PAR Laboratory Assignment

Lab 2: Brief tutorial on OpenMP programming model

UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

20/10/2021

Fall 2021-22

Index

1 OpenMP questionnaire

Day 1

1.hello.c

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

2 times. Writes as many times as processors have.

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

Executing the command `OMP_NUM_THREADS=4 ./1.hello`

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's not correct, the variable *id* is shared by all threads, to make it correct we must make *id* private. The statement will be: `#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).

The lines are not printed in the same order, that's because some threads start to print the line before others, because of we are not deciding the order in which each thread has to be executed.

3.how_many.c:

Assuming the OMP NUM THREADS variable is set to 8 with "OMP NUM THREADS=8 ./3.how_many"

1. What does omp get num threads return when invoked outside and inside a parallel region?

Outside the parallel region returns 1, because of there is only one thread executing the statement, on the other hand, inside the parallel region returns the number of threads that are executing the statement, this number is stablished before by the piece of code: `#pragma omp parallel` **`num_threads(x)`**

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

- #pragma omp parallel num_threads(x)
- With the function: omp_set_num_threads(x)

In both cases x is the number of threads that we will use to execute the parallel region.

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

For the first case (#pramga omp parallel num_threads(x)), the lifespan will be applied for the parallel region inside "{ … }" or the line after the declaration. Alternately, if we use the function omp_set_num_threads(x), x will be the number of threads executed for the whole program that remains.

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)

El resultado de la ejecucion es el siguiente:

```
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 shared part, the number that we obtain not always will be 120, because there is danger of data race. The variable x could change as result of an assignation of other thread while the initial thread is computing the sum.

The second parallel region the variable *x* is initialize with a 5, and for each thread that is executed next the variable *x* is a copy, because of `private(x)` statement, so we are not changing the real value of *x*.

To third parallel region happens the same as before, inside the parallel region the value of *x* is changing, but outside the parallel region remains the same, because of the *x* inside the region is only a copy.

The last parallel region, reduction, each thread creates a private copy of *x* and at the end of the parallel region ensures that the shared variable is properly updated with the partial values of each thread. Is doing the same operation as the first paralell region, but instead of the initial value of *x* be 0 is 5.

5.datarace.c

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

Not always returns the correct result, as consequence of the order that the code is executed, the code could erase a *maxvalue*, depending on how the value of the variable is evaluated and assigned.

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.

Solution 1:

```c
#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];
    }
    }
```

By applying reduction, the compiler ensures the correct evaluation and assignation of the *maxvalue* variable.

Solution 2:

```c
    omp_lock_t lock;
    int i, maxvalue=0;

    omp_init_lock(&lock);
    omp_set_num_threads(8);
    #pragma omp parallel private(i)
    {
    int id = omp_get_thread_num();
    int howmany = omp_get_num_threads();

    for (i=id; i < N; i+=howmany) {
        omp_set_lock(&lock);
        if (vector[i] > maxvalue)
            maxvalue = vector[i];
        omp_unset_lock(&lock);
    }
    }
    omp_destroy_lock(&lock);
```

This solution is based on the lock the piece of code that could cause problems with the different threads, because of we not decide the order of the thread's execution.

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.

By dividing the number of elements by the number of threads we can distribute the work equally.

```c
    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();

    int it = N/howmany;
    int start = id*it;

    for (i=start; i < start+it; i++) {
        if (vector[i] > maxvalue)
           maxvalue = vector[i];
    }
    }
```

6.datarace.c

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

As well as the previous code, the threads execution order can cause incorrect adding's and evaluations, this will lead to an incorrect result.

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.

Solution 1:

```
    omp_set_num_threads(8);
    #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++;
    }
    }
```

This solution is the same as the previous part.

Solution 2:

```
    omp_set_num_threads(8);
    #pragma omp parallel private(i)
    {
    int id = omp_get_thread_num();
    int howmany = omp_get_num_threads();

    for (i=id; i < N; i+=howmany) {
        if (vector[i]==maxvalue) {
            #pragma omp atomic
            countmax++;
        }
    }
    }
```

By applying atomic we ensure that the variable *countmax* avoids the possibility of multiple writing threads.

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)

The program is not executing correctly, because of each thread maintains a local variable of *countmax* and *maxvalue*, each one maintains its own values that do not correspond to reality.

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

The first think to do is to share the variables *countmax* and *maxvalue*, to do that we delete the reduction statements and a add a shared instead, doing this the threads are able to know the real maximum value found and the *countmax*.

To avoid wrong executions, we add the statement `#pragma omp atomic`, we could add a `#pragma omp critical`, but usually is more efficient the first option.

```
omp_set_num_threads(8);
    #pragma omp parallel private(i) shared(countmax, maxvalue)
    {
    int id = omp_get_thread_num();
    int howmany = omp_get_num_threads();

    for (i=id; i < N; i+=howmany) {
        if (vector[i]==maxvalue) {
            #pragma omp atomic
            countmax++;
        }

        if (vector[i] > maxvalue) {
            maxvalue = vector[i];
            countmax = 1;
        }
    }
    }
```

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?

We can predict that the messages “going to sleep” will print before the messages “wakes up and enters barrier” and in the same way these will print before the messages “We are all awake!”, but we can’t specify the order of threads for each message, only for the “going to sleep” messages, because we sent to sleep different milliseconds.

Day 2

1.single.c

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

The implicit barrier that we have at the end of the single is removed, that allows threads to start doing the next task early.

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 loop have to do 20 iterations, as we have 4 threads, each of them will execute 20/4=5 iterations and those instances appear to be executed in bursts because of the sleep(1) statement.

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?

We assign 6 threads to the program, but we do not call any declaration to parallelize the tasks.

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

We added `#pragma omp parallel` to parallelize the tasks and `#pragma omp single` to the threads calculates the Fibonacci sequence correctly, if we do not add the single the threads calculate random numbers.

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

All the threads create a new variable with the same initial value, if we comment it the original code, doesn't do anything different because we only use one thread to execute all the code. If we comment it with the above code, we use less threads, because we have to use the original variable.

3.taskloop.c

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

The first thread generates all the tasks sequentially, after that, the available threads executes the tasks.

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?

It depends on the available threads. The number of iterations is defined by *N*.

3. Can grainsize and num tasks be used at the same time in the same loop?

If we try to add both clauses, when we try to compile the code it will give the error that the directive cannot contain both grainsize and num_tasks clauses.

4. What is happening with the execution of tasks if the nogroup clause is uncommented in the first loop? Why?

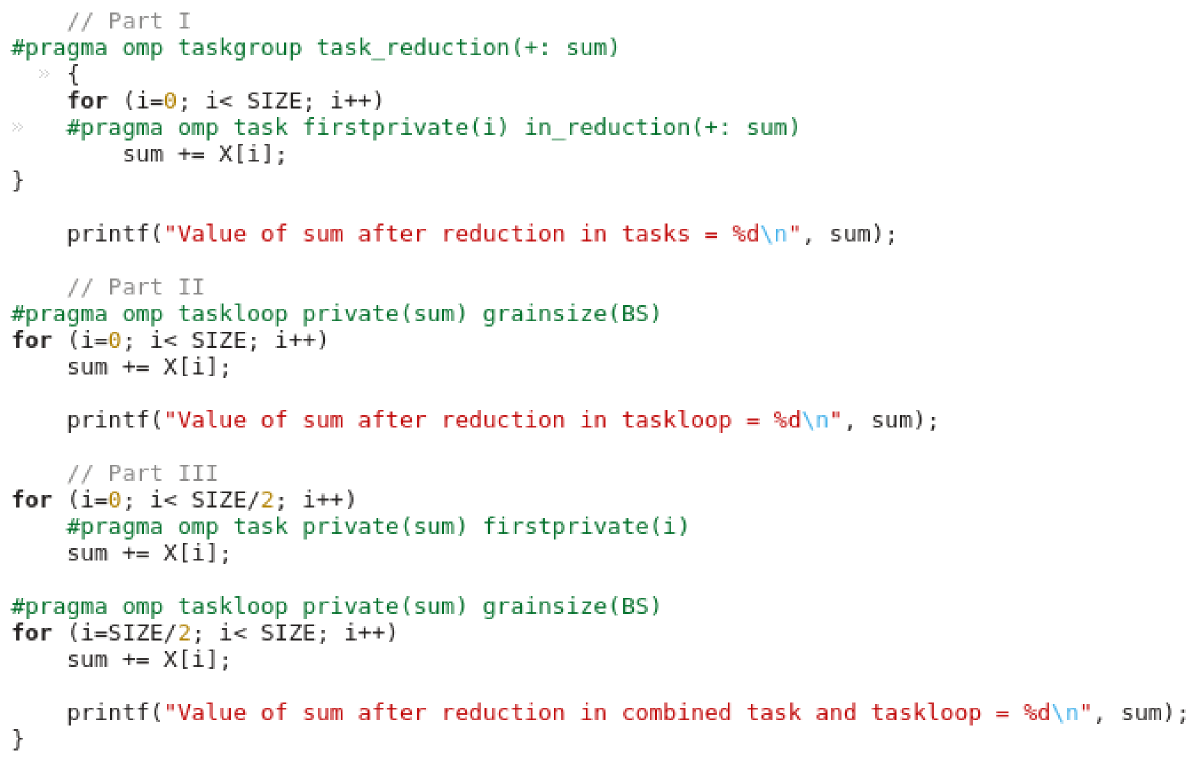
That the loop 1 and loop 2 executes simultaneously, because we override the taskgroup.

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.

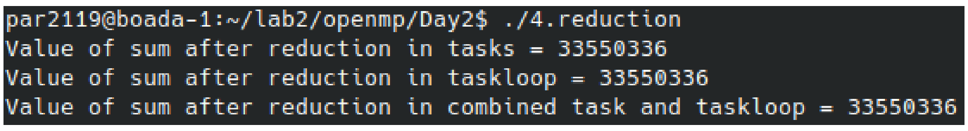
The problem with this code is that the sum result in part 2 and 3 are incorrect because the sum variable is not initialized when it is recalculated.

To solve this problem we add different private(sum) when it is going to be recalculated to initialize the value of the sum variable to 0:



```
    // Part I
#pragma omp taskgroup task_reduction(+: sum)
  » {
    for (i=0; i< SIZE; i++)
»   #pragma omp task firstprivate(i) in_reduction(+: sum)
        sum += X[i];
}

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

    // Part II
#pragma omp taskloop private(sum) grainsize(BS)
for (i=0; i< SIZE; i++)
    sum += X[i];

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

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

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

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

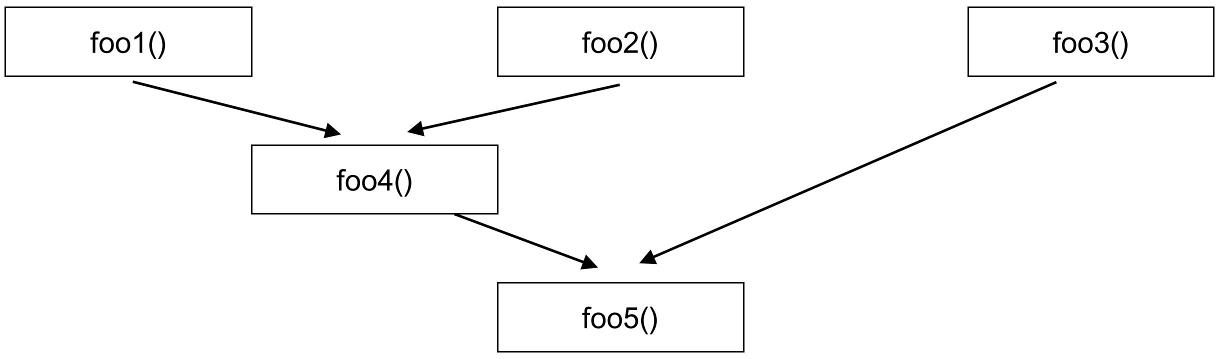
With this modification each printf returns the correct value of sum:



```
par2119@boada-1:~/lab2/openmp/Day2$ ./4.reduction
Value of sum after reduction in tasks = 33550336
Value of sum after reduction in taskloop = 33550336
Value of sum after reduction in combined task and taskloop = 33550336
```

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.

```
    #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();
    printf("Creating task foo4\n");
    #pragma omp taskwait
    foo4();
    printf("Creating task foo5\n");
    #pragma omp taskwait
    foo5();
    }
```

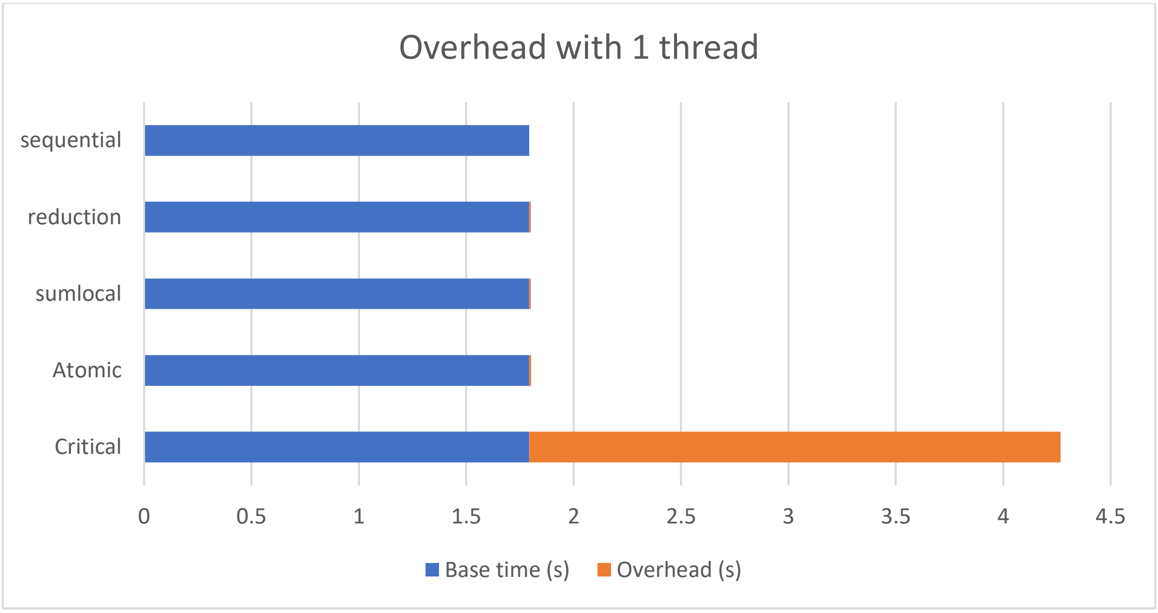
3. Rewrite the program using only taskgroup as task synchronisation mechanism (no depend clauses allowed), again trying to achieve the same potential parallelism that was obtained when using depend.

```
    #pragma omp parallel
    #pragma omp single
    {
    #pragma omp taskgroup
    {
        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 taskgroup
    {
        printf("Creating task foo4\n");
        #pragma omp taskgroup
        foo4();
    }

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

2 Observing overheads

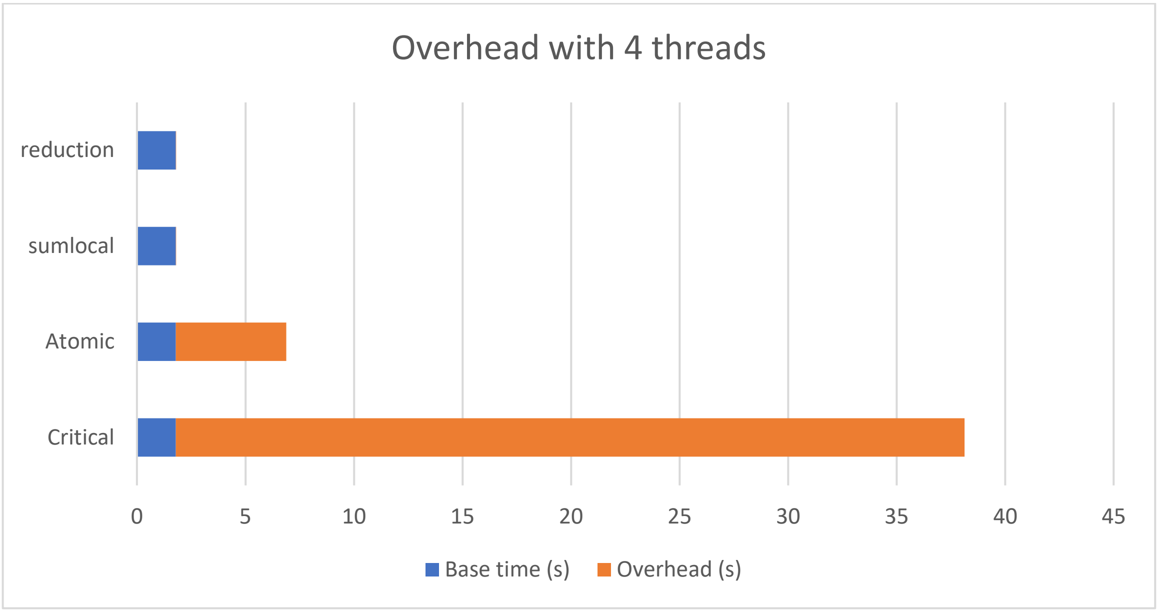
Day 1

1. **If executed with only 1 thread and 100.000.000 iterations, do you notice any major overhead in the execution time caused by the use of the different synchronisation mechanisms? You can compare with the baseline execution time of the sequential version in pi sequential.c.**

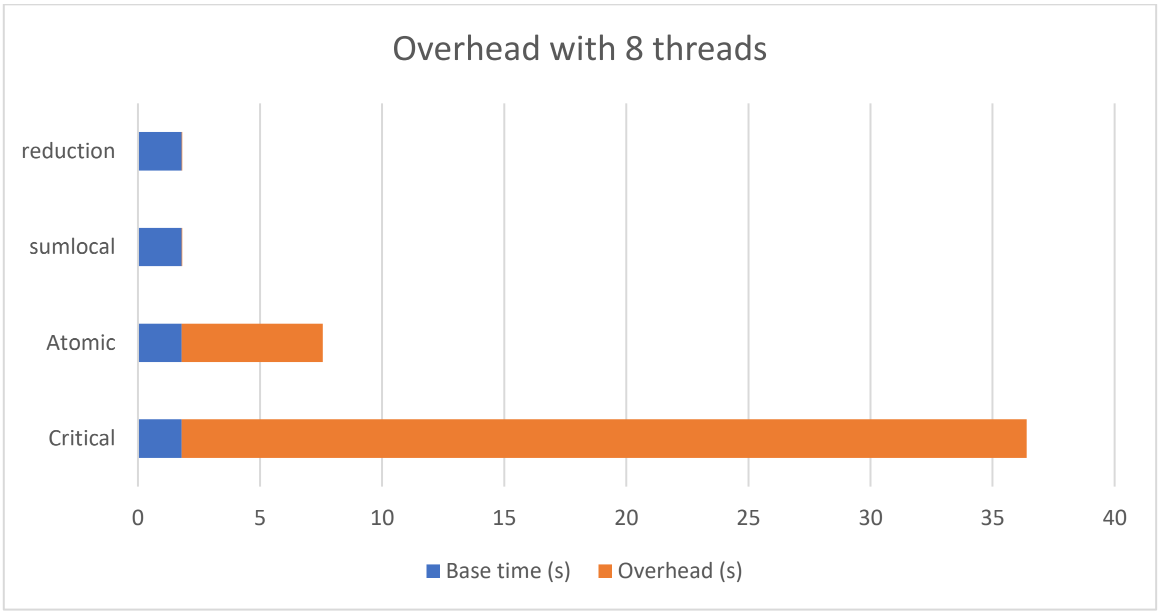


We can notice that the overhead of the *critical* synchronization mechanism is the most inefficient of all four options by far. The other three options have a similar overhead.

2. **If executed with 4 and 8 threads and the same number of iterations, do the 4 programs benefit from the use of several processors in the same way? Can you guess the reason for this behaviour?**



With 4 threads, unlike the previous case, where the *atomic*, *sumlocal* and *reductions* synchronization mechanisms had approximately the same overhead, now we observe that the atomic mechanism grows more that the *sumlocal* and *reduction* synchronizations, but still far from *critical* one.



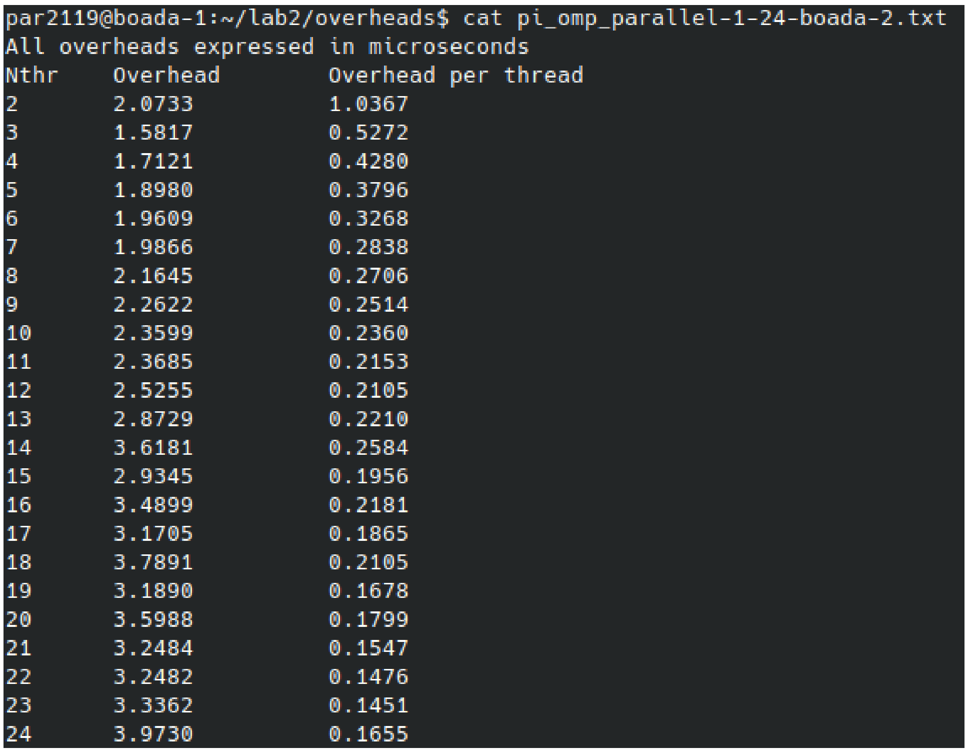
In this case we are left with approximately the same results as the 4 thread executions.

Conclusions

The conclusions are that the *atomic* and *critical* synchronization mechanisms have much more overhead than *sumlocal* and *reduction* synchronizations, the first two options ensure the access to memory positions, as consequence the waits to access these positions and the final synchronization of all the threads will increase the execution time severally. We also notice that the more threads we use for these two cases the higher overhead increases, but as well we observe that the changes between 4 thread and 8 thread execution, for these two cases, it remains approximately the same.

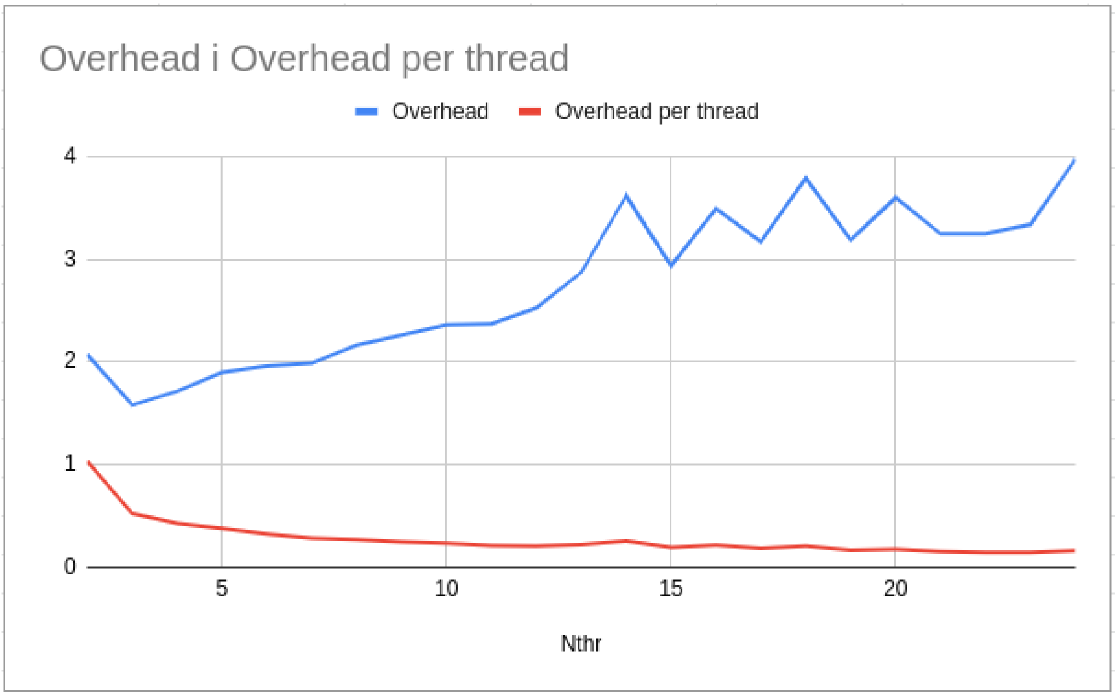
Day 2

Thread creation and termination

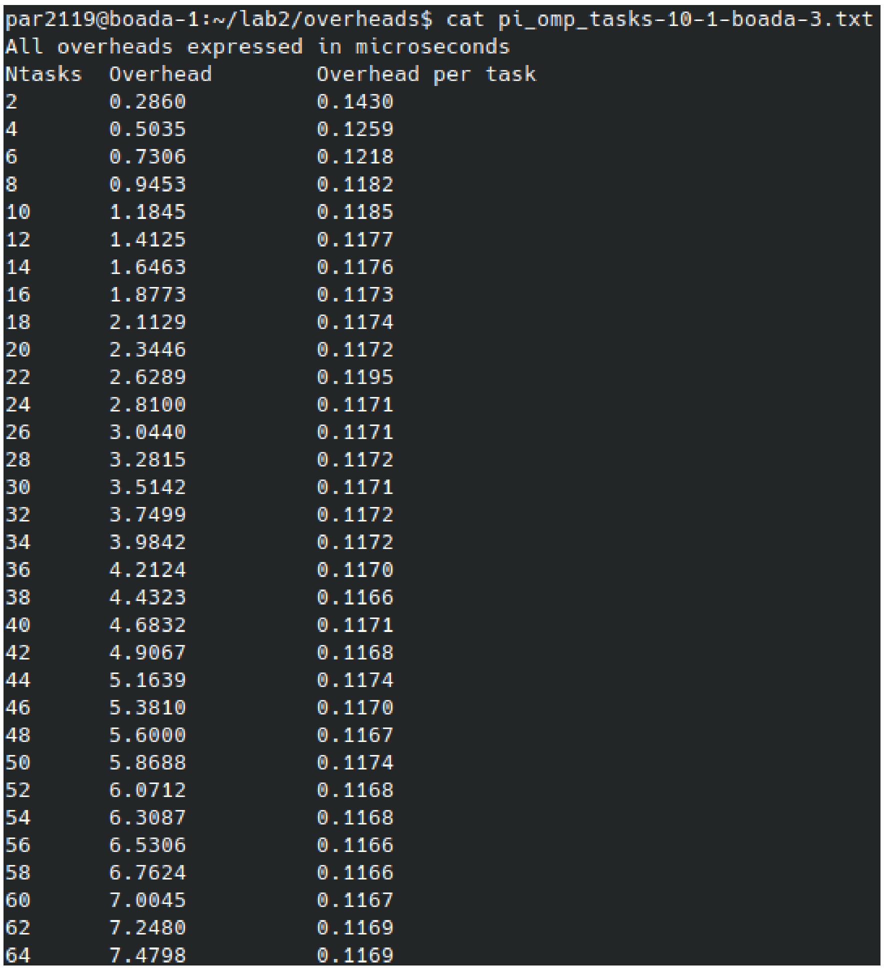


```
par2119@boada-1:~/lab2/overheads$ cat pi_omp_parallel-1-24-boada-2.txt
All overheads expressed in microseconds
Nthr    Overhead        Overhead per thread
2       2.0733          1.0367
3       1.5817          0.5272
4       1.7121          0.4280
5       1.8980          0.3796
6       1.9609          0.3268
7       1.9866          0.2838
8       2.1645          0.2706
9       2.2622          0.2514
10      2.3599          0.2360
11      2.3685          0.2153
12      2.5255          0.2105
13      2.8729          0.2210
14      3.6181          0.2584
15      2.9345          0.1956
16      3.4899          0.2181
17      3.1705          0.1865
18      3.7891          0.2105
19      3.1890          0.1678
20      3.5988          0.1799
21      3.2484          0.1547
22      3.2482          0.1476
23      3.3362          0.1451
24      3.9730          0.1655
```

As we can see, the overhead time per thread decreases as the number of threads to creating / terminating increases. The overhead time increases as we increase the number of threads but the increase is not very significant taking into account that we are talking about microseconds.

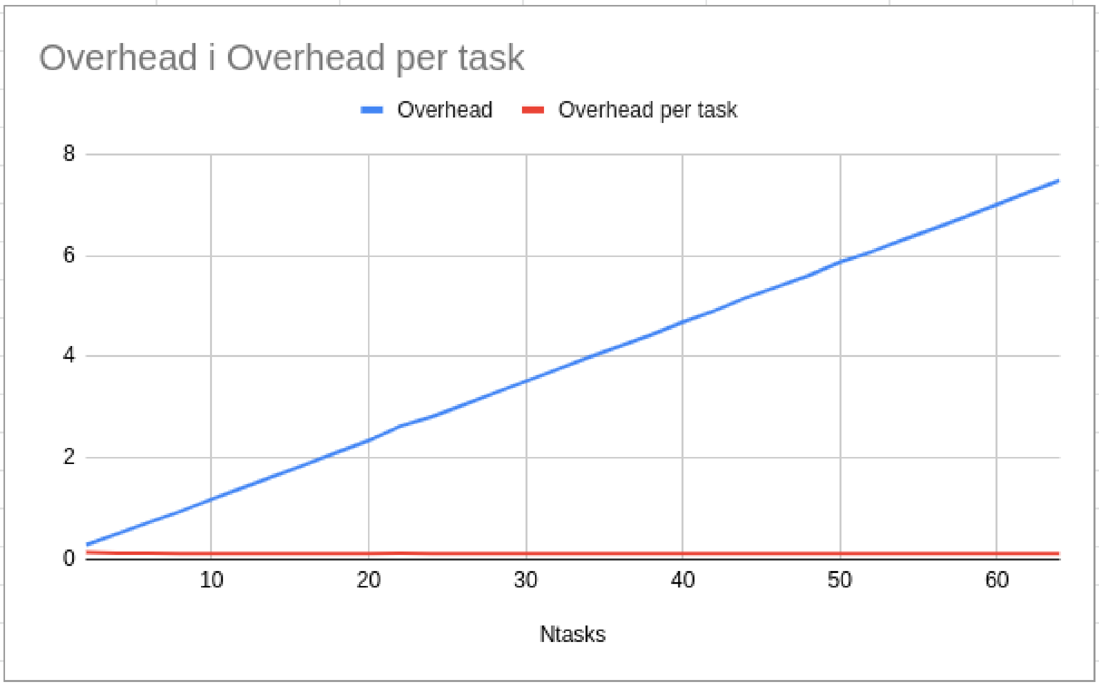


Task creation and synchronisation



```
par2119@boada-1:~/lab2/overheads$ cat pi_omp_tasks-10-1-boada-3.txt
All overheads expressed in microseconds
Ntasks  Overhead        Overhead per task
2       0.2860          0.1430
4       0.5035          0.1259
6       0.7306          0.1218
8       0.9453          0.1182
10      1.1845          0.1185
12      1.4125          0.1177
14      1.6463          0.1176
16      1.8773          0.1173
18      2.1129          0.1174
20      2.3446          0.1172
22      2.6289          0.1195
24      2.8100          0.1171
26      3.0440          0.1171
28      3.2815          0.1172
30      3.5142          0.1171
32      3.7499          0.1172
34      3.9842          0.1172
36      4.2124          0.1170
38      4.4323          0.1166
40      4.6832          0.1171
42      4.9067          0.1168
44      5.1639          0.1174
46      5.3810          0.1170
48      5.6000          0.1167
50      5.8688          0.1174
52      6.0712          0.1168
54      6.3087          0.1168
56      6.5306          0.1166
58      6.7624          0.1166
60      7.0045          0.1167
62      7.2480          0.1169
64      7.4798          0.1169
```

Unlike execution with threads, in this case the overhead of creating/synchronising tasks is linearly proportional to the number of tasks created,



Conclusions

With the implementation of tasks, the balance of overheads remains constant regardless of the threads that the system has. On the other hand, in the implementation with threads, the overhead depends on the number of threads that the system has, so an incorrect assignment of threads can cause a significant increase in overhead.