LABORATORY 2

Group components: Albert Bausili Fernández, Noa-Yu Ventura Vila

Group identifier: par2301

29/09/2022

Index

1. A very practical introduction to OpenMP (I)

1.1 Day 1: Parallel regions and implicit tasks

1.hello.c

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

> We see it 2 times, because there are two threads executing the same code which is a print of “Hello world!”.

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

> OMP_NUM_THREADS=4 ./1.hello makes each thread execute the message, and there are 4 threads so the print of hello is executed 4 times.

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 is not correct, as we see some Thids repeatedly appearing. In order to solve this, we have to add a private(id) clause as shown in the next code snippet extracted from the 2.hello.c file modified by ourselves:

```
…
int main ()
{
   int id;
   #pragma omp parallel private(id) num_threads(8)
   {
     id =omp_get_thread_num();
     printf("(%d) Hello ",id);
     printf("(%d) world!\n",id);
   }
   return 0;
}

…
```

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

> The Thid from the Hello does not always match with the other Thid. Furthermore, sometimes the lines have more (or less) than 2 outputs.
>
> We cannot solve this because as the code is parallel the order is determined by the order of execution established by the SO.

3.how_many.c

Assuming "OMP NUM THREADS=8 ./3.how_many"

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

The function get num threads in the parallel section when invoked outside a parallel section is always 1, because when the code is unparalleled it is executed only by one thread. This is also the reason why the message outside the parallel region is only printed once.

Inside the parallel region the number of active threads is printed. In some cases, we specify in the code how many threads we want to use, otherwise it uses 8 threads because we forced it to be 8 using the clause OMP_NUM_THREADS.

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

2.1 Writing the clause #pragma omp parallel num_threads(N_threads)

```
…
printf("Starting,    I'm    alone    ...    (%d    thread)\n",
omp_get_num_threads());
   #pragma omp parallel
   printf("Hello  world  from  the  first  parallel  (%d)!\n",
omp_get_num_threads());


   #pragma omp parallel num_threads(4)
   printf("Hello  world  from  the  second  parallel  (%d)!\n",
omp_get_num_threads());


   #pragma omp parallel
…
   printf("Outside parallel, nobody else here ... (%d thread)\n",
omp_get_num_threads());


   #pragma omp parallel num_threads(4)
   printf("Hello  world  from  the  fifth  parallel  (%d)!\n",
omp_get_num_threads());


   #pragma omp parallel
   printf("Hello  world  from  the  sixth   parallel  (%d)!\n",
omp_get_num_threads());
…
```

Code extracted from the 3.how_many.c file and modified adding the num_threads(N_threads)

2.2 Using the function omp_set_num_threads(N_threads)

```
…
for (int i=2; i<4; i++) {
        omp_set_num_threads(i);
        #pragma omp parallel
        printf("Hello world from the fourth parallel (%d)!\n",
omp_get_num_threads());
    }
…
```

Code extracted from the 3.how_many.c file adding the omp_set_num_threads(N_threads)

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

3.1 Lasts until the region of code for #pragma omp parallel ends.

3.2 Lasts while the loop is not over.

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).

- Shared: it can be any number between 0-31, because the variable x is shared between all threads, and its value depends on the function omp_get_thread_num. Since we have 32 threads (specified at the beginning of the main function) the possible return values of this function is from 0 to 31.

- Private and firstprivate: it can be only 5 because the result doesn't "go outside", it can't have its value changed outside the pragma omp (first)private clause.

- Reduction: can only be 501, because it's the sum of the partial sums.

5.datarace.c

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

No, because the variable maxvalue suffers dataraces.

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.

- reduction (max:maxvalue): it works because the maximum is a commutative/associative operation, so the partial result can be combined in a complete one.

```
…
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();
…
```

Code extracted from the 5.datarace.c file modified adding a reduction(max:maxvalue) clause

- The code will be slower, but we can also solve the problem by adding a critical clause inside the for loop and outside the if condition.

```
…
for (i=id*N/howmany; i < (id+1)*N/howmany; i++) {
      #pragma omp critical
      {
            if (vector[i] > maxvalue)
            {
                  sleep(1); // this is just to force problems
                  maxvalue = vector[i];
            }
      }
}
…
```

Code extracted from the 5.datarace.c file modified adding a critical clause

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).

```
for (i=id*N/howmany; i < (id+1)*N/howmany; i++)
```

Code extracted from the 5.datarace.c modifying the for iterations

The vector is divided into different parts or equal size and gives each part to a different thread. Thanks to the maximum's properties (explained in question 2) each thread can calculate a partial maximum, and once all of them are finished a thread can calculate the final maximum, giving a correct answer.

6.datarace.c

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

No, because there are data races in countmax.

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.

- Adding: #pragma omp parallel private(i) reduction(+:countmax)

  It is correct because the attribute i is kept out of dataraces with the private clause so the execution of the fragmented loop is done without dataraces thus there is no problem in summing all the partial results in the countmax variable.

```
…
int i, countmax = 0;
int maxvalue = 15;


omp_set_num_threads(8);
#pragma omp parallel private(i) reduction(+:countmax)
{
      int id = omp_get_thread_num();
      int howmany = omp_get_num_threads();
…
```

Code extracted from the 6.datarace.c with the aforementioned clause added

- Inside the if condition and before the sentence countmax++ we need to erite #pragma omp atomic

  It is correct because the atomic clause ensures all the threads have to modify the countmax value in a "sequential way" thus no dataraces occur.

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

Code extracted from the 6.datarace.c adding the #pragma omp atomic clause

1.2 Day 1: Synchronization overheads

1.2.1 Results

Take a look at the four different versions and make sure you understand them. How many synchronisation operations (critical or atomic) are executed in each version?

pi_omp_critical.c: Executes 4 times the critical operation.

pi_omp_atomic.c: Executes 4 times the atomic operation.

pi_omp_sumlocal.c: Executes 4 times the critical operation. Although the section is smaller than in the first one.

pi_omp_reduction.c: Does not execute any critical/atomic operation.

Compile all four versions (use the appropriate entries in the Makefile) and queue the execution of the binaries generated using the submit-omp.sh script (which requires the name of the binary, the number of iterations for Pi computation and the number of threads). Answer the following questions:

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.

   There's no overhead when the code is executed sequentially, since it's executed in only one thread. Critical has some overheads because it downloads external code from libraries, but our code is executed sequentially. We can notice these characteristics in the following table.

Version	Time overhead (microseconds)
critical	1862184.0000
atomic	-4178.0000
sum_local	-3949.0000
reduction	-1916.0000

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?

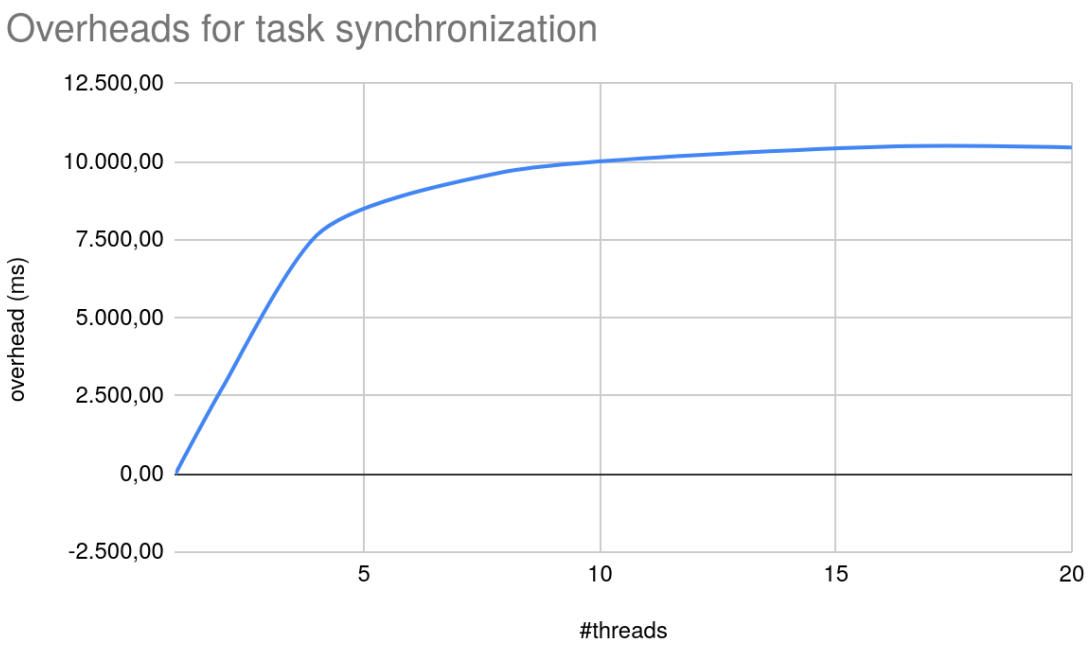
   Executing the code with 4 and 8 threads the time overhead is higher, because it takes a lot of time to synchronize all of them. Check the next table to see that now there aren't any negative values of time overhead and notice that the time overhead of 8 threads is greater than using 4 threads.

Version	#threads	Time overhead (microseconds)
critical	4	50401203.7500
	8	43830557.8750
atomic	4	7649572.5000
	8	9683089.5000
sum_local	4	10273.0000
	8	9683089.5000
reduction	4	10932.5000
	8	15398.6250

1.2.2 Conclusions

Overall, the executions using one thread have been the ones with a better performance due to the huge synchronization times of the parallel versions, either they are critical, atomic, sum_local or reduction. It's also because when there are too many tasks the size of each task is very small. In the following graph we can see that the correlation between the number of threads used and the overhead times of program pi_omp_atomic.c proves our conclusions for 100.000.000 iterations.

#threads	overhead (ms)
1	-4,18
2	2.766,50
4	7.649,57
8	9.683,09
16	10.480,80
20	10.457,31



2. A very practical introduction to OpenMP (II)

2.1 Day 2: Explicit tasks

1.single.c

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

All threads need to execute a loop the same number of times. In this case there are 4 threads and 20 iterations, which means that each thread will execute 5 iterations. A thread does one iteration and lets the other 3 do their iteration, then it can start the second iteration (which hasn't been done before) and let the others finish their second iteration.

Nowait makes this possible, because threads don't need to wait for the others to finish their iterations, so thread 1 can execute code without waiting for thread 0 to execute this code (which would be the case for single without nowait clause).

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

Because this code is parallelizable, threads don't depend on anything else to execute their iterations. They appear to be executed in bursts because each thread starts one iteration at the same time (there are 4 iterations executing) and, since they take more or less the same time to be executed, they finish more or less at the same time. The same happens in the next iteration.

2.fibtasks.c

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

Because the code depends on the value obtained the previous iteration. Fibonacci numbers are calculated using the previous obtained values, that's why it cannot be parallelized. Moreover, the #pragma omp task sentence creates tasks but doesn't parallelize them.

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

On the following page we have the code modified 2.fibtasks.c that works properly.

```
…
int main(int argc, char *argv[]) {
    struct node *temp, *head;


    omp_set_num_threads(6);
    printf("Staring computation of Fibonacci for numbers in linked
list \n");


    p = init_list(N);
    head = p;


    #pragma omp parallel
    #pragma omp single
```

```
    {
       while (p != NULL) {
                printf("Thread %d creating task that will
compute %d\n", omp_get_thread_num(), p->data);
                #pragma omp task firstprivate(p)
                processwork(p);
                p = p->next;
         }
    }
    printf("Finished creation of tasks to compute the Fibonacci
for numbers in linked list \n");


    printf("Finished computation of Fibonacci for numbers in
linked list \n");
    p = head;
    while (p != NULL) {
       printf("%d: %d computed by thread %d \n", p->data, p-
>fibdata, p->threadnum);
       temp = p->next;
       free (p);
       p = temp;
    }
    free (p);


    return 0;
}
```

Code extracted from the 2.fibtask.c modified with the correct clauses

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

If we comment the firstprivate(p) clause all tasks are executed by the same thread, but before tasks were randomly distributed among the threads.

The function of firstprivate is to keep the initial value of p before executing processwork(). This is why we know that the statement p->pnext can be executed in one thread while the function processwork() is being executed in another thread, there is parallelization, because the value of p won't change inside the function.

3.taskloop.c

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

The clause grainsize(N) doesn't mean there will be N iterations, it means that there can be N <= iterations < 2N for each task.

Some threads execute a block of iterations greater or equal than grainsize. In this case grainsize is 4, so the number of iterations that will be executed by the same thread is 4. This is because one thread (usually thread 0) assigns the grainsize of a task (4 iterations) to a random thread that isn't executing anything, and it's possible that it assigns a task

to the same thread consecutively more than once. For example, thread 1 may execute iterations 4 to 11 and thread 3 iterations 0 to 3.

The clause num_tasks(N) doesn't mean there will be N tasks, it means that there can be N <= tasks, it doesn't matter how many iterations a task has.

There will be num_tasks threads executing a series of iterations. In this case, num_tasks is equal to 4, and there are 12 iterations, so the number of threads needed is 4, and each one will execute 12/4=3 iterations. Following the previous case, the thread assigned to do that task might be the same one assigned to previous iterations, so we won't always have 3 different threads executing 3 iterations, but the amount of iterations a thread executes will always be greater or equal than 3.

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

Now, for grainsize 5 each thread executes 5 or more iterations, because we have 12/5=2.4 ≈ 2 tasks, therefore we get 2 tasks to do the 12 iterations → 6 iterations for each task. That's why there are always only two threads executing something when grainsize is 5.

We have 12 iterations and we want to at least have 5 tasks, so we get 12/5 = 2.4 ≈ 2 iterations for each task and 12-2*5 = 2 iterations → two tasks will execute 3 iterations and three tasks will execute 2 iterations.

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

No, they can't. Plus it wouldn't make any sense, since they have different organizations to do the iterations.

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

We see that the output from grainsize and num_tasks are mixed up, which means that the for loop is only executed once, and both num tasks and grainsize codes are executed at the same time.

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.

In the next page the code that computes the Gauss sum correctly using the three methods.

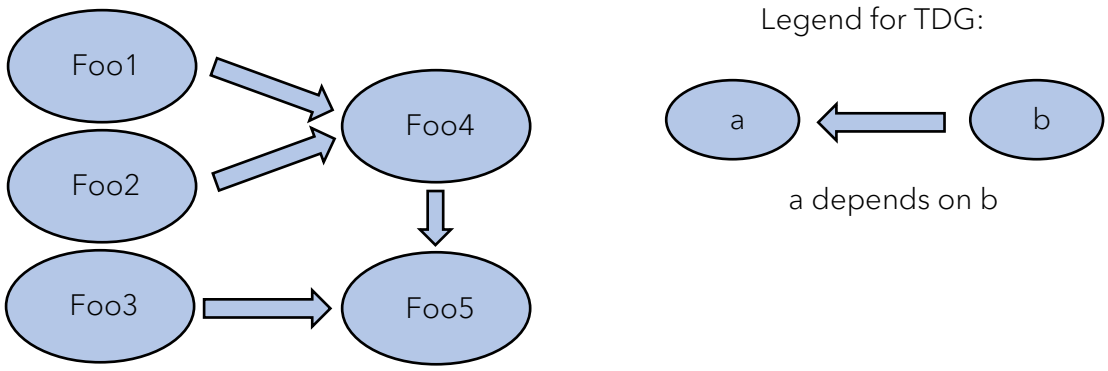
```
…
int main()
{
   int i;


   for (i=0; i<SIZE; i++)
       X[i] = i;
```

```
   omp_set_num_threads(4);
   #pragma omp parallel
   #pragma omp single
   {
       // 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 grainsize(BS) firstprivate(sum)
       for (i=0; i< SIZE; i++)
           sum += X[i];


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


       // Part III
       #pragma omp taskgroup task_reduction(+: sum)
       {
               #pragma      omp      taskloop      grainsize(BS)
firstprivate(sum)
               for (i=0; i< SIZE/2; i++)
               sum += X[i];
       }


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


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


   return 0;
}
```

Code extracted from the 4.reduction.c adding the appropriate clauses

5.synchtasks.c

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



Task Dependency Graph (TDG) of program 5.synchtask.c

foo1, foo2 and foo3 don't depend on anyone, because the clause depend(out) means they're only writing a value into the variable, but they don't need to verify their previous value is correct.

foo4 depends on foo1 and foo2 due to the clause depend(in). We are telling the processor this function will read the value for variables a and b, so foo4 needs to wait for foo1 and foo2 to finish writing the value to the variables a and b to ensure their values are correct, and then read them.

Using the same logic, we know that foo5 depends on foo3 and foo4, because these two functions modify the value of variables c and d.

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

```
…
int main(int argc, char *argv[]) {

   #pragma omp parallel
   #pragma omp single
   {
        printf("Creating task foo1\n");
        #pragma omp task
        foo1();
        printf("Creating task foo2\n");
        #pragma omp task
        foo2();
        printf("Creating task foo4\n");
        #pragma omp taskwait
        foo4();
        printf("Creating task foo3\n");
```

```
        #pragma omp task
        foo3();
        printf("Creating task foo5\n");
        #pragma omp taskwait
        foo5();
    }
    return 0;
```

Region of code that we changed from file 5.synchtask.c using the clause taskwait instead of using dependecies

Before the functions that depend on another function we write a taskwait clause, this way we guarantee that the functions it depends on have finished their execution and the variables' values are correct.

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.

In this case we are told to use the taskgroup clause. In order to make it function the same way, we need to group as many tasks as possible until we find a task that has a dependency, and this task will belong to another taskgroup. This way, a group of functions can finish whenever they want knowing that the variables' values are correct. Once they finish, the next taskgroup has a function that depends on a function located inside the previous taskgroup, that's why we separate them. This means there are no dependencies between functions within the same taskgroup.

```
…
int main(int argc, char *argv[]) {

   #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();
       }
       #pragma omp taskgroup
       {
               printf("Creating task foo4\n");
               #pragma omp task
               foo4();
               printf("Creating task foo3\n");
               #pragma omp task
               foo3();
       }
       printf("Creating task foo5\n");
       #pragma omp task
       foo5();
   }
   return 0;
}
```

Region of code that we changed from file 5.synchtask.c without using dependencies or task wait clauses

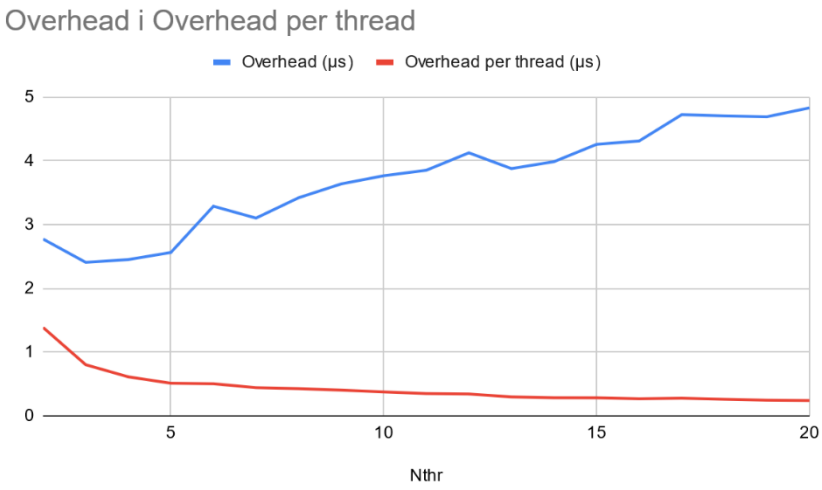
2.2 Day 2: Observing overheads

Please explain in this section of your deliverable the main results obtained and your conclusions in terms of overheads for parallel tasks and the different synchronization mechanisms. Include any tables/plots that support your conclusions.

2.2.1 Results

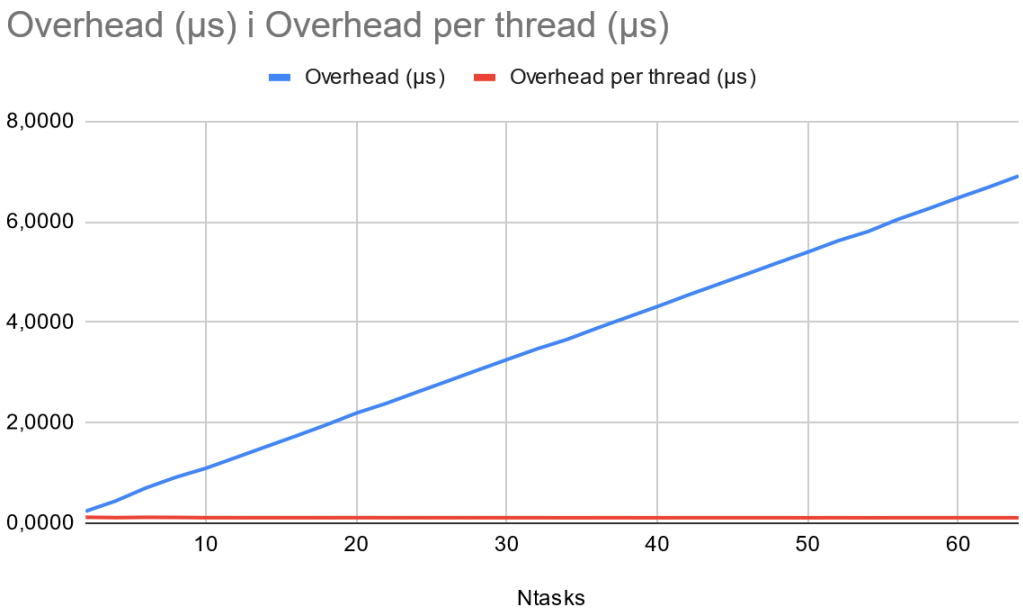
The overhead of creating/terminating threads increases proportionally to the number of threads created, while the overhead per thread reduces inversely.

The order of magnitude of creating/terminating an individual thread is around the microsecond.



Nthr	Overhead (µs)	Overhead per thread (µs)
2	2,7736	1,3868
3	2,4087	0,8029
4	2,4520	0,6130
5	2,5634	0,5127
6	3,2890	0,5048
7	3,1026	0,4432
8	3,4214	0,4277
9	3,6386	0,4043
10	3,7668	0,3767
11	3,8529	0,3503
12	4,1264	0,3439
13	3,8790	0,2984
14	3,9877	0,2848
15	4,2594	0,2840
16	4,3120	0,2695
17	4,7251	0,2779
18	4,7053	0,2614
19	4,6917	0,2469
20	4,8314	0,2416

Ntasks	Overhead (μs)	Overhead per thread (μs)
2	0,2363	0,1182
4	0,4412	0,1103
6	0,6995	0,1166
8	0,9144	0,1143
10	1,0939	0,1094
12	1,3078	0,1090
14	1,5231	0,1088
16	1,7377	0,1086
18	1,9578	0,1088
20	2,1922	0,1096
22	2,3861	0,1085
24	2,6065	0,1086
26	2,8222	0,1085
28	3,0411	0,1086
30	3,2562	0,1085
32	3,4697	0,1084
34	3,6585	0,1076
36	3,8840	0,1079
38	4,0990	0,1079
40	4,3132	0,1078
42	4,5380	0,1080
44	4,7525	0,1080
46	4,9669	0,1080
48	5,1854	0,1081
50	5,3986	0,1075
52	5,6218	0,1081
54	5,8074	0,1075
56	6,0514	0,1081
58	6,2597	0,1079
60	6,4801	0,1080
62	6,6890	0,1079
64	6,9119	0,1080



The overhead of creating/terminating tasks increases lineally to the number of tasks created, while the overhead per thread keeps almost constant.

The order of magnitude of creating/terminating an individual tasks is around the microsecond.

2.2.2 Conclusions

Summing up, when parallelizing a code, a huge overhead is created by the addition of little overheads of each individual thread. This explains the reduction in overhead per thread. Whereas the execution has no relation to the task number as it's always constant. This difference can be explained because when creating tasks there is no synchronization nor scheduling done thus the overhead per core stays the same.