

# Lab 2: Brief tutorial on OpenMP programming model

par4111

Adrià Cabeza, Xavier Lacasa

Departament d' Arquitectura de Computadors

March 17, 2019

2018 - 19 PRIMAVERA



**UNIVERSITAT POLITÈCNICA  
DE CATALUNYA  
BARCELONATECH**

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>OpenMP questionnaire</b>	<b>3</b>
2.1	Parallel regions . . . . .	3
2.1.1	1.hello.c . . . . .	3
2.1.2	2.hello.c . . . . .	3
2.1.3	3.how_many.c . . . . .	4
2.1.4	4.data_sharing.c . . . . .	5
2.2	Loop parallelism . . . . .	6
2.2.1	1.schedule.c . . . . .	6
2.2.2	2.nowait.c . . . . .	8
2.2.3	3.collapse.c . . . . .	9
2.3	Synchronization . . . . .	10
2.3.1	1.data_race.c . . . . .	10
2.3.2	2.barrier.c . . . . .	12
2.3.3	3.ordered.c . . . . .	12
2.4	Tasks . . . . .	13
2.4.1	1.single.c . . . . .	13
2.4.2	2.fibtask.c . . . . .	13
2.4.3	3.synctasks.c . . . . .	13
2.4.4	4.taskloop.c . . . . .	13
<b>3</b>	<b>Observing overheads</b>	<b>13</b>
3.1	Thread creation and termination . . . . .	13
3.2	Task creation and synchronization . . . . .	15
<b>4</b>	<b>Conclusions</b>	<b>17</b>

# 1 Introduction

In this laboratory assignment we will learn the basics of OpenMP extensions to the C programming language. At this moment we already know the hardware and tools we are going to use but this is the first approach to learn about the software needed to parallelize real applications.

This is the scope of the next two sessions of PAR laboratory: introducing the main components of the OpenMP programming model, from the simpler to the most complicated pragmas.

## 2 OpenMP questionnaire

### 2.1 Parallel regions

#### 2.1.1 1.hello.c

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

We see the message 24 times. This is due to the `#pragma omp parallel` call before `printf("Hello world! \n");`, which makes every available thread execute the `printf`. In Boada 1, they happen to be 24 threads, and so the message is printed 24 times.

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

By setting the number of threads available to 4, only 4 threads would execute the `printf("Hello world! \n");` line and so the message would be displayed only 4 times. We can accomplish this by adding `num_threads(4)` after `#pragma omp parallel`. Another way would be by using `export OMP_NUM_THREADS=4` before the execution of `1.hola`, so that only 4 threads are available.

#### 2.1.2 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?**

It is not correct, sometimes errors like `(2) Hello (1) Hello (1) world! (1) world!` occur. These happen because the variable `id` is declared before `#pragma omp parallel`, meaning all threads share it. Then what can happen is that for example thread 2 reads the variable `id`, prints the `"(2) Hello"` line, then thread 1 changes the variable, and when it is time to print the `"(2) world!"`, thread 2 prints `"(1) world!"` instead, because thread 1 assigned a new value before thread 2 was done.

To make it correct, we can add the `private(id)` tag to the `#pragma omp parallel num_threads(8)` line, so that every thread has its own local value of the variable

*id*. This way, when they assign their own id, they do not change the value other threads are reading from the var *id*.

**2. Are the lines always printed in the same order? Why the messages sometimes appear intermixed? (Execute several times in order to see this).** No, the lines are not always printed in the same order. This happens because threads do not get tasks assigned by #id order, so whichever thread gets the task earlier in that execution will print earlier.

Messages appear intermixed because even though we added the *private(id)* tag, threads do not wait for other threads to finish their execution before printing their lines (that would be sequential instead of parallel). This way, one thread might print *(id) Hello* and right then another thread's execution might start, printing *(id) Hello* again with a different *id*. After that, they both will finish with *(id) world!* and messages will have been intermixed. To avoid this, we should make the execution of the other threads stop when one thread starts to print until it finishes (we could use the *critical* construct), but that would make the execution sequential (with the added overhead of parallelization, even worse).

### 2.1.3 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?**

20 *Hello world ...* lines are printed on the screen. Each line is printed as many times as the number of threads available for the region of that line.

- Hello world from the first parallel (8)!: 8 times
- Hello world from the second parallel (2)!: 2 times
- Hello world from the second parallel (3)!: 3 times
- Hello world from the third parallel (4)!: 4 times
- Hello world from the fourth parallel (3)!: 3 times

**2. What does *omp\_get\_num\_threads* return when invoked outside and inside a parallel region?**

Outside a parallel region it returns the number of threads during the sequential execution, which is always just one.

Inside a parallel region however, it returns the number of threads available (busy or not) for that region:

1. For the first parallel region it returns all threads available for the program which are 8 (because of the *export OMP\_NUM\_THREADS=8* restriction).
2. The second parallel region is inside a for loop which completes 2 iterations with *i = 2 and 3*. Since the number of threads is set to *i* at each iteration by calling *omp\_set\_num\_threads(i)*, in the first iteration

`omp_get_num_threads()` returns 2 and in the second and final one it returns 3.

Note that `omp_set_num_threads()` overrides the number of threads set by `export OMP_NUM_THREADS=8`, which means that for the rest of the execution, unless otherwise specified, the number of available threads will be 3 (because the last call to `omp_set_num_threads(i)` was made in the second iteration, with  $i = 3$ .)

3. Before the third parallel region, the `#pragma omp parallel` directive is extended by adding `num_threads(4)`, which overrides the number of available threads just for this entire parallel region to 4. This way `omp_get_num_threads()` returns 4.
4. For this fourth parallel region the number of threads is not overridden, so it takes the value 3 (which is what `omp_get_num_threads()` returns) from the `omp_set_num_threads(3)` call in the for loop.

#### 2.1.4 4.data\_sharing.c

**1. 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)**

1. After first parallel (shared) x's value keeps varying depending on the execution. It should be 120 as intended by the code because there are 16 threads (as set by `omp_set_num_threads(16)`) and  $\sum_{i=0}^{15} i = 120$ , but its value after the execution is usually between 100 and 120. This variation occurs because all threads are sharing the same variable x, and to execute `x+=omp_get_thread_num();` they must first read the value of x and then write the result of the sum to it. When they do this, however, it can happen that for example thread 3 reads the value 10 in x, then thread 1 also reads the value 10, after that thread 3 updates x with  $10+3=13$  and thread 1, that has the old value of x, finally updates x with the value it knows:  $10+1=11$ . And so x instead of being  $10+3+1=14$ , is 11 because thread 1 did not notice thread 3's update of x. This happens quite randomly depending on the order of tasks assigned and threads occupation, so the final value of x keeps changing execution after execution.
2. After second parallel (private) x's value is always 5. This is expected as x is assigned the value of 5 right before the second parallel region. Then at the time of declaration of the parallel region, x is declared private (`#pragma omp parallel private(x)`), which means that each thread will modify a local copy (to each thread) of x, but will in no case modify the shared value of the variable (private(x) also makes the local copies of x start without an initialised value instead of the value of x previous to the parallel region (5)). This way threads can't modify the value other threads have of x, but

this also means that after the parallel region, unless strictly specified, the shared value of `x` is still the same as before executing the region, because none of the threads has written the value of their local copy of `x` to the shared variable `x`. This is why in the end of the second parallel region `x` is always equal to 5.

3. After third parallel (`firstprivate`) `x`'s value is also always 5. This is expected as well, since `firstprivate(x)` has the same behaviour as `private(x)` with the exception that `firstprivate(x)` initializes each local copy of `x` for each thread with the value that `x` had before the start of the parallel region. In this case, that value is 5. Still, this does not affect the value of `x` after the execution of the parallel region, but it changes the behaviour inside.
4. After fourth parallel (`reduction`) `x`'s value is always 125. This is correct because `pragma omp parallel reduction(+:x)` is supposed to make local copies of `x` for each thread, just like `private`, but initializing them to the neutral element (for addition, this would be 0). Then each thread executes its instructions inside the parallel region and after that all local copies' values are added together with the initial value of `x` (which was 5). This is therefore equivalent to computing the following (remember there are 16 threads):

$$5 + \sum_{i=0}^{15} i = 125 \quad (1)$$

And 125 is indeed what is printed after each execution for the fourth parallel region, so it is safe to assume it works as intended.

## 2.2 Loop parallelism

### 2.2.1 1.schedule.c

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

- **Schedule(static)**: There is an equal distribution of iterations for each thread. As we can see in the outputted execution of the loop, each of the four threads gets a chunk of 3 iterations, having in total of 12 iterations.

```
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: (3) gets iteration 9
Loop 1: (3) gets iteration 10
```

```
Loop 1: (3) gets iteration 11
Loop 1: (2) gets iteration 6
Loop 1: (2) gets iteration 7
Loop 1: (2) gets iteration 8
```

- **Schedule(static,2):** There is also an equal distribution of iterations for each thread but in this case each thread is executing only 2 consecutive iterations of the for loop each time according to the thread order. As we can see in the outputted execution of the loop, the iterations are given in chunks of size 2 and in order: thread 0 takes iteration 1 and 2, thread 1 takes iteration 2 and 3,...

```
Going to distribute 12 iterations with schedule(static, 2)
...
Loop 2: (3) gets iteration 6
Loop 2: (0) gets iteration 0
Loop 2: (0) gets iteration 1
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: (3) gets iteration 7
Loop 2: (0) gets iteration 8
Loop 2: (0) gets iteration 9
Loop 2: (2) gets iteration 4
Loop 2: (2) gets iteration 5
```

- **Schedule(dynamic,2):** Each thread is executing 2 consecutive iterations of the loop each time. However, the order is guided by the first thread that is free. As we can see in the outputted execution of the loop, the iterations are given also in chunks of size 2, but here they are not sorted, the assignment is given depending on the availability of the thread.

```
Going to distribute 12 iterations with schedule(dynamic, 2)
...
Loop 3: (1) gets iteration 0
Loop 3: (1) gets iteration 1
Loop 3: (1) gets iteration 8
Loop 3: (1) gets iteration 9
Loop 3: (1) gets iteration 10
Loop 3: (1) gets iteration 11
Loop 3: (0) gets iteration 6
Loop 3: (0) gets iteration 7
Loop 3: (3) gets iteration 4
```

```
Loop 3: (3) gets iteration 5
Loop 3: (2) gets iteration 2
Loop 3: (2) gets iteration 3
```

- **Schedule(guided,2)**: In this case, the size of each chunk is proportional to the number of unassigned iterations divided by the number of the threads. Therefore the size of the chunks decreases. The order is also given by the first who is free and the chunk-size specified (in this case 2) represents the minimum size of a chunk.

```
Going to distribute 12 iterations with schedule(guided, 2)
...
Loop 4: (3) gets iteration 0
Loop 4: (3) gets iteration 1
Loop 4: (3) gets iteration 8
Loop 4: (0) gets iteration 2
Loop 4: (0) gets iteration 3
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 4
Loop 4: (2) gets iteration 5
Loop 4: (3) gets iteration 9
```

### 2.2.2 2.nowait.c

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

The nowait clause removes the synchronization time needed between threads. If we remove the clause, the threads don't wait the others to finish their parallel region executions so they continue executing code. That means that without the nowait clause the sequence of printf is printed randomly by both for loops (but respecting the code sequence in a thread). Let's see an example of the outputted execution (we changed the number of iterations of the code in order to be able to check our hypothesis):

```
Loop 1: thread (0) gets iteration 0
Loop 1: thread (4) gets iteration 1
Loop 1: thread (7) gets iteration 2
Loop 2: thread (6) gets iteration 6
Loop 2: thread (5) gets iteration 7
Loop 1: thread (2) gets iteration 3
Loop 2: thread (3) gets iteration 4
```



```
Loop 2: thread (1) gets iteration 5
```

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

If we did not have the nowait clause we would see first the sequence of printf done by the first loop and then the sequence done by the second loop: In order to check if our hypothesis is correct let's see the outputed execution:

```
Loop 1: thread (1) gets iteration 1
Loop 1: thread (0) gets iteration 0
Loop 2: thread (0) gets iteration 3
Loop 2: thread (1) gets iteration 2
```

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

Even though we kept the nowait clause we would still get a correct printf sequence: first the first loop and then the second one. But why? The static clause creates a static scheduling type, that is, the iterations are grouped into chunks and each thread executes the threads in circular order. So there is implicitly an order into the static schedule.

If we check the ouputed execution we can see that out hypothesis is true.

```
Loop 1: thread (0) gets iteration 0
Loop 1: thread (3) gets iteration 3
Loop 1: thread (2) gets iteration 2
Loop 1: thread (1) gets iteration 1
Loop 2: thread (0) gets iteration 4
Loop 2: thread (2) gets iteration 6
Loop 2: thread (1) gets iteration 5
Loop 2: thread (3) gets iteration 7
```

### 2.2.3 3.collapse.c

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

The collapse clause will parallelize more than one level of the loop. So each thread will execute consecutively a set of consecutive iterations that corresponds to the sequential iterations of the loops combined. To sum up, the iterations are in groups like it was only one loop.

```
(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)
(3) Iter (2 0)
(3) Iter (2 1)
(3) Iter (2 2)
...
```

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

If we delete the clause the execution is not correct because only the `i` is protected and the `j` is shared. In order to solve it we could make private the variable `j` to avoid having different iterations of the loop overriding the variable `j`.

```
int main()
{
    int i,j;

    omp_set_num_threads(8);
    #pragma omp parallel for private(j)
    for (i=0; i < N; i++) {
        for (j=0; j < N; j++) {
            int id=omp_get_thread_num();
            printf("(%d) Iter (%d %d)\n",id,i,j);
        }
    }

    return 0;
}
```

## 2.3 Synchronization

### 2.3.1 1.datarace.c

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

No. There is a data race.

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

Alternative 1

```
int main()
{
```

```

int i, x=0;
omp_set_num_threads(8);
#pragma omp parallel private(i)
{
    int id = omp_get_thread_num();
    for (i=id; i < N;i+=8) {
        #pragma omp critical
        x++;
    }
}

if (x==N) printf("Congratulations!, program executed correctly
               (x = %d)\n", x);
else printf("Sorry, something went wrong, value of x = %d\n",
           x);

return 0;
}

```

This version protects the operation `x++`, where we had the data race. We created a critical region that only one thread can execute at once.

## Alternative 2

```

int main()
{
    int i, x=0;
    omp_set_num_threads(8);
    #pragma omp parallel private(i)
    {
        int id = omp_get_thread_num();
        for (i=id; i < N;i+=8) {
            #pragma omp atomic
            x++;
        }
    }

    if (x==N) printf("Congratulations!, program executed correctly
                   (x = %d)\n", x);
    else printf("Sorry, something went wrong, value of x = %d\n",
               x);

    return 0;
}

```

```
}
```

Like in the previous version, here we are also protecting the operation `x++`, in this case we made the operation `x++` indivisible using the `atomic` pragma directive, forcing to make the read and write operations on one thread before allowing another thread to access the variable.

### 2.3.2 2.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. A barrier defines a point in the code where all the active threads will stop until all threads have arrived at that point. So we can suppose that we will get all the last messages about being awake at the same time. Also, we can say that all the messages about going to sleep are going to be printed before the ones about being awake, and the sequence of threads being awake can be predicted (because of the seconds that each thread is going to sleep). Moreover, the threads don't exit the barrier in any specific order.

### 2.3.3 3.ordered.c

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

The `ordered` clause works like this: different threads execute concurrently until they encounter the ordered region, which is executed sequentially in the same order as it would get executed in a serial loop. So, the order in which the *Outside* and *Inside* messages are printed is because of the fact that the inside message is inside the ordered region (that is executed sequentially) and the outside message is outside that region where all the different threads are executing concurrently.

```
Before ordered - (0) gets iteration 0
Inside ordered - (0) gets iteration 0
Before ordered - (4) gets iteration 1
Inside ordered - (4) gets iteration 1
Before ordered - (4) gets iteration 3
Before ordered - (0) gets iteration 2
Inside ordered - (0) gets iteration 2
Before ordered - (2) gets iteration 5
Before ordered - (3) gets iteration 4
Before ordered - (5) gets iteration 6
Before ordered - (0) gets iteration 7
Inside ordered - (4) gets iteration 3
Before ordered - (6) gets iteration 10
Before ordered - (7) gets iteration 9
Before ordered - (1) gets iteration 8
Inside ordered - (3) gets iteration 4
```

```
Inside ordered - (2) gets iteration 5
Before ordered - (2) gets iteration 13
...
```

**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?**

Changing from `schedule dynamic` to `schedule(dynamic,2)`

## 2.4 Tasks

### 2.4.1 1.single.c

**1. 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?**

### 2.4.2 2.fibtask.c

**1. Why all tasks are created and executed by the same thread? In other words, why the program is not executing in parallel? 2. Modify the code so that the program correctly executes in parallel, returning the same answer that the sequential execution would return.**

### 2.4.3 3.synctasks.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)**

### 2.4.4 4.taskloop.c

**1. 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. 2. What does occur if the `nogroup` clause in the first `taskloop` is uncommented?**

## 3 Observing overheads

In this last section we will be observing the main results obtained in terms of overheads for *parallel,task* and the different synchronisation mechanisms. We will discuss our conclusions and support them with tables and plots.

### 3.1 Thread creation and termination

In this part we will learn about the thread creation and termination overhead and how it varies. We will be using the *pi\_omp\_parallel.c* file, which it computes the difference of time between the sequential execution and the parallel execution when using a certain number of threads and finally uses that information to print the difference in time (in microseconds), which is the overhead introduced by the parallel construct.

In the results of the execution we can see that the order of magnitude of the overhead generated is in microseconds, we can see it clearly in figure 2, when the execution starts, the first message informs about it.

Observing the results on the execution of *pi\_omp\_parallel* with just 1 iteration and a maximum of 24 threads, see in figure 1, we can confirm that the time of the overhead is not constant, it increases linearly when the number of threads increase. Moreover, the overhead per thread tends to decrease because when the number of threads increases every thread has to do less calculations.

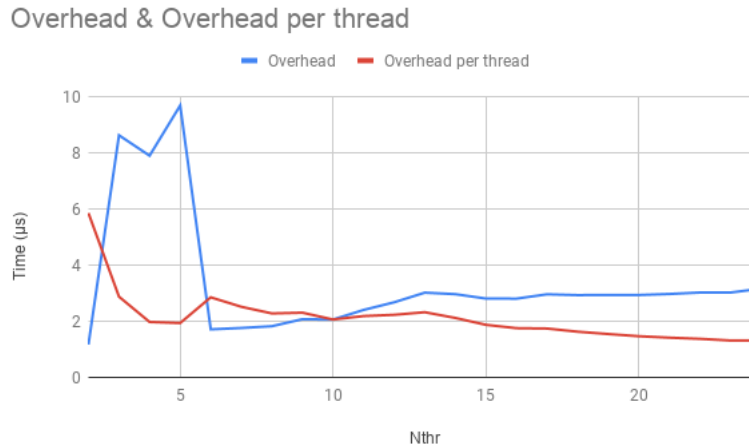


Figure 1: Plot of the overhead and overhead per thread depending on the number of threads

```

par4111@boada-1:~/lab2/overheads$ qsub -l execution submit-omp.sh
    pi_omp_parallel 1 24
Your job 97651 ("submit-omp.sh") has been submitted

par4111@boada-1:~/lab2/overheads$ more pi_omp_parallel_1_24.txt
All overheads expressed in microseconds
Nthr Overhead Overhead per thread
2 1.1726 0.5863
3 0.8639 0.2880
4 0.7910 0.1978
5 0.9701 0.1940
6 1.7148 0.2858
7 1.7640 0.2520
8 1.8267 0.2283
9 2.0780 0.2309
10 2.0664 0.2066
11 2.4082 0.2189
12 2.6808 0.2234
13 3.0237 0.2326
14 2.9701 0.2122
15 2.8141 0.1876
16 2.8110 0.1757
17 2.9655 0.1744
18 2.9356 0.1631
19 2.9420 0.1548
20 2.9402 0.1470
21 2.9768 0.1418
22 3.0285 0.1377
23 3.0284 0.1317
24 3.1657 0.1319

```

Figure 2: Execution of `pi_omp_parallel.c` with 1 iteration and a maximum of 24 threads

### 3.2 Task creation and synchronization

In this part we will learn about the overhead related to the creation of **tasks** and their synchronization at **taskwait** and how it varies. We will be using the `pi_omp_tasks.c` file, which it computes the difference of time between the sequential execution and the version that creates the tasks, when using a certain number of tasks and finally uses that information to print the difference in time (in microseconds), which is the overhead introduced by the **task** and **taskwait** constructs.

In the results of the execution we can see that the order of magnitude of the

overhead generated is in microseconds, we can see it clearly in figure 4, when the execution starts, the first message informs about it.

Observing the results on the execution of *pi\_omp\_tasks* with 10 iterations and 1 thread, see figure 3, we can confirm that the time of the overhead does not depend on the number of tasks. The overhead will cost the same independently of the number of tasks, so it is constant. This affirmation is defended mainly by the tiny variation of time, probably given by the instability of the machine.

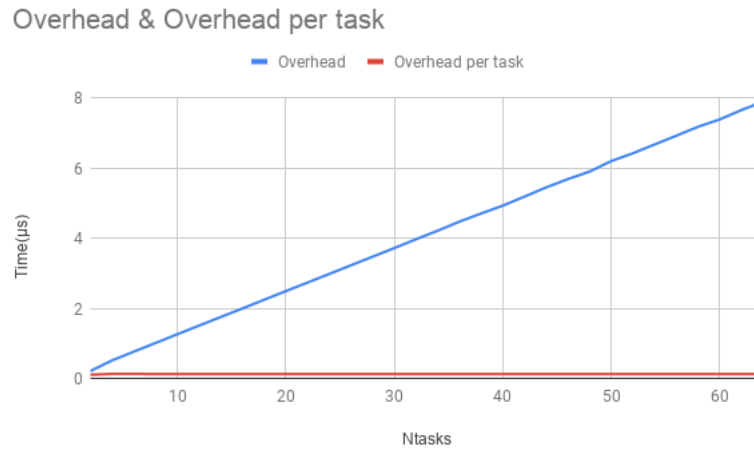


Figure 3: Plot of the overhead and overhead per task depending on the number of tasks



```

par4111@boada-1:~/lab2/overheads$ qsub -l execution submit-omp.sh
    pi_omp_tasks 10 1
Your job 97659 ("submit-omp.sh") has been submitted
par4111@boada-1:~/lab2/overheads$ more pi_omp_tasks_10_1.txt
All overheads expressed in microseconds
Ntasks  Overhead Overhead per task
2 0.2105      0.1052
4 0.5151      0.1288
6 0.7635      0.1273
8 1.0115      0.1264
10 1.2594     0.1259
12 1.5057     0.1255
14 1.7508     0.1251
16 1.9950     0.1247
18 2.2435     0.1246
20 2.4908     0.1245
22 2.7343     0.1243
24 2.9798     0.1242
26 3.2272     0.1241
28 3.4727     0.1240
30 3.7192     0.1240
32 3.9713     0.1241
34 4.2150     0.1240
36 4.4738     0.1243
38 4.7067     0.1239
40 4.9300     0.1232
42 5.1890     0.1235
44 5.4513     0.1239
46 5.6880     0.1237
48 5.9030     0.1230
50 6.2037     0.1241
52 6.4264     0.1236
54 6.6773     0.1237
56 6.9273     0.1237
58 7.1824     0.1238
60 7.3931     0.1232
62 7.6592     0.1235
64 7.8985     0.1234

```

Figure 4: Execution of pi\_omp\_tasks.c with 10 iteration and 1 thread

## 4 Conclusions

Jo borraria lo de conlusions, fa tot el pal escriure-ho