PAR Laboratory Assignment
LAB 2: Brief Tutorial on OpenMP Programming Model

GROUP 23 - PAR1405

Raimon Mercé Gotsens

Meir Carlos Mouyal Amselem

March 21st, 2019
2nd Semester

OpenMP questionnaire

A) Parallel regions

1.hello.c

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

   After executing the program ***1.hello***, the ***"Hello world!"*** message was printed in the console 24 times. This happens because the program is executed in parallel and there is no clause to limit the number of threads to execute the parallel region.

   Therefore, since the node architecture of **boada-1** contains 2 **sockets** per **node**, with 6 **cores** per **socket** and 2 threads per **core**:

   Each **socket** contains 6 **cores** x 2 threads/**core** = 12 threads.

   And 2 **sockets** x 12 threads/**socket** = 24 threads in total.

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

   We made the program to print 4 times the ***"Hello world!"*** message by executing the program with:

   #OMP_NUM_THREADS=4 ./1.hello

   This way we set the environment variable **nthreads-var** ICV = 4, which is the number of threads to use for parallel regions.

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?*

   The execution of the program is not correct. This is due to that the variable **id** is being shared between threads, and it's being overwritten every time a thread accesses it.

   In order to correct this we edited the line of code **#pragma omp parallel num_threads(8)** by adding the clause **private(id)**.

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).*

The lines are not always printed in the same order, in fact they sometimes appear inter-mixed because we are currently not ensuring in any way the order in which threads will execute the code.

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?*

   In total, 22 lines of "**Hello World ...**" are printed on the screen.

   This is caused by the number of threads that execute each line of code in the program, which is explained in the next question.

2. *What does* **omp_get_num_threads** *return when invoked outside and inside a parallel region?*

   Outside:

   It will always be 1 thread because is outside of the parallel region

   - Starting → 1 thread
   - Outside → 1 thread
   - Finishing → 1 thread

   Inside:

   - First → 8 threads, since is the number of threads we specified for the environment variable OMP_NUM_THREADS.
   - Second → 2 and 3 threads, due to that the number of threads is changed during execution with the line of code: **omp_set_num_threads(i);** therefore for the 1st iteration being 2 the number of threads and 3 for the second.
   - Third → 6 threads, due to that for this parallel region of the code, the number of threads to use is specified to be 6 with: **num_threads(6)**
   - Fourth → 3 threads, because in the line of code of the for loop where **omp_set_num_threads(i);** was executed, it lastly changed the environment variable OMP_NUM_THREADS to equal 3 temporarily for subsequent parallel regions. Therefore modifying the number of threads to use by default on parallel regions, which we already set before executing this program, for the remaining of its code.

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)*

   Values of variable **x**:

   - Shared → 120 (varies)
   - Private → 5 (constant)
   - Firstprivate → 5 (constant)
   - Reduction → 125 (constant)

   The only value we can't expect is the value for the parallel region of the data-sharing attribute **shared**, since due to that **x** is a shared value it's possible that data-races happen, which is what sometimes occurs being the variable **x** smaller than 120.

B) Loop parallelism

1.schedule.c

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

   Beginning from the premise that 4 threads will execute the 4 **for** loops in parallel with different **schedule** properties, the distribution is as follows:

- **Loop 1 — schedule(static)**:
  - Thread 0 → executes iterations 0, 1 and 2.
  - Thread 1 → executes iterations 3, 4 and 5.
  - Thread 2 → executes iterations 6, 7 and 8.
  - Thread 3 → executes iterations 9, 10 and 11.

  This is due to that the **static** property divides the total number of iterations for every thread, therefore giving in this case: the first three iterations to the Thread 0, the next three ones to the Thread 1, and so on.

- **Loop 2 — schedule(static, 2)**:
  - Thread 0 → executes iterations 0, 1, 8 and 9.
  - Thread 1 → executes iterations 2, 3, 10 and 11.
  - Thread 2 → executes iterations 4 and 5.

        - Thread 3 → executes iterations 6 and 7.

This is due to that the **static** property with a **chunk** value, beginning from Thread 0, it assigns the amount of the **chunk's** value iterations, in this case 2, to every thread in the team, in order, until there are no more iterations left to distribute. As shown in this example, Thread 0 and Thread 1 each have 2 more iterations than the other Threads due to that after a complete round of distributing iterations, there were still more iterations to assign, therefore starting a second round of distributions.

- **Loop 3** — schedule(dynamic, 2):
    - Thread 0 → executes iterations 6 and 7.
    - Thread 1 → executes iterations 4 and 5.
    - Thread 2 → executes iterations 2 and 3.
    - Thread 3 → executes iterations 0, 1, 8, 9, 10 and 11.

This is due to that the **dynamic** property with a **chunk** value assigns iterations like the **static** property with a **chunk**, but without following an order. As a consequence, it works by assigning the amount of the **chunk's** value iterations, in this case 2, to the first free thread.

Respectively, what can happen in some cases is that a specific thread always ends fast enough to be assigned the next group of iterations, resulting to be the only thread executing the program. It's also important to notice that in the **dynamic** property is impossible to predict what will be executed by each thread.

- **Loop 4** — schedule(guided, 2):
    - Thread 0 → executes iterations 2 and 3.
    - Thread 1 → executes iterations 4 and 5.
    - Thread 2 → executes iterations 6 and 7.
    - Thread 3 → executes iterations 0, 1, 8, 9, 10 and 11.

This is due to that the **guided** property with a **chunk** value, the size of each chunk is proportional to the number of unassigned iterations divided by the number of threads in the team with the restriction that the chunks do not contain fewer than the chunk's value iterations, in this case 2.

It's really curious how specifically for this case the iterations distribution is really similar to that in *Loop 3*, however this was pure coincidence, since as when using the dynamic property, it's impossible to predict what will be executed by each thread.

2.nowait.c

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

   There are 4 threads that can execute this program in parallel, and since each **for** loop has the properties **schedule(dynamic, 1) nowait**, then both of the loops can be executed at the same time, therefore being a possible sequence:

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

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

   Then *Loop* 2 will have to wait until *Loop 1* ends its execution. As a consequence only 2 of the 4 threads are used, specifically threads 0 and 1, due to that in *Loop 1* threads 0 and 1 are assigned each to one iteration, while threads 3 and 4 don't have any work to do. Once finished, *Loop* 2 begins, and since threads 0 and 1 are free, then they are assigned each to one iteration again.

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

   Then since its **static**, even if both **for** loops keep the **nowait** clause, *Loop* 2 needs to wait until *Loop 1* is finished. This happens because **static** always assigns the first threads, which in this case are already occupied by *Loop 1*.

3.collapse.c

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

   The **collapse** clause specifies how many loops are associated with the loop construct, therefore the iterations of all associated loops are collapsed into one larger iteration space. That's why:

   - Thread 0 will always execute → iterations (0,0), (0,1), (0,2) and (0,3).
   - Thread 1 will always execute → iterations (0,4), (1,0) and (1,1).

- Thread 2 will always execute → iterations (1,2), (1,3) and (1,4).
- Thread 3 will always execute → iterations (2,0), (2,1) and (2,2).
- Thread 4 will always execute → iterations (2,3), (2,4) and (3,0).
- Thread 5 will always execute → iterations (3,1), (3,2) and (3,3).
- Thread 6 will always execute → iterations (3,4), (4,0) and (4,1).
- Thread 7 will always execute → iterations (4,2), (4,3) and (4,4).

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

   No, since then the only loop that is associated with the loop construct is the one that immediately follows the directive and therefore the variable **j** of the second loop is not being privatized as a consequence affecting the behaviour of threads with the inner loop. The clause **private(j)** should be added.

C) Synchronization

1.datarace.c

1. *Is the program always executing correctly?*

   No, it prints on the console the message *"Sorry, something went wrong…"*. After analyzing the code, we know that this happens because at the end of the execution of the **for** loop, the variable **x** is not equal to variable **N**. And this takes place because **x** is public and data-races occur between threads.

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

   1. Inserting the line of code: **#pragma omp critical** before the **x++**.
   This alternative is correct because the **critical** clause provides a region of code where only one thread will execute at a given time.

   2. Inserting the line of code: **#pragma omp atomic** before the **x++**.
   This alternative is correct because the **atomic** clause ensures the indivisible execution of read and write operations.

2.barrier.c

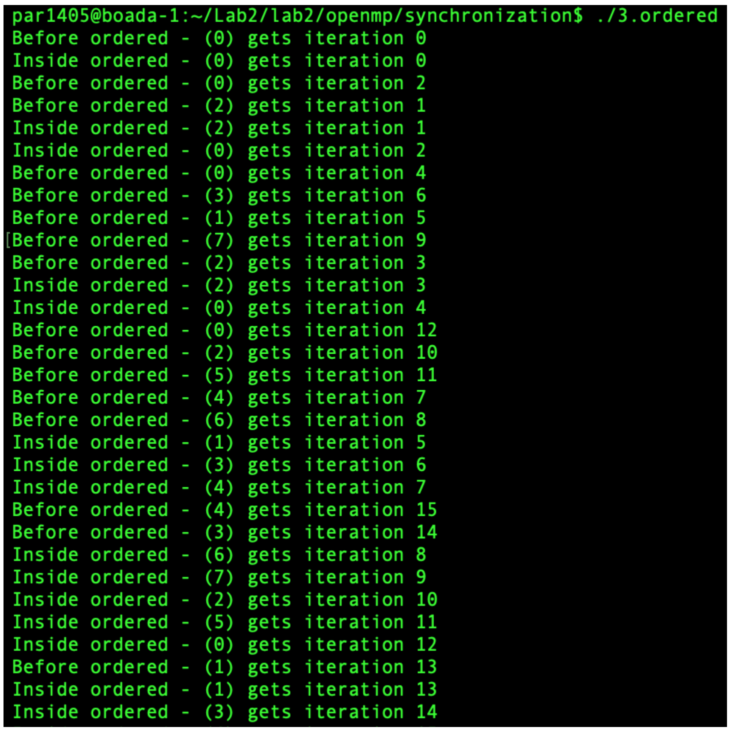
1. *Can you predict the sequence of messages in this program? Do threads exit from the barrier in any specific order?*

No, it’s impossible to predict the sequence of messages of this program. This is due to that in the first message of “... **going to sleep** ...”, the order is completely random because of the assignment of threads, and the last message of “... **We are all awake!**” is also random, because all 4 threads wait for each other at the **barrier** so then its impossible for us to know the order in which threads will exit from the barrier.

3.ordered.c

1. *Can you explain the order in which the ~~“Outside”~~/“Before” and “Inside” messages are printed?*

   The order in which both messages will be printed as a group we can’t know it, since it’s impossible for us to know the order in which threads will print each message due to parallelism.
   However, analyzing them separately, for the “**Before**” messages we still can’t, but for the “**Inside**” messages we can, since everything that appears after line 23, where its written: **#pragma omp ordered**, makes the code of the region to appear in the order of the loop iterations, as can be seen on Figure 1.



```
par1405@boada-1:~/Lab2/lab2/openmp/synchronization$ ./3.ordered
Before ordered - (0) gets iteration 0
Inside ordered - (0) gets iteration 0
Before ordered - (0) gets iteration 2
Before ordered - (2) gets iteration 1
Inside ordered - (2) gets iteration 1
Inside ordered - (0) gets iteration 2
Before ordered - (0) gets iteration 4
Before ordered - (3) gets iteration 6
Before ordered - (1) gets iteration 5
Before ordered - (7) gets iteration 9
Before ordered - (2) gets iteration 3
Inside ordered - (2) gets iteration 3
Inside ordered - (0) gets iteration 4
Before ordered - (0) gets iteration 12
Before ordered - (2) gets iteration 10
Before ordered - (5) gets iteration 11
Before ordered - (4) gets iteration 7
Before ordered - (6) gets iteration 8
Inside ordered - (1) gets iteration 5
Inside ordered - (3) gets iteration 6
Inside ordered - (4) gets iteration 7
Before ordered - (4) gets iteration 15
Before ordered - (3) gets iteration 14
Inside ordered - (6) gets iteration 8
Inside ordered - (7) gets iteration 9
Inside ordered - (2) gets iteration 10
Inside ordered - (5) gets iteration 11
Inside ordered - (0) gets iteration 12
Before ordered - (1) gets iteration 13
Inside ordered - (1) gets iteration 13
Inside ordered - (3) gets iteration 14
```

Figure 1 - ./3.ordered

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 assigning to that thread chunks of 2 or more iterations. This can be achieved by changing the actual schedule from the code to for example **schedule(dynamic, 2)**.

D) Tasks

1.single.c

1. *Can you explain why all threads contribute to the execution of instances of the single work-sharing construct? Why are those instances appear to be executed in bursts?*

   All threads contribute to the execution of instances of the single work-sharing construct because before the **for** loop, the **#pragma omp parallel**, forces all 4 threads to execute the entire loop, nonetheless, afterwards the line **#pragma omp single nowait** is executed, in which the **single** clause only allows 1 thread of the threads of the team to execute the associated region.

   Then, what makes the instances appear to be executed in bursts is because of the **nowait** clause, since all 4 threads execute a different iteration, obligated by the **single** clause, and then they all together sleep during 1 second.

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?*

   Because there is no team of threads when **#pragma omp task** is executed, therefore there is no parallelism.

2. *Modify the code so that the program correctly executes in parallel, returning the same answer that the sequential execution would return.*

   We managed to parallelize the code by inserting the following lines on top of the first while loop:

   ```
   #pragma omp parallel
   #pragma omp single
   ```

   And also by adding the clause **firstprivate(p)** to the **#pragma omp task** line.

3.synchtasks.c

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

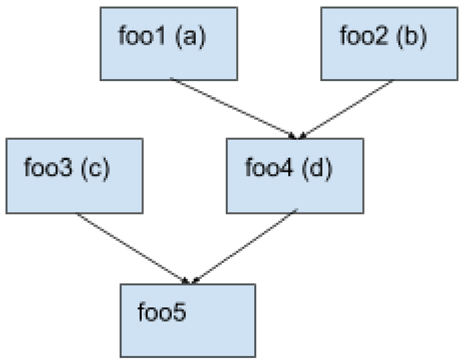
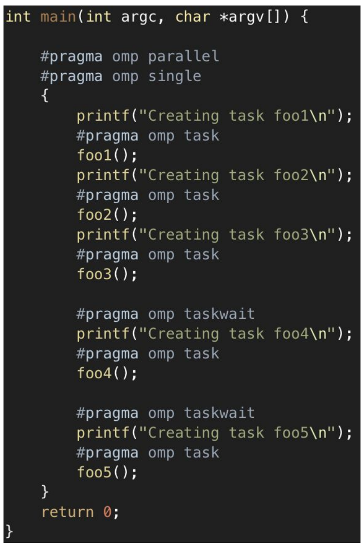


Figure 2 - task dependence graph

2. *Rewrite the program using only* ***taskwait*** *as task synchronisation mechanism (no* ***depend*** *clauses allowed)*



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

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

        #pragma omp taskwait
        printf("Creating task foo5\n");
        #pragma omp task
        foo5();
    }
    return 0;
}
```

Figure 3 -synchtasks.c with taskwait

4.taskloop.c

3. *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.*

On taskloop, the grainize property controls how many logical loop iterations are assigned to each created task. This value is greater or equal to the **grain-size** value passed and also smaller than 2 times this value. For this example, in *Loop 1*, the **grain-size** value is 5, and since it consists of 12 iterations, then there are only 2 tasks with 6 iterations each.

On the contrary, the **num_tasks** property creates as many tasks as the value of **num-tasks** given and the number of logical loop iterations. As in the example of *Loop* 2, the **num-tasks** is 5, creates a minimum of 5 tasks which are distributed in a team of 4 threads.

4. *What does occur if the **nogroup** clause in the first **taskloop** is uncommented?*

   The **nogroup** clause acts as the **nowait** clause, by allowing the parallel execution of the 2 loops. This happens because **taskloop** executes as if it was enclosed in a **taskgroup** and by uncommenting the **nogroup** clause then no implicit **taskgroup** is created.

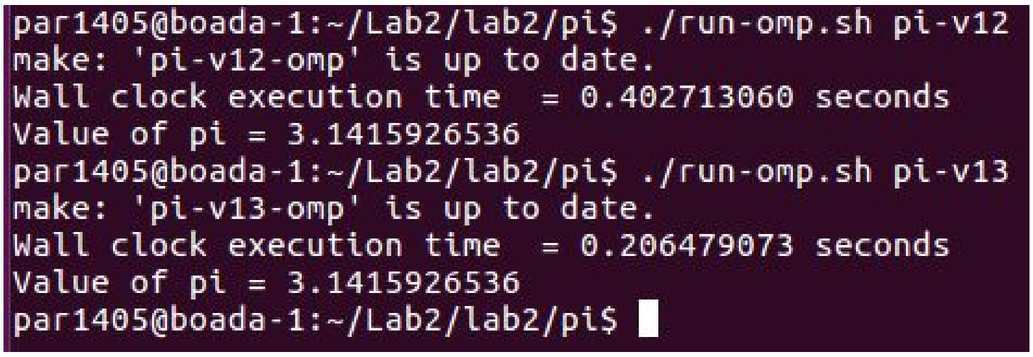
Observing overheads

Parallelisation with OpenMP

2.1.1 The single work–sharing construct

1. No, because on **single**, all threads wait until the one executing the code ends, so we aren't exploiting any parallelism.

2. When we use **nowait**, the thread executing the single part will stop, but the other ones will continue and execute the second one exploiting parallelism. As we see in the Figure 4, **pi_v13** is almost double faster than **pi_v12**.



```
par1405@boada-1:~/Lab2/lab2/pi$ ./run-omp.sh pi-v12
make: 'pi-v12-omp' is up to date.
Wall clock execution time  = 0.402713060 seconds
Value of pi = 3.1415926536
par1405@boada-1:~/Lab2/lab2/pi$ ./run-omp.sh pi-v13
make: 'pi-v13-omp' is up to date.
Wall clock execution time  = 0.206479073 seconds
Value of pi = 3.1415926536
par1405@boada-1:~/Lab2/lab2/pi$
```

Figure 4 - Execution of pi-v12 and pi-v13

2.1.2 Tasking execution model

1. Because nothing limits the number of threads that can execute each part, so the four threads execute each one.

2. Using **shared** and declaring the vector **sumalocal[ ]**, so the result of each execution is stored on **sumalocal[thread number]**, and at the end sum all of the values in the vector.

2.3 Observing overheads

2.3.1 Thread creation and termination

As Figure 5 shows, at Thread 4 the overhead value start growing up until iteration 20 more or less, where the overhead takes a constant value of 3,2 approximately. On the other hand, the Overhead per Threads are on its highest point on the first Thread, and then it starts going down, except for a peak on the 5th Thread.

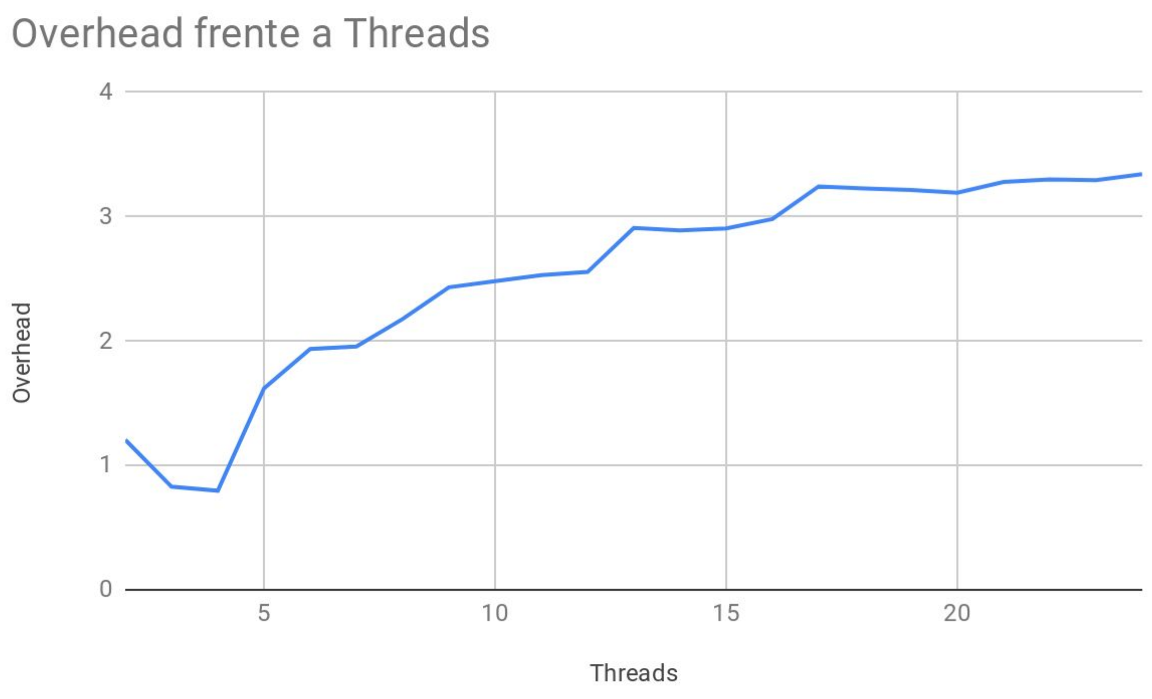


Figure 5 - Overheads/ Number of Threads of pi_omp_parallel

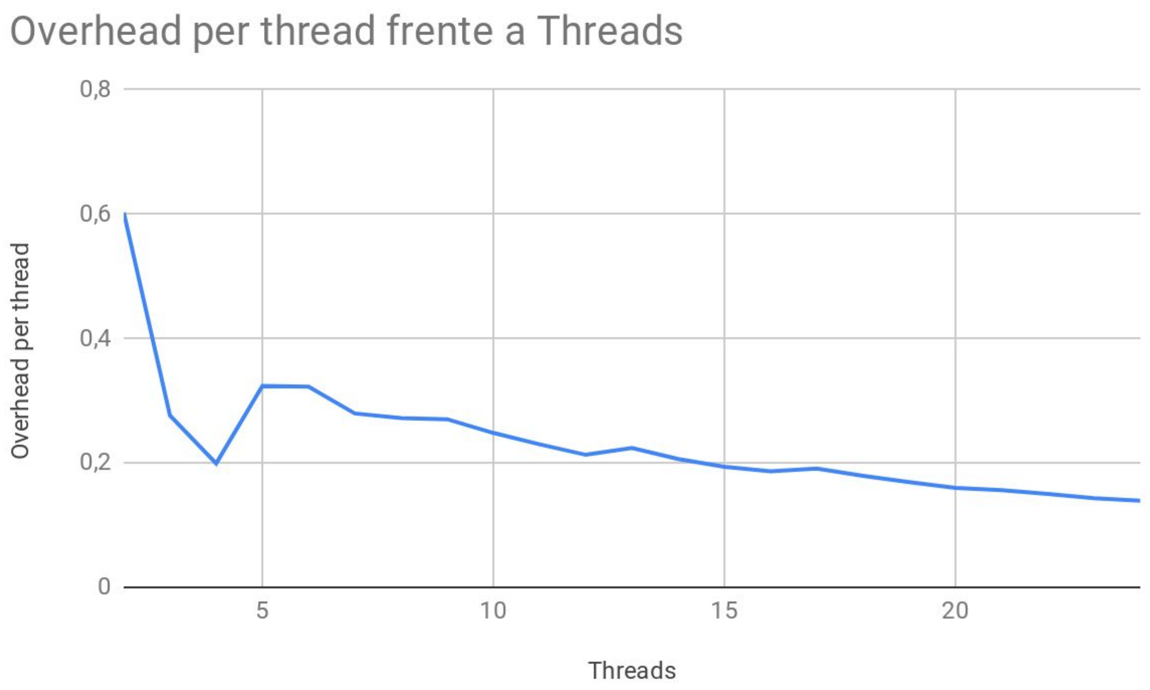


Figure 6 - Overheads per Thread/ Number of Threads of pi_omp_parallel

2.3.2 Task creation and synchronization

The difference between the sequential execution and the version that creates tasks, as we see in the nest figures increase on each Task.

As we can see on Figure 7, from the first Task until the last we can appreciate, the Overhead value increase as a constant, except from a little insignificant peaks.

The Overhead per Task value is really stable during each task, as we can see on Figure 8, gets stuck at value 1.15.

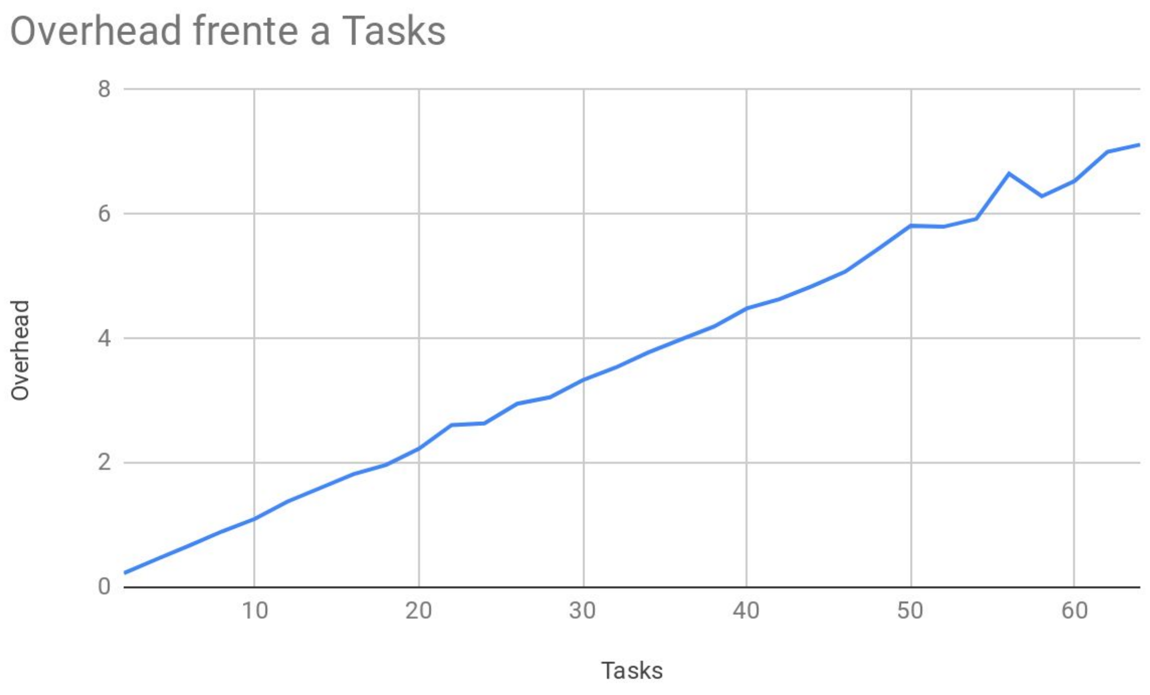


Figure 7 - Overheads / Number of Tasks of pi_omp_tasks

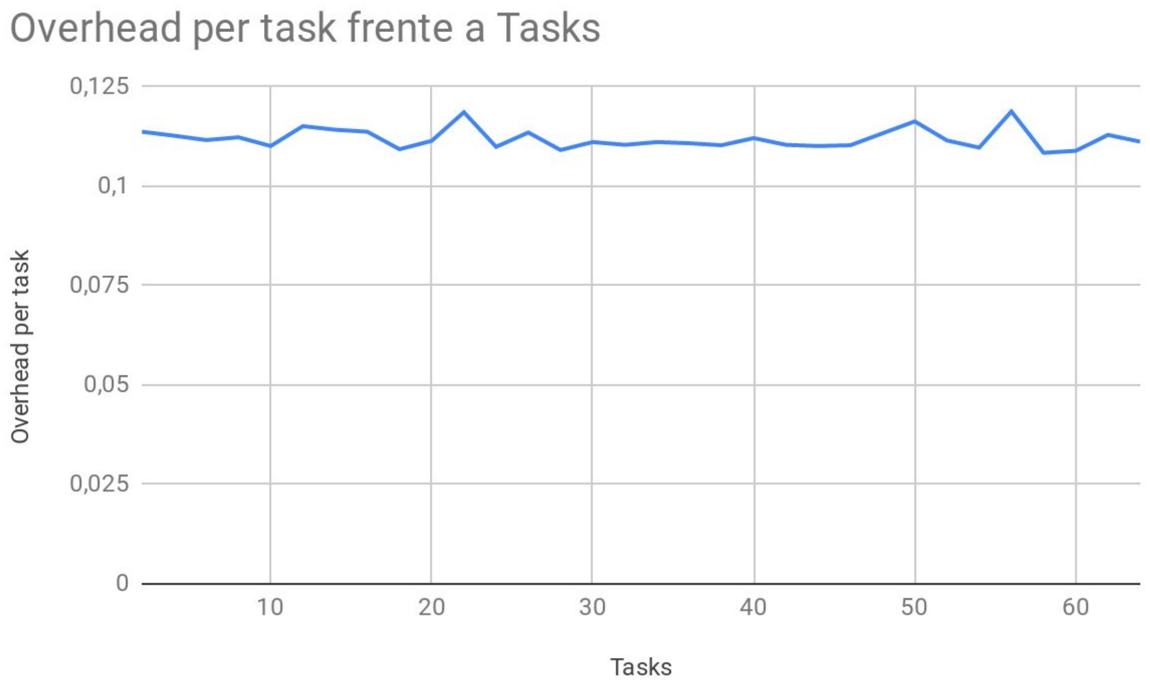


Figure 8 - Overheads per Task / Number of Tasks of pi_omp_tasks