    for(int j = 0; j < 8; ++j) if(maxvalue[j] > max) max = maxvalue[j];

    if (max==15)
        printf("Program executed correctly - maxvalue=%d found\n", max);
    else printf("Sorry, something went wrong - incorrect maxvalue=%d found\n", max);

    return 0;
}
```

## 6.datarace.c

1. Should this program always return a correct result? Reason either your positive or negative answer.

No because can be a data race with the countmax++. It is a shared variable and it will affects to the other threads, so it can be possible to do countmax++ several times.

2. Propose two alternative solutions to make it correct, without changing the structure of the program (just using directives or clauses) and never making use of critical. Explain why they make the execution correct.

The solutions that I propose are similar. is to put a #pragma omp critical or #pragma omp atomic before count++ to make sure that only one thread at the same time can increment the value of countmax.

## 7.datarace.c

1. Is this program executing correctly? If not, explain why it is not providing the correct result for one or the two variables (countmax and maxvalue)

Is not executing correctly because the executions don't show the correct result. It don't protect the two ifs inside the for at the same time, because it can be possible that a countmax++ is done, but then countmax = 1.

2. Write a correct way to synchronise the execution of implicit tasks (threads) for this program.

```
#include <stdio.h>
#include <omp.h>
/* Execute several times before answering the questions */
/* with ./3.datarace */
/* Q1: Is the program executing correctly? If not, explain */
/* why it is not providing the correct result for one */
/* or the two variables (countmax and maxvalue) */
/* Q2: Write a correct way to synchronize the execution */
/* of implicit tasks (threads) for this program. */

#define N 1 << 20
int vector[N]={0, 0, 0, 1, 2, 3, 4, 5, 6, 7, 15, 14, 13, 12, 11, 10, 9, 8, 15, 15};

int main()
{
    int i, maxvalue=0;
    int countmax = 0;

    omp_set_num_threads(8);
    #pragma omp parallel private(i) reduction(max: maxvalue)
    {
        int id = omp_get_thread_num();
        int howmany = omp_get_num_threads();

        for (i=id; i < N; i+=howmany) if (vector[i] > maxvalue) maxvalue = vector[i];
    }
    #pragma omp parallel private(i) reduction(+: countmax)
    {
        int id = omp_get_thread_num();
        int howmany = omp_get_num_threads();
        for (i=id; i < N; i+=howmany) if (vector[i]==maxvalue) countmax++;
    }

    if ((maxvalue==15) && (countmax==3))
        printf("Program executed correctly - maxvalue=%d found %d times\n", maxvalue, countmax);
    else printf("Sorry, something went wrong - maxvalue=%d found %d times\n", maxvalue, countmax);

    return 0;
}
```

## 8.barrier.c

1. Can you predict the sequence of printf in this program? Do threads exit from the #pragma omp barrier construct in any specific order?

I can predict that the 4 first messages will be the same thread with the same time to sleep, but the order can be changed because they don't have any order to execute it.

The next four messages will be exactly the same, because they have an order made when they go to sleep and always will wake up the thread with id = 0, then 1, then 2 and then 3.

The last 4 messages will be the same, but can change the order of the messages because the threads get out the barrier at the same time with no order.

## Day 2: explicit tasks

### 1.single.c

1. What is the nowait clause doing when associated to single?

Without the nowait, it executes 1 by 1 in order all the threads.

If we put nowait, the implicit barriers disappears and the threads don't have to wait to the other threads finish.

2. Then, can you explain why all threads contribute to the execution of the multiple instances of single? Why those instances appear to be executed in bursts?

The thread master creates 20 tasks and the threads will executes one by one (without repeats a task that another thread did by single clause), so they will do  $20/4 = 5$  tasks each.

It's like burst because when a thread prints the result, it has to wait 1s with the next line(sleep(1)). The task is less than 1 s to execute and they will execute the result (we will see 4 prints) and wait a second (it's the first burst). In the next second, we will see the next 4 executions, and it is repeating 5 times in total.

### 2.fibtasks.c

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

It is because there isn't any directive #pragma omp parallel and it will execute only one thread.

2. Modify the code so that tasks are executed in parallel and each iteration of the while loop is executed only once.

```
#pragma omp parallel
#pragma omp single
while (p != NULL) {
    printf("Thread %d creating task that will compute %d\n", omp_get_thread_num(), p->data);
    #pragma omp task firstprivate(p)
    processwork(p);
    p = p->next;
}
```

3. What is the firstprivate(p) clause doing? Comment it and execute again. What is happening with the execution? Why?

It makes the ini\_list p private for every thread. If we get it out, p is shared and when a thread executes  $p = p \rightarrow \text{next}$ , all the threads are working with the next element (it could be the next or more because some threads can do it at the same time).

### 3.taskloop.c

1. Which iterations of the loops are executed by each thread for each task grainsize or num tasks specified?

The grain size is 4 (all the threads that execute a for, will compute 4 iterations) and there will be 12 iterations (to compute a for there will be  $12/4 = 3$  threads), so 3 different threads will execute the first for.

In the second for, num\_taks = 4 and it means that the taskloop will make taks of  $12/4 = 3$  iterations by tasks. The other thread that didn't compute any iteration of the first loop will compute one of the tasks and the other 3 threads will execute one task each one.

In total, 3 threads will execute  $4 + 3 = 7$  iterations and one thread 3 iterations.

2. Change the value for grainsize and num tasks to 5. How many iterations is now each thread executing? How is the number of iterations decided in each case?

Now, the threads will compute 5 iterations + 7iterations in the first for. It is like this because the last computer doesn't have more iterations. For the second for, the number of tasks is 5 and  $12/5 = 2,4$ , so the minimum number of tasks is 5 so the number of iteration will be 2 iterations by tasks and 6 tasks in total.

one thread will execute the five iterations of the fist loop, the other the last iterations of the first loop and the last two threads will execute 3 tasks each one of the second for.

In total the iterations will be 5, 7,  $3 * 2 = 6$  and  $3 * 2 = 6$ .

3. Can grainsize and num tasks be used at the same time in the same loop? What is happening with the execution of tasks if the nogroup clause is uncommented in the first loop? Why?

The directive nogroup gets out the implicit taskgroup.

### 4.reduction.c

1. Complete the parallelisation of the program so that the correct value for variable sum is returned in each printf statement. Note: in each part of the 3 parts of the program, all tasks generated should potentially execute in parallel.

```
#pragma omp parallel
#pragma omp single
{
    #pragma omp taskloop grainsize(BS) reduction(+:sum)
    for (i=0; i< SIZE; i++)
        sum += X[i];

    printf("Value of sum after reduction in taskloop = %d\n", sum);
}

// Part III
#pragma omp parallel
#pragma omp single
{
    for (i=0; i< SIZE/2; i++)
        #pragma omp task firstprivate(i)
        #pragma omp critical
        sum += X[i];

    #pragma omp taskloop grainsize(BS) reduction(+:sum)
    for (i=SIZE/2; i< SIZE; i++)
        sum += X[i];

    printf("Value of sum after reduction in combined task and taskloop = %d\n", sum);
}
```



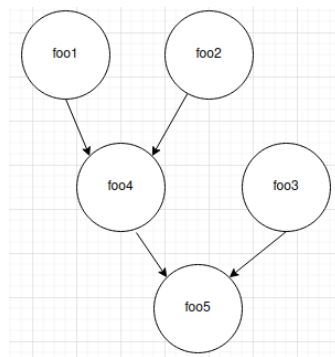
I add the `#pragma omp parallel` and `pragma omp single` directives in each part because with that, the code will execute in parallel one time, distributing the tasks created by the thread master by the other threads.

To minimise the overheads, I used a `reduction(+:sum)` in the third part to don't have problems with coherence like 2 thread read the value `sum = 5`, one thread do `sum += X[i] = 5+4 = 9` and the other do `sum += 5 + 5 = 10` and the value of `sum` after that will be 10 instead of `5+4+5 = 14`.

In the second part, I tried to correct the same problem as in the third part, but I couldn't put the `reduction(+:sum)`, so I used the `#pragma omp critical` but the cost is higher than the `reduction` clause.

## 5.synchtasks.c

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



2. Rewrite the program using only `taskwait` as task synchronisation mechanism (no depend clauses allowed), trying to achieve the same potential parallelism that was obtained when using `depend`.

```

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();

        #pragma omp taskwait
        printf("Creating task foo3\n");
        #pragma omp task
        foo3();
        printf("Creating task foo4\n");
        #pragma omp task
        foo4();

        #pragma omp taskwait
        printf("Creating task foo5\n");
        #pragma omp task
        foo5();
    }
}

```

I add two `taskwait` to do it, as we can see in the picture on the left that have my code.

In the first `taskwait`, we wait for the executions of the first level of the TDG (`foo1` and `foo2`) shown before. In the second `taskwait`, we wait for executions of the second second level of the TDG (`foo4` and `foo3`). Finally, it executes `foo5`, so all the dependencies are respected.

I put the `foo3` task in the second level because if I assume that all the functions have the same cost, we execute more faster with two or more processors than if we put the `foo3` in the first level.

3. Rewrite the program using only `taskgroup` as task synchronisation mechanism (no depend clauses allowed), again trying to achieve the same potential parallelism that was obtained when using `depend`.

```

#pragma omp single
{
#pragma omp taskgroup
{
    printf("Creating task foo1\n");
    #pragma omp task
    foo1();
    printf("Creating task foo2\n");
    #pragma omp task
    foo2();
}
#pragma omp taskgroup
{
    printf("Creating task foo3\n");
    #pragma omp task
    foo3();
    printf("Creating task foo4\n");
    #pragma omp task
    foo4();
}
}
printf("Creating task foo5\n");
#pragma omp task
foo5();
}

```

I add two taskgroup to do it, as we can see in the picture on the left that have my code.

In the first taskgroup, it waits for the executions of the first level of the TDG (foo1 and foo2) shown before. In the second taskgroup, we wait for the executions of the second level of the TDG (foo4 and foo3). Finally, it executes foo5, so all the dependencies are respected.

I put the foo3 task in the second level because if I assume that all the functions have the same cost, we can execute more faster with two or more processors than if we put the foo3 in the first level.

## Overheads

I executed the 4 programs and I earned the next results (the rows is the name of the program, the columns the number of threads. The overhead time is in seconds):

	1	4	8
atomic	4,688	4,839	5,763
critical	2,474	36,756	35,562
reduction	0,010	0,011	0,022
sumlocal	0,006	0,009	0,020

As we can see in the table, the overhead will exists because we will need synchronization time.

In the critical, the overhead is so hgher and I think It could be better if we don't use it because If we execute sequentially, It will be faster than the critical.

I think this is who has more time because if there are some threads, they won't execute 2 lines of the code simultaneously.

The atomic overhead is higher too because it takes 4,839 s with 4 threads and a bit less, 5,763 s with 8 threads, so to execute this code, the overhead will be higher than the execution time of the sequential code (1,797 s).

With the other two codes, the overheads costs are so low and it will be alle explote the parallelization.

We can see that in the critical, the overheads with 4 threads are higher than the 8 threads, it send a sync every time that will execute the line code in the critical clause, so it will do 100000000 times in boths cases, but for some reason that I don't know, it is faster with 8 than with 4. In the other program codes, the overhead is higher in 4 than in 8 threads. It could be because there are more tasks simultaneously waiting for the line code with the critical clause in 4 than in 8 threads.

The reason that there is a big difference in the overheads of the to first programs is because a synchronization time is sent in every iteration of the for (100000000), but in the other two is sent once by thread.

Finally I can try to aproximate the sincronyzation time. In critical or atomic, it will be  $\text{overhead}/100000000 * 1/10^6$  because in every time that is executed the line in clause critical or atomic. The next table shows it (the time is in  $\mu\text{s}$ ):

	1	4	8
atomic	0,0488	0,048	0,058
critical	0,025	00,37	0,356

The time for the synchronization of the other two programs is sent one time in every thread, because every thread compute the iterations and then rewrites the sum variable with the critical, so with one thread, we have the time, with 4 threads is the time/4 and with 8 threads is time/8 as we can see in the next table (the time is in  $\mu\text{s}$ ):

	1	4	8
reduction	10200	2725	2750
sumlocal	6000	2250	2487,5