PAR Laboratory Assignment

Lab 2: Brief tutorial on OpenMP programming model

PAR2303
Joan Manuel Ramos Refusta
Àlex Aguilera Martínez

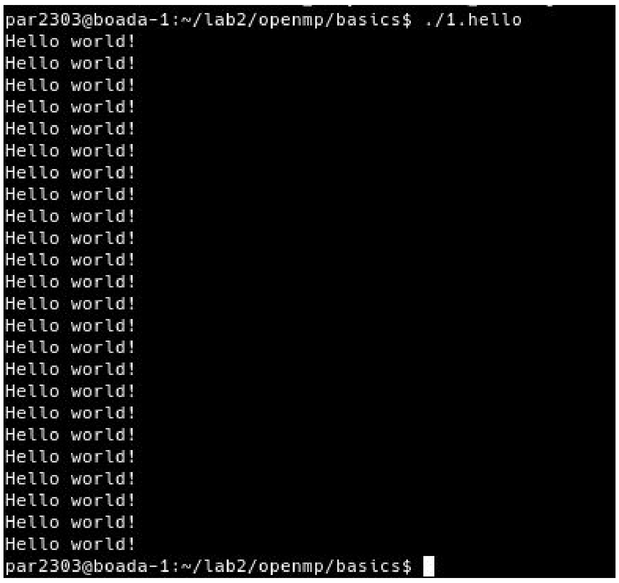
1.OpenMP questionnaire

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"?

24 times. That's because the cluster has 6 threads per core * 2 sockets per node * 2 cores per socket = 24 threads.

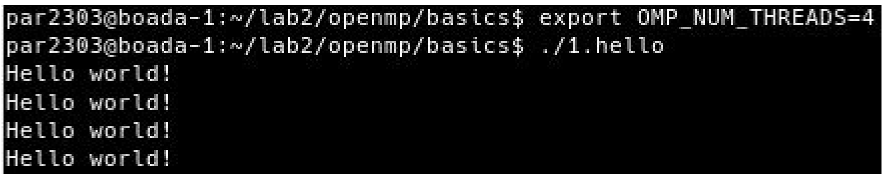


```
par2303@boada-1:~/lab2/openmp/basics$ ./1.hello
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
par2303@boada-1:~/lab2/openmp/basics$
```

Figure1, execution of ./1.hello

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

Changing the number of threads, by doing the command line:
export OMP_NUM_THREADS=4



```
par2303@boada-1:~/lab2/openmp/basics$ export OMP_NUM_THREADS=4
par2303@boada-1:~/lab2/openmp/basics$ ./1.hello
Hello world!
Hello world!
Hello world!
Hello world!
```

Figure 2, execution of .1.hello with 4 threads

2.hello.c:

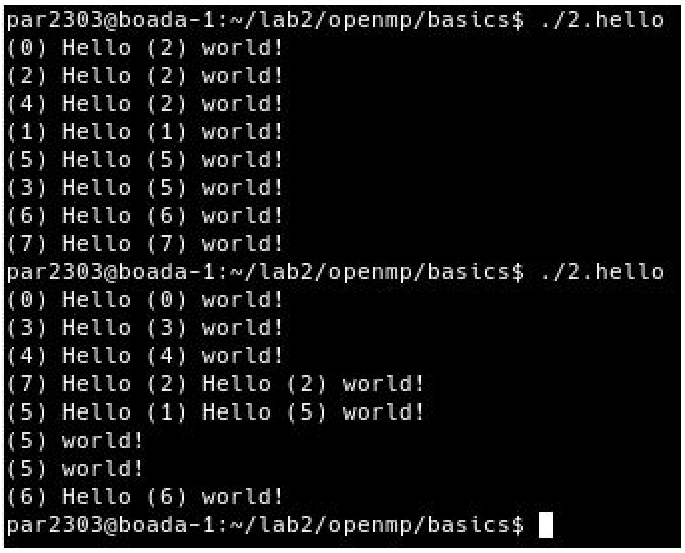
1. Is the execution of the program correct? (i.e., prints a sequence of "(Thid) Hello (Thid) world!" being Thid the thread identifier). If not, add a data sharing clause to make it correct?

No, it isn't correct. The id variable is changing all the time because it is in a parallel region and hence is shared between the threads.

In order to correct this error, this id variable must be private (we need to change #pragma omp parallel to #pragma omp parallel 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).

No, the lines aren't always printed in the same order because the threads don't have an order defined when they make the prints. The execution is not sequential. We include an example of that:



```
par2303@boada-1:~/lab2/openmp/basics$ ./2.hello
(0) Hello (2) world!
(2) Hello (2) world!
(4) Hello (2) world!
(1) Hello (1) world!
(5) Hello (5) world!
(3) Hello (5) world!
(6) Hello (6) world!
(7) Hello (7) world!
par2303@boada-1:~/lab2/openmp/basics$ ./2.hello
(0) Hello (0) world!
(3) Hello (3) world!
(4) Hello (4) world!
(7) Hello (2) Hello (2) world!
(5) Hello (1) Hello (5) world!
(5) world!
(5) world!
(6) Hello (6) world!
par2303@boada-1:~/lab2/openmp/basics$
```

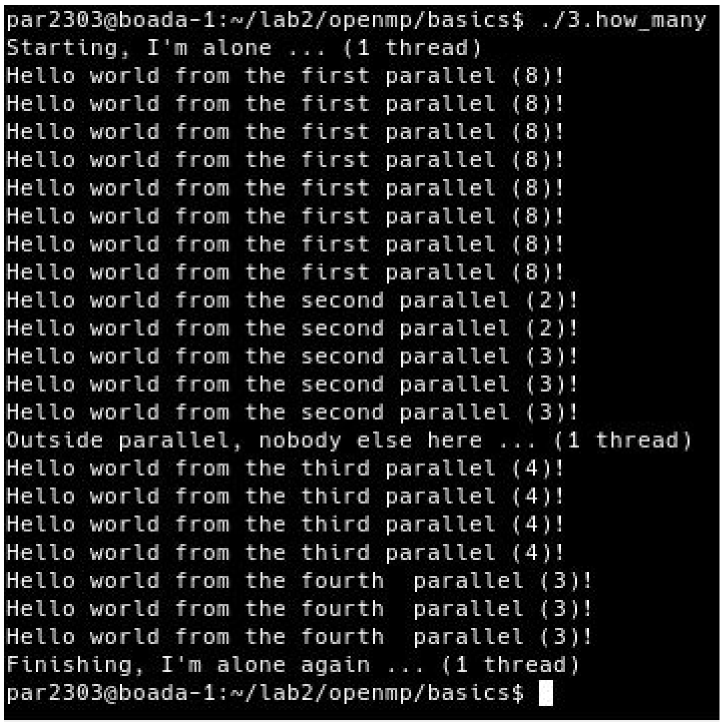
Figure 3, execution of ./2.hello

3.howmany.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 times:
- 8 times for the first region (export OMP_NUM_THREADS=8)
- 2 times for the second region (omp_set_num_threads(i), with i =2 at the beginning)
- 3 times for the second region (omp_set_num_threads(i), with i =3 at the end)
- One thread is alone because we just set the number of threads to 3 and there was 4.
- 4 times for the third region (#pragma omp parallel num_threads(4))
- 3 times for the fourth region, (omp_set_num_threads(i), with i=3 at the end, we have no #pragma omp parallel, there is no parallel region) and one thread is alone.



```
par2303@boada-1:~/lab2/openmp/basics$ ./3.how_many
Starting, I'm alone ... (1 thread)
Hello world from the first parallel (8)!
Hello world from the first parallel (8)!
Hello world from the first parallel (8)!
Hello world from the first parallel (8)!
Hello world from the first parallel (8)!
Hello world from the first parallel (8)!
Hello world from the first parallel (8)!
Hello world from the first parallel (8)!
Hello world from the second parallel (2)!
Hello world from the second parallel (2)!
Hello world from the second parallel (3)!
Hello world from the second parallel (3)!
Hello world from the second parallel (3)!
Outside parallel, nobody else here ... (1 thread)
Hello world from the third parallel (4)!
Hello world from the third parallel (4)!
Hello world from the third parallel (4)!
Hello world from the third parallel (4)!
Hello world from the fourth  parallel (3)!
Hello world from the fourth  parallel (3)!
Hello world from the fourth  parallel (3)!
Finishing, I'm alone again ... (1 thread)
par2303@boada-1:~/lab2/openmp/basics$ 
```

Figure 4, execution of ./3.how_many.c

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

Outside the parallel region, it returns us only 1 thread (because this thread is executing sequentially) while inside the parallel region, it depends on the parallel region and the code. Some regions are 8 and others 2, 3 or 4 as you can see in figure 4.

4. data_sharing.c

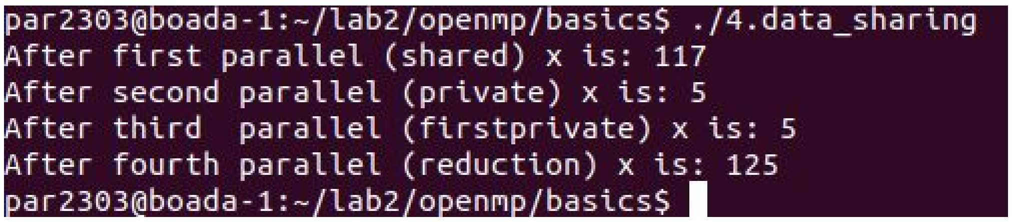
1. Which is the value of variable `x` after the execution of each parallel region with different data sharing attribute (shared,private,firstprivate and reduction)? Is that the value you would expect? (Execute several times if necessary)

The first value of x (shared) is different in each execution because x is shared and that means that every thread can modify variable x and change his value. Every thread is modifying the same variable summing the number of threads.

The second value (private) is always the same, 5. In this case, x is private inside the parallel zone and which means that a new copy of the variable is created in order to "protect" the real x. Inside, the variable is not initialized with 5 and the sum starts with a 0 (this is important because with "firstprivate", is different). Outside, the program only prints the "real" x and not the copy of it.

The third value (firstprivate) is always the same, 5. In this case, inside, x is private and as well as the "private" clause, the program creates a copy of the real x. However, x is initialized with the value of the real x (value=5). Outside, it happens the same as private clause and the program only prints the real x.

And the last one (reduction) is always the same, 125. x is calculating the sum of the number of threads correctly and dividing the work proportionally between the different threads.



```
par2303@boada-1:~/lab2/openmp/basics$ ./4.data_sharing
After first parallel (shared) x is: 117
After second parallel (private) x is: 5
After third  parallel (firstprivate) x is: 5
After fourth parallel (reduction) x is: 125
par2303@boada-1:~/lab2/openmp/basics$
```

Figure 5, execution of data_sharing.c

B) Loop parallelism

1.schedule.c

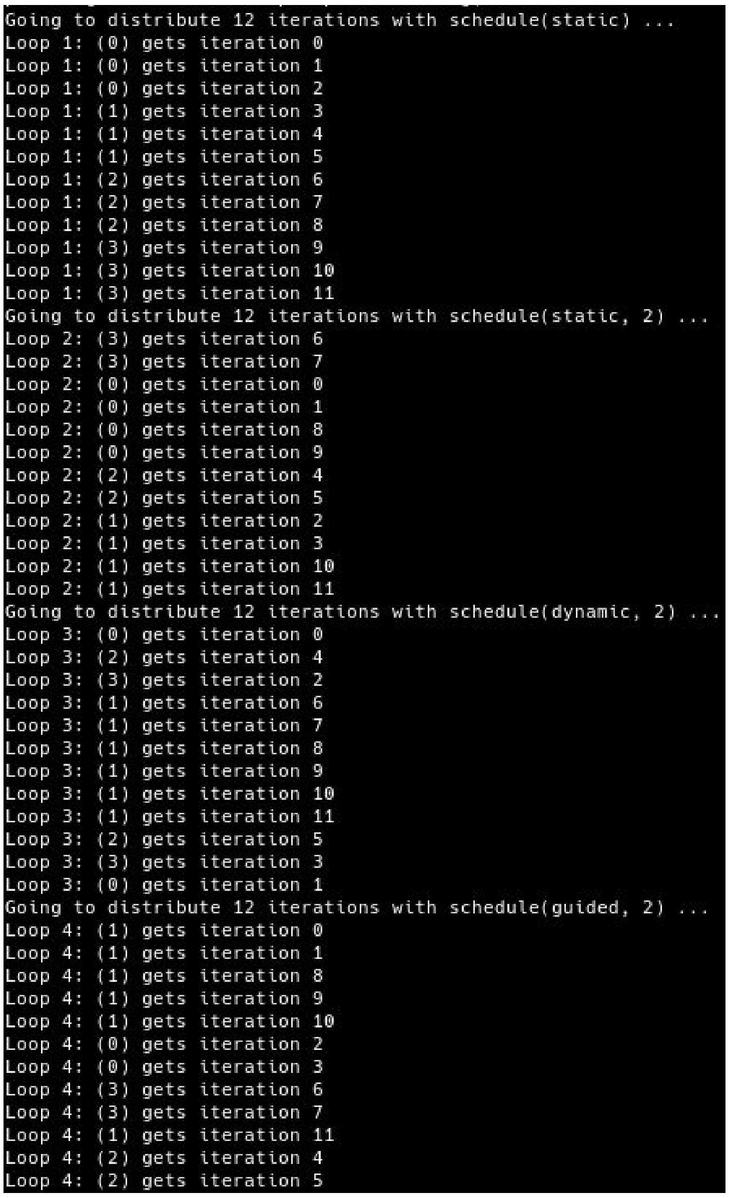
1. Which iterations of the loops are executed by each thread for each schedule kind?

- Static: The iterations are equally distributed between all the threads, so each thread executes the half part of the total iterations consecutively.

- Static (chunk = 2): The iterations are equally distributed between all the threads and every thread is executing only 2 consecutively iterations.

- Dynamic (chunk = 2): The iterations are distributed between all the threads but in this case each group of 2 iterations is executed for the first free thread.

- Guided (chunk = 2): It’s similar to dynamic, but in this case, chunk_size defines a lower bound of the number of consecutive iterations done for a thread, that thread can do more in order to balance the work between processors.



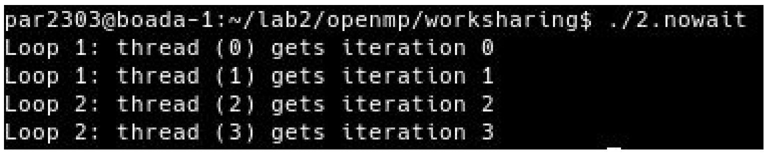
```
Going to distribute 12 iterations with schedule(static) ...
Loop 1: (0) gets iteration 0
Loop 1: (0) gets iteration 1
Loop 1: (0) gets iteration 2
Loop 1: (1) gets iteration 3
Loop 1: (1) gets iteration 4
Loop 1: (1) gets iteration 5
Loop 1: (2) gets iteration 6
Loop 1: (2) gets iteration 7
Loop 1: (2) gets iteration 8
Loop 1: (3) gets iteration 9
Loop 1: (3) gets iteration 10
Loop 1: (3) gets iteration 11
Going to distribute 12 iterations with schedule(static, 2) ...
Loop 2: (3) gets iteration 6
Loop 2: (3) gets iteration 7
Loop 2: (0) gets iteration 0
Loop 2: (0) gets iteration 1
Loop 2: (0) gets iteration 8
Loop 2: (0) gets iteration 9
Loop 2: (2) gets iteration 4
Loop 2: (2) gets iteration 5
Loop 2: (1) gets iteration 2
Loop 2: (1) gets iteration 3
Loop 2: (1) gets iteration 10
Loop 2: (1) gets iteration 11
Going to distribute 12 iterations with schedule(dynamic, 2) ...
Loop 3: (0) gets iteration 0
Loop 3: (2) gets iteration 4
Loop 3: (3) gets iteration 2
Loop 3: (1) gets iteration 6
Loop 3: (1) gets iteration 7
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: (2) gets iteration 5
Loop 3: (3) gets iteration 3
Loop 3: (0) gets iteration 1
Going to distribute 12 iterations with schedule(guided, 2) ...
Loop 4: (1) gets iteration 0
Loop 4: (1) gets iteration 1
Loop 4: (1) gets iteration 8
Loop 4: (1) gets iteration 9
Loop 4: (1) gets iteration 10
Loop 4: (0) gets iteration 2
Loop 4: (0) gets iteration 3
Loop 4: (3) gets iteration 6
Loop 4: (3) gets iteration 7
Loop 4: (1) gets iteration 11
Loop 4: (2) gets iteration 4
Loop 4: (2) gets iteration 5
```

Figure 6, execution of 1.schedule.c

2.nowait.c

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

In every execution, we have the same result: (with the nowait clause)



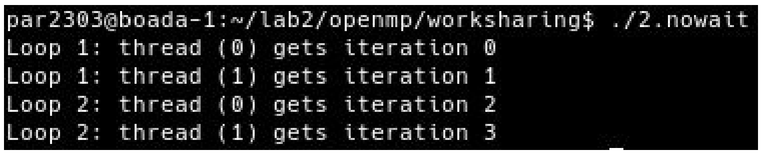
```
par2303@boada-1:~/lab2/openmp/worksharing$ ./2.nowait
Loop 1: thread (0) gets iteration 0
Loop 1: thread (1) gets iteration 1
Loop 2: thread (2) gets iteration 2
Loop 2: thread (3) gets iteration 3
```

Figure 7, execution of 2.nowait.c

Any thread waits the team to continue executing the loop 2. Thread 2 and 3 do the work.

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

In this case, if we remove the nowait clause, we get the following result:



```
par2303@boada-1:~/lab2/openmp/worksharing$ ./2.nowait
Loop 1: thread (0) gets iteration 0
Loop 1: thread (1) gets iteration 1
Loop 2: thread (0) gets iteration 2
Loop 2: thread (1) gets iteration 3
```

Figure 8, execution of 2.nowait.c (without nowait)

The nowait clause is used to remove the synchronization between the different threads. Therefore, without the nowait clause, threads can continue the execution of a parallel region without waiting for the other threads in the team to complete the region. In the figure 8, we can see how thread 0 is not executed until thread 1 ends the job in loop 1. Every execution, the prints are different.

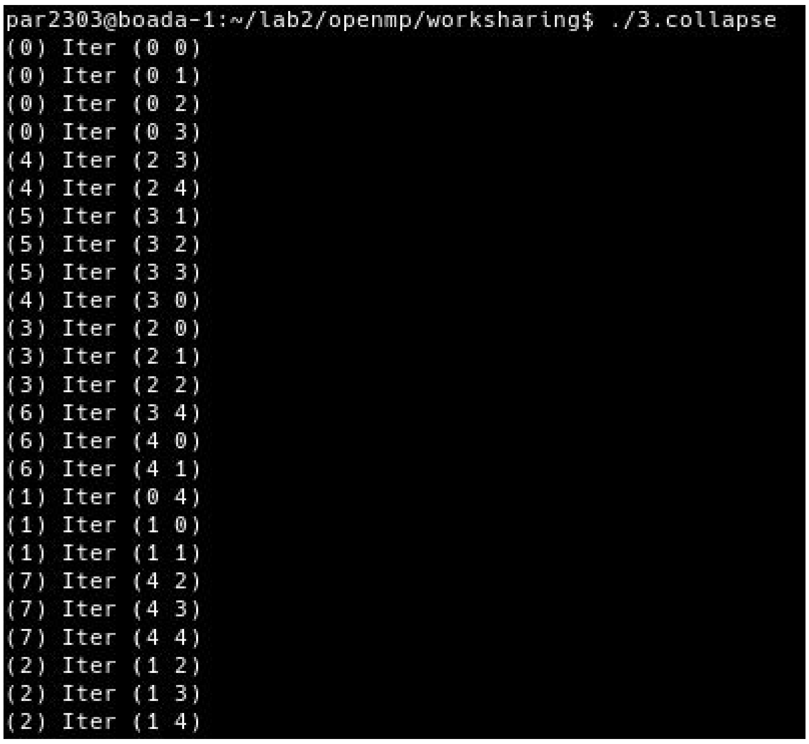
3. What would happen if dynamic is changed to static in the schedule in both loops? (keeping the nowait clause)

In this case, the execution is quite random too. The static clause means that OpenMP divides the iterations into chunks of size chunk-size (1, in this code) and it distributes the chunks to threads in a circular order. Therefore, we get a similar result as figure 8.

3.collapse.c

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

Part of the instructions of the interior loop are executed sequentially (collapse clause group sets of iterations of the interior loop as a only iteration of the exterior loop). Each thread executes a consecutive number of iterations of the two fors combined.



```
par2303@boada-1:~/lab2/openmp/worksharing$ ./3.collapse
(0) Iter (0 0)
(0) Iter (0 1)
(0) Iter (0 2)
(0) Iter (0 3)
(4) Iter (2 3)
(4) Iter (2 4)
(5) Iter (3 1)
(5) Iter (3 2)
(5) Iter (3 3)
(4) Iter (3 0)
(3) Iter (2 0)
(3) Iter (2 1)
(3) Iter (2 2)
(6) Iter (3 4)
(6) Iter (4 0)
(6) Iter (4 1)
(1) Iter (0 4)
(1) Iter (1 0)
(1) Iter (1 1)
(7) Iter (4 2)
(7) Iter (4 3)
(7) Iter (4 4)
(2) Iter (1 2)
(2) Iter (1 3)
(2) Iter (1 4)
```

Figure 9, execution of 3.collapse.c

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

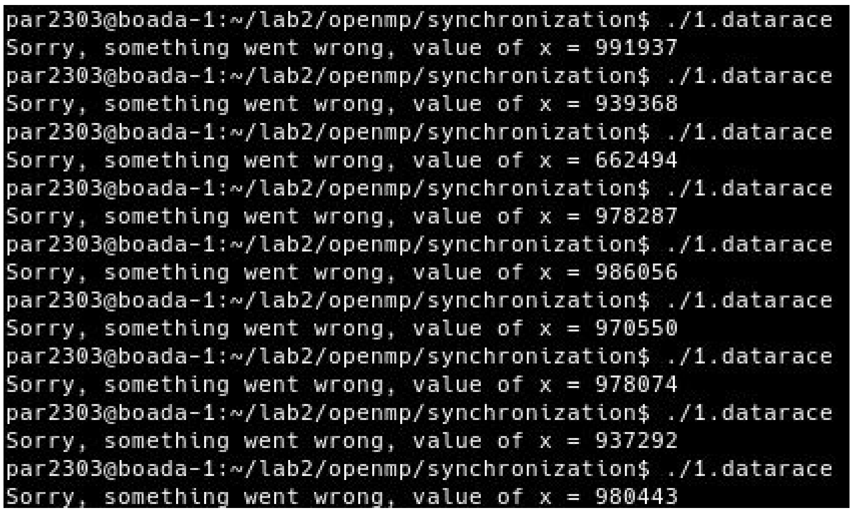
The execution is not correct because some executions are repeated.
One form to prevent this is making private the variable j, (with the line #pragma omp parallel for private(j) in order to avoid the overriding of the variable j for every thread execution.

C) Synchronization

1.datarace.c (execute several times before answering the questions)

1. Is the program always executing correctly?

No, most of the time the execution will show:

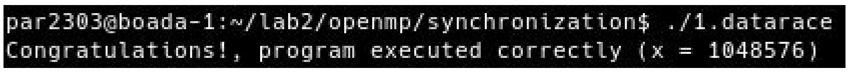


```
par2303@boada-1:~/lab2/openmp/synchronization$ ./1.datarace
Sorry, something went wrong, value of x = 991937
par2303@boada-1:~/lab2/openmp/synchronization$ ./1.datarace
Sorry, something went wrong, value of x = 939368
par2303@boada-1:~/lab2/openmp/synchronization$ ./1.datarace
Sorry, something went wrong, value of x = 662494
par2303@boada-1:~/lab2/openmp/synchronization$ ./1.datarace
Sorry, something went wrong, value of x = 978287
par2303@boada-1:~/lab2/openmp/synchronization$ ./1.datarace
Sorry, something went wrong, value of x = 986056
par2303@boada-1:~/lab2/openmp/synchronization$ ./1.datarace
Sorry, something went wrong, value of x = 970550
par2303@boada-1:~/lab2/openmp/synchronization$ ./1.datarace
Sorry, something went wrong, value of x = 978074
par2303@boada-1:~/lab2/openmp/synchronization$ ./1.datarace
Sorry, something went wrong, value of x = 937292
par2303@boada-1:~/lab2/openmp/synchronization$ ./1.datarace
Sorry, something went wrong, value of x = 980443
```

Figure 10, execution of 1.datarace.c

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

1. #pragma omp single, by this way, every iteration it's executed by the first thread that finds it.
2. #pragma omp for schedule(static) nowait, by this way, the nowait clause allows thread to continue the execution of a parallel region without waiting for the other threads, and with the static clause iterations are equally distributed between all the threads.



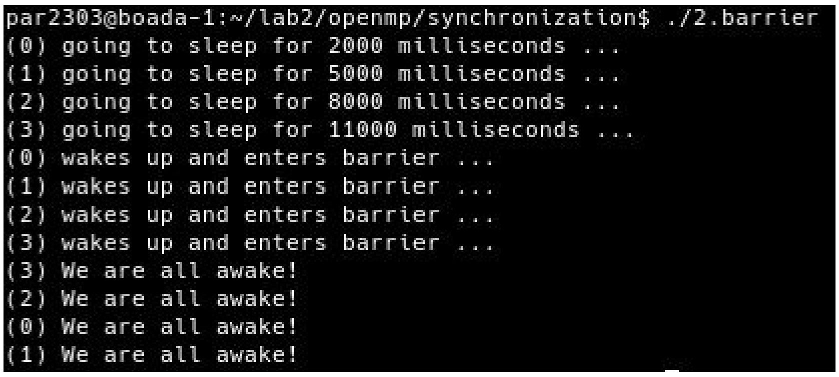
```
par2303@boada-1:~/lab2/openmp/synchronization$ ./1.datarace
Congratulations!, program executed correctly (x = 1048576)
```

Figure 11, 1.datarace.c executed correctly

2.barrier.c

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

The barrier clause provides us a point in the execution of the program where threads wait for each other. No thread is allowed to continue after this barrier until all threads in a team reach this point.
Therefore, we are always going to get different prints order before and after the barrier because threads have to wait in that point until the rest end.



```
par2303@boada-1:~/lab2/openmp/synchronization$ ./2.barrier
(0) going to sleep for 2000 milliseconds ...
(1) going to sleep for 5000 milliseconds ...
(2) going to sleep for 8000 milliseconds ...
(3) going to sleep for 11000 milliseconds ...
(0) wakes up and enters barrier ...
(1) wakes up and enters barrier ...
(2) wakes up and enters barrier ...
(3) wakes up and enters barrier ...
(3) We are all awake!
(2) We are all awake!
(0) We are all awake!
(1) We are all awake!
```

Figure 12, execution of 2.barrier.c

3.ordered.c

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

We can be sure that inside the ordered pragma every iteration are going to execute in order, but before the ordered pragma we can’t be sure about the order of the printfs because 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?

Changing the schedule to static with chunk_size = 2 and ordered.
#pragma omp for schedule(static,2) ordered

D) Tasks

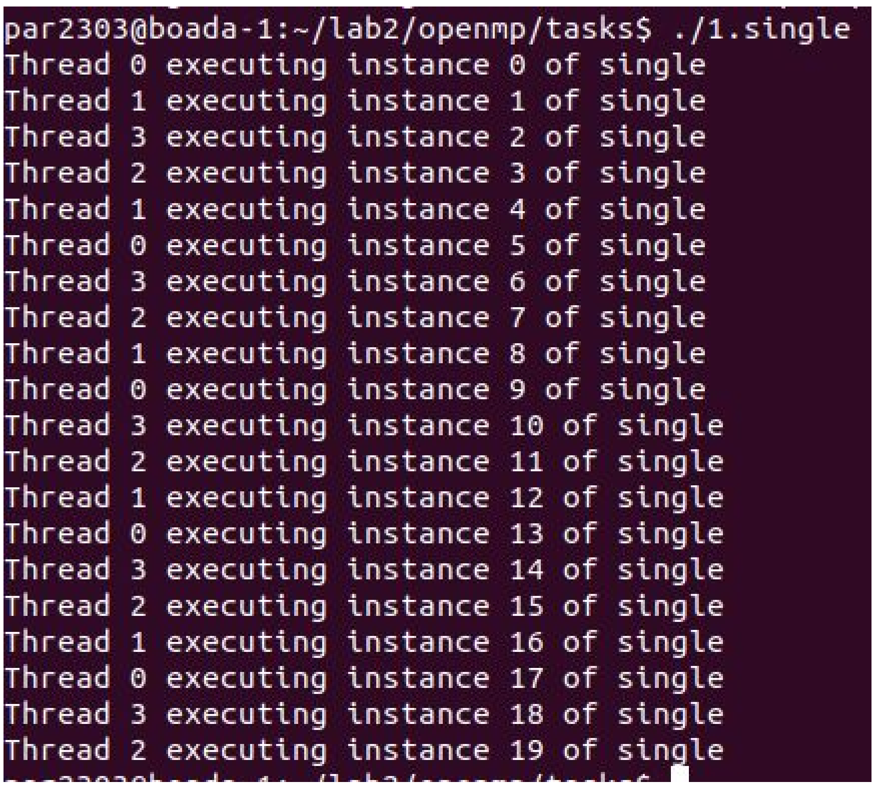
1.single.c

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?

We have two clauses in this example, but the important one is the single clause which identifies a section of the code which must be executed only once by a single available thread and threads should wait at the implicit barrier, but nowait clause is specified as well and then the rest of the threads immediately continue executing the code after the region.

Therefore, the different threads execute only one time each iteration or instance because only one thread can execute that section of the code. We can see that in figure 13.

Moreover, those instances appear to be executed in bursts of 4 because there is a sleep(1) and all the threads (only 4, specified with omp_set_num_threads(4)) wait there 1 second.



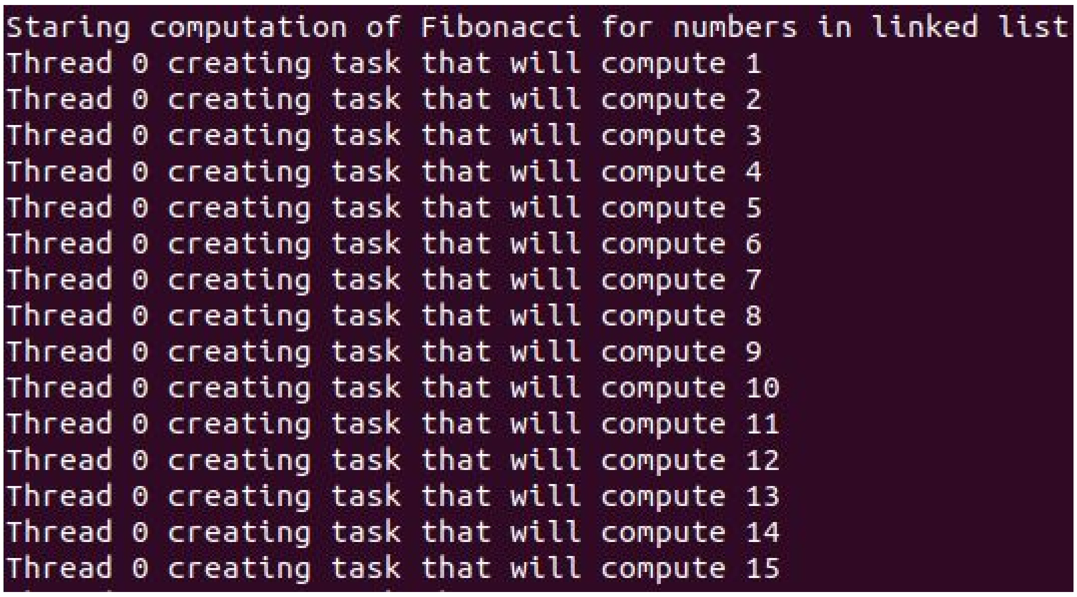
```
par2303@boada-1:~/lab2/openmp/tasks$ ./1.single
Thread 0 executing instance 0 of single
Thread 1 executing instance 1 of single
Thread 3 executing instance 2 of single
Thread 2 executing instance 3 of single
Thread 1 executing instance 4 of single
Thread 0 executing instance 5 of single
Thread 3 executing instance 6 of single
Thread 2 executing instance 7 of single
Thread 1 executing instance 8 of single
Thread 0 executing instance 9 of single
Thread 3 executing instance 10 of single
Thread 2 executing instance 11 of single
Thread 1 executing instance 12 of single
Thread 0 executing instance 13 of single
Thread 3 executing instance 14 of single
Thread 2 executing instance 15 of single
Thread 1 executing instance 16 of single
Thread 0 executing instance 17 of single
Thread 3 executing instance 18 of single
Thread 2 executing instance 19 of single
```

Figure 13, execution of 1.single.c

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?

The program is not executing in parallel, all the work is done by the thread 0, we can see it in figure 14. There is not any #pragma omp parallel in the code, the task clause needs that in order to execute in parallel. The final result of the execution is correct, it is printing the fibonacci numbers, but sequentially.



```
Staring computation of Fibonacci for numbers in linked list
Thread 0 creating task that will compute 1
Thread 0 creating task that will compute 2
Thread 0 creating task that will compute 3
Thread 0 creating task that will compute 4
Thread 0 creating task that will compute 5
Thread 0 creating task that will compute 6
Thread 0 creating task that will compute 7
Thread 0 creating task that will compute 8
Thread 0 creating task that will compute 9
Thread 0 creating task that will compute 10
Thread 0 creating task that will compute 11
Thread 0 creating task that will compute 12
Thread 0 creating task that will compute 13
Thread 0 creating task that will compute 14
Thread 0 creating task that will compute 15
```

Figure 14, execution of 2.fibtasks.c (not in parallel)

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

A way to solve the problem and to execute the program in parallel, is making the following change: (it must be other ways to do it)

```
while (p != NULL) {
   #pragma omp parallel
   #pragma omp single
   {
      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;
   }
}
```

Figure 15, a solution of 2.fibtasks.c

The single clause is important to include it because we know that the section of the code which must be executed only once by a single available thread and the firstprivate(p) clause is needed because specifies that each thread should have its own instance of a variable and they will not share that variable.

3.synchtasks.c

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

If we follow the task dependencies of the code (in the main), we can obtain a task dependence graph like this:

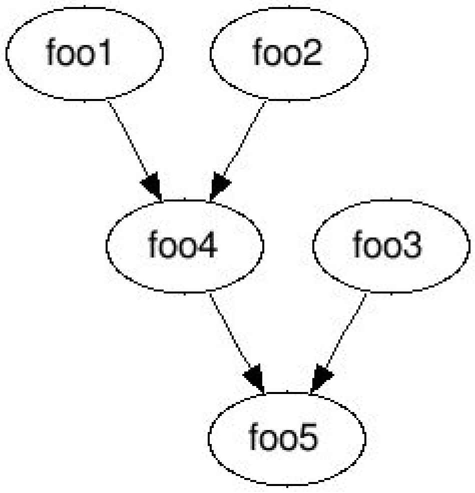
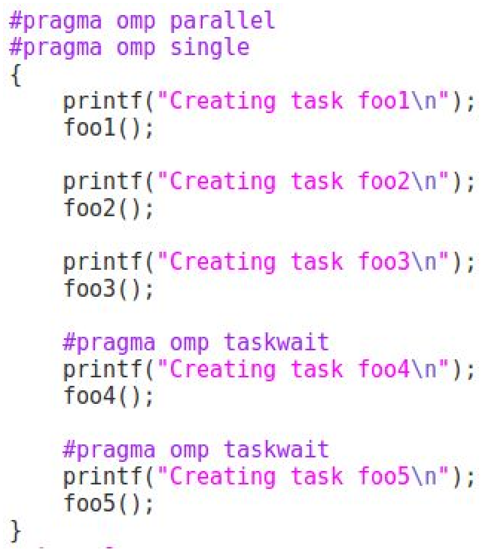


Figure 16, task dependence graph of synchtasks.c

In the result of the execution, we can see the dependencies as well. Until foo1 and foo2 are not finished, foo4 is not created. And the same happens with foo5, until foo4 and foo3 finish, foo5 is not created.

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



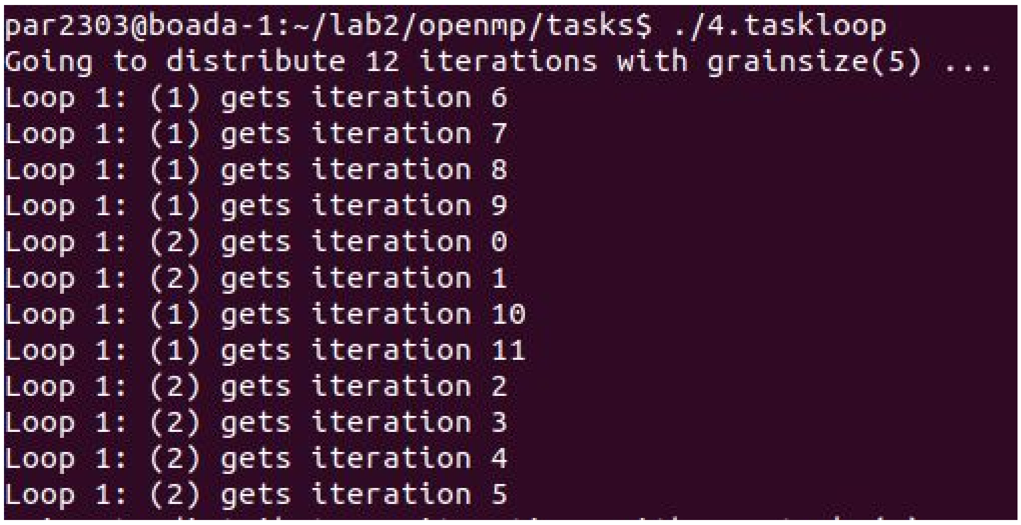
```
#pragma omp parallel
#pragma omp single
{
    printf("Creating task foo1\n");
    foo1();

    printf("Creating task foo2\n");
    foo2();

    printf("Creating task foo3\n");
    foo3();

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

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

Figure 17, synchtasks.c without depend clauses

4.taskloop.c

1. Find out how many tasks and how many iterations each task execute when using the grain size 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.

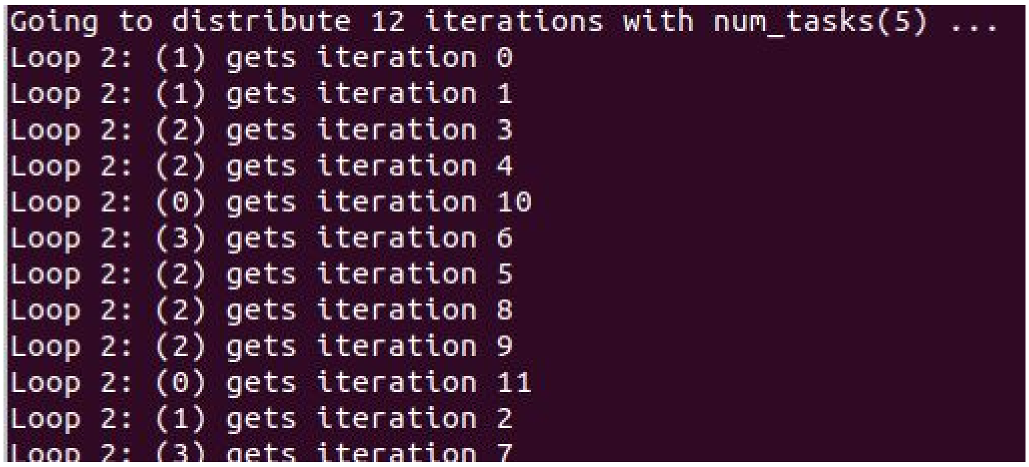
Using **gran size**, we control how many loop iterations are assigned to each task and it will be greater or equal than the minimum of the value of the grain size (in our case, 5). Therefore, we will always obtain different results, but always having a greater or equal to 5 of loop iterations assigned. However, the number of tasks are usually 1 or 2, because N is quite low (12). We can see this much better if we increase N (the total number of iterations).



```
par2303@boada-1:~/lab2/openmp/tasks$ ./4.taskloop
Going to distribute 12 iterations with grainsize(5) ...
Loop 1: (1) gets iteration 6
Loop 1: (1) gets iteration 7
Loop 1: (1) gets iteration 8
Loop 1: (1) gets iteration 9
Loop 1: (2) gets iteration 0
Loop 1: (2) gets iteration 1
Loop 1: (1) gets iteration 10
Loop 1: (1) gets iteration 11
Loop 1: (2) gets iteration 2
Loop 1: (2) gets iteration 3
Loop 1: (2) gets iteration 4
Loop 1: (2) gets iteration 5
```

Figure 18, example of taskloop.c execution with gransize(5)

Using **num tasks clause**, the taskloop construct creates as many tasks as the minimum of the num_tasks expression and the number of loop iterations. Therefore, the program will be always executed with less or equal than 5 tasks and loop iterations assigned.



```
Going to distribute 12 iterations with num_tasks(5) ...
Loop 2: (1) gets iteration 0
Loop 2: (1) gets iteration 1
Loop 2: (2) gets iteration 3
Loop 2: (2) gets iteration 4
Loop 2: (0) gets iteration 10
Loop 2: (3) gets iteration 6
Loop 2: (2) gets iteration 5
Loop 2: (2) gets iteration 8
Loop 2: (2) gets iteration 9
Loop 2: (0) gets iteration 11
Loop 2: (1) gets iteration 2
Loop 2: (3) gets iteration 7
```

Figure 19, example of taskloop.c execution with num_tasks(5)

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

If the nogroup clause is present, no implicit taskgroup region is created and then the two loops are executed at the same time. The two prints before loops are not executed.

2.Observing overheads

Thread creation and termination

Executing the command line: “qsub -l execution ./submit-omp.sh pi_omp_parallel 1 24”, the following file is generated:

```
All overheads expressed in microseconds
Nthr   Overhead   Overhead per thread
2   1.2642      0.6321
3   1.6165      0.5388
4   1.4424      0.3606
5   1.5811      0.3162
6   1.6825      0.2804
7   1.7490      0.2499
8   1.8407      0.2301
9   1.9313      0.2146
10   2.0231     0.2023
11   2.0223     0.1838
12   2.2300     0.1858
13   2.4175     0.1860
14   2.4842     0.1774
15   2.4631     0.1642
16   2.6875     0.1680
17   3.0034     0.1767
18   2.9857     0.1659
19   2.9916     0.1575
20   3.0272     0.1514
21   3.0630     0.1459
22   3.0724     0.1397
23   3.0749     0.1337
24   3.5485     0.1479
```

Figure 20, pi_omp_parallel_1_24.txt

We can see that with 1 unique iteration and 24 threads, we can prove that the time of the overheads (with order of magnitude in microseconds) is increasing linearly but is not constant at all. We can observe that the overhead per thread tend to decrease because when the number of threads increases, every thread has to do less calculations.

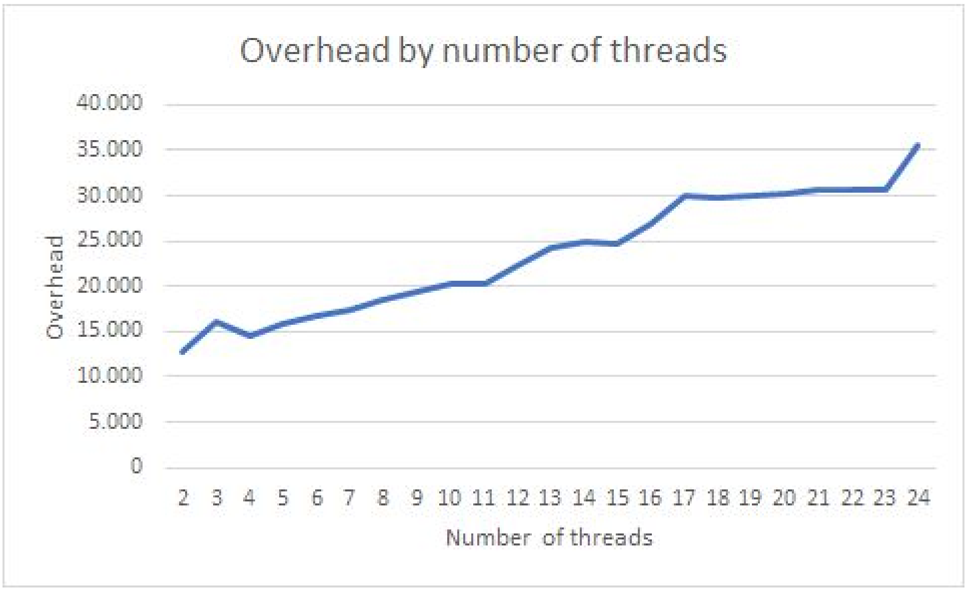


Figure 21, Plot of the overhead by number of threads based on figure 20.

Task creation and synchronization

```
All overheads expressed in microseconds
Ntasks    Overhead    Overhead per task
2    0.0849       0.0425
4    0.5049       0.1262
6    0.7460       0.1243
8    0.9884       0.1235
10    1.2279     0.1228
12    1.4656     0.1221
14    1.7044     0.1217
16    1.9476     0.1217
18    2.1897     0.1217
20    2.4673     0.1234
22    2.6633     0.1211
24    2.8977     0.1207
26    3.1478     0.1211
28    3.3848     0.1209
30    3.6237     0.1208
32    3.8672     0.1208
34    4.1051     0.1207
36    4.3494     0.1208
38    4.5881     0.1207
40    4.8294     0.1207
42    5.0622     0.1205
44    5.3061     0.1206
46    5.5422     0.1205
48    5.8230     0.1213
50    6.0353     0.1207
52    6.2643     0.1205
54    6.5051     0.1205
56    6.7358     0.1203
58    7.0157     0.1210
60    7.2119     0.1202
62    7.4659     0.1204
64    7.7045     0.1204
```

Figure 22, pi_omp_tasks_10_1.txt

We can see that with 10 iterations and 1 unique thread, we can prove that the time of the overheads (with order of magnitude in microseconds) is increasing linearly. We can observe that the overhead per task it's almost constant, it's doesn't matter the number of tasks, this value will cost the same, that event is produced because the variation is practically null.

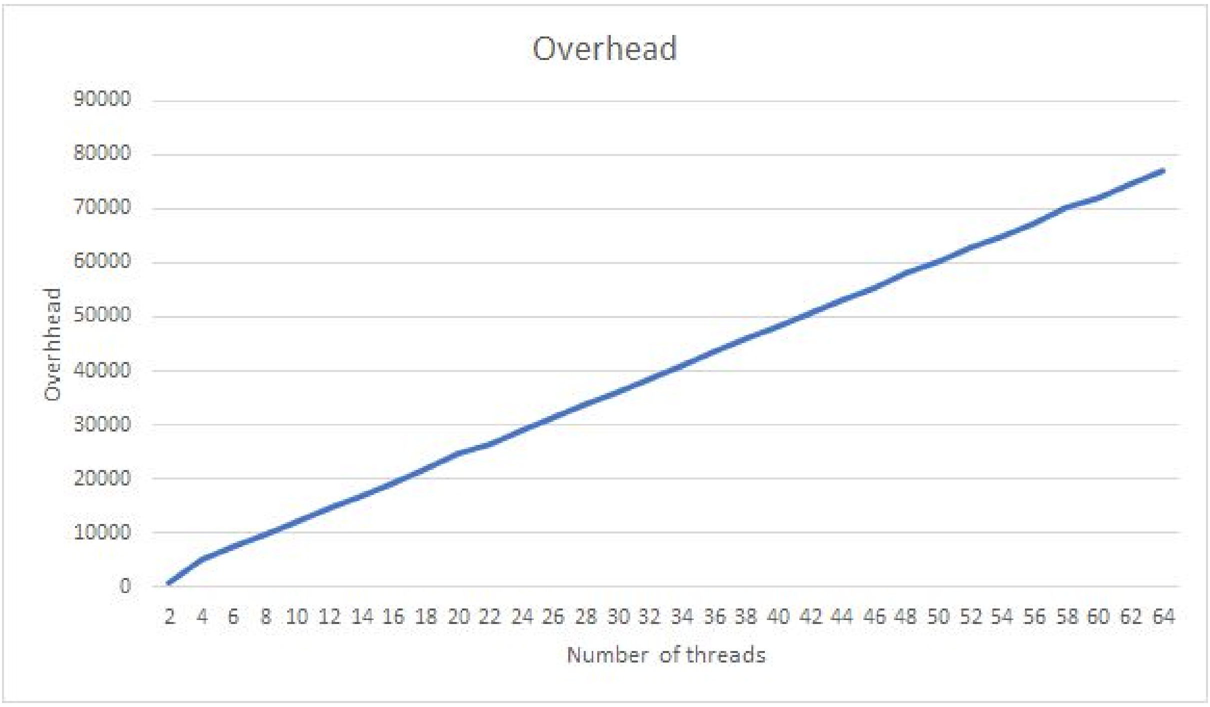


Figure 23, Plot of the overhead by number of threads based on figure 22.