DELIVERABLE
LABORATORY 2
OpenMP

Adrià Redondo
Miguel Gutiérrez Jariod
Group 23
24/03/2021 - Q2021-2022

Index:

OpenMP questionnaire

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

1. We see that the output “Hello World” message appears twice.
2. Increasing the number of threads to 4 for the interactive part so we will see that prints the message four 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?
2. Are the lines always printed in the same order? Why the messages sometimes appear intermixed? (Execute several times in order to see this).

```
par2311@boada-1:~/lab2/openmp/Day1$ ./2.hello
(0) Hello (2) Hello (2) world!
(4) Hello (2) world!
(5) Hello (5) world!
(2) world!
(1) Hello (5) world!
(6) Hello (6) world!
(3) Hello (3) world!
(7) Hello (7) world!
par2311@boada-1:~/lab2/openmp/Day1$
```

(2.1 v1: 2.hello)

1. As we can see in the image 2.1 the execution of the program is not correct because it doesn’t print the format that we are speacting. It is because of the data race.

```
par2311@boada-1:~/lab2/openmp/Day1$ ./2.hello
(0) Hello (3) Hello (3) world!
(4) Hello (4) world!
(6) Hello (6) world!
(2) Hello (2) world!
(5) Hello (5) world!
(1) Hello (1) world!
(0) world!
(7) Hello (7) world!
par2311@boada-1:~/lab2/openmp/Day1$
```

(2.2 v2: 2.hello)

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

(2.3 v2: code improved to solve data race)

2. No, they aren't printed in the same order. It's because it depends on the first thread that proceeds to do the task and it changes every time. It appears intermixed because a task can be executed in a lower term than another that has begun later (Figure 2.2)

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?
2. Indicate the two alternatives to supersede the number of threads that is specified by the `OMP_NUM_THREADS` environment variable.
3. Which is the lifespan for each way of defining the number of threads to be used?

1. When omp_get_num_threads is invoked in the outside region returns a message of the situation (like the starting and the beginning) and the unique number of one thread, while if is in an inside region returns a message of the different parallel regions and the number of threads in that proceed on each part of it.

```
par2311@boada-1:~/lab2/openmp/Day1$ ./3.how_many
Starting, I'm alone ... (1 thread)
Hello world from the first parallel (2)!
Hello world from the first parallel (2)!
Hello world from the szecond parallel (4)!
Hello world from the szecond parallel (4)!
Hello world from the szecond parallel (4)!
Hello world from the szecond parallel (4)!
Hello world from the third parallel (2)!
Hello world from the third parallel (2)!
Hello world from the fourth parallel (2)!
Hello world from the fourth parallel (2)!
Hello world from the fourth parallel (3)!
Hello world from the fourth parallel (3)!
Hello world from the fourth parallel (3)!
Outside parallel, nobody else here ... (1 thread)
Hello world from the fifth parallel (4)!
Hello world from the fifth parallel (4)!
Hello world from the fifth parallel (4)!
Hello world from the fifth parallel (4)!
Hello world from the sixth  parallel (3)!
Hello world from the sixth  parallel (3)!
Hello world from the sixth  parallel (3)!
Finishing, I'm alone again ... (1 thread)
```

(Figure 3.1)

2. The 2 alternatives are:
    - omp_set_num_threads(i)
    - #pragma omp parallel num_threads(i)

3. For each parallel region are executed a number of threads and when they are finished they stay on a barrier established by the default proceed of OpenMP, so it determines the lifespan.

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)

```
par2311@boada-1:~/lab2/openmp/Day1$ ./4.data_sharing
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
par2311@boada-1:~/lab2/openmp/Day1$ ./4.data_sharing
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
par2311@boada-1:~/lab2/openmp/Day1$ ./4.data_sharing
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
par2311@boada-1:~/lab2/openmp/Day1$ ./4.data_sharing
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
par2311@boada-1:~/lab2/openmp/Day1$
```

(Figure 4.1)

Later the first parallel (shared), the value of x is 120.

Later the second parallel (privative), the value of x is 5.

Later the third parallel (firstprivate), the value of x is 5.

Later the fourth parallel (reduction), the value of x is 125.

We can expect the values of x for the private and firstprivate, because it is the value of x before the parallel region. But we couldn't expect what would be the result for the shared ones. (Figure 4.1)

5.datarace.c

1. Is the program executing correctly? Why?
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.
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.

1. This program is not executing correctly because the maxvalue variable can generate data race. It's important to keep control to solve this problem to make the process successful, because in the executions the output seems to be good while in reality it doesn't pay attention to data race. There can be possible times that the program doesn't obtain the correct maxvalue.

```c
#define N 1 << 20
int vector[N]={0, 0, 0, 1, 2, 3, 4, 5, 6, 7, 15, 14, 13, 12, 11, 10, 9, 8, 15, 15};

int main()
{
    int i, maxvalue=0;

    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)
           maxvalue = vector[i];
    }
    }

    if (maxvalue==15)
        printf("Program executed correctly - maxvalue=%d found\n", maxvalue);
    else printf("Sorry, something went wrong - incorrect maxvalue=%d found\n", maxvalue);

    return 0;
}
```

(5.1 v1:5.datarace.c code)

2. Two alternative solutions: (Figure 5.2 and 5.3)

```c
#define N 1 << 20
int vector[N]={0, 0, 0, 1, 2, 3, 4, 5, 6, 7, 15, 14, 13, 12,

int main()
{
    int i, maxvalue=0;

    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();

    for (i=id; i < N; i+=howmany) {
        if (vector[i] > maxvalue)
           maxvalue = vector[i];
    }
    }

    if (maxvalue==15)
        printf("Program executed correctly - maxvalue=%d fou
    else printf("Sorry, something went wrong - incorrect maxv

    return 0;
}
```

In our first version (5.2) we have added a reduction to the variable maxvalue so when all threads finish we will make the join by the operand "max" (maximum). Also we privatize the maxvalue variable because it is made by default.

```
#define N 1 << 20
int vector[N]={0, 0, 0, 1, 2, 3, 4, 5, 6, 7, 15, 14, 13, 12, 11

int main()
{
    int i, maxvalue=0;

    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) {
        #pragma omp critial
        if (vector[i] > maxvalue)
           maxvalue = vector[i];
    }
    }

    if (maxvalue==15)
         printf("Program executed correctly - maxvalue=%d found
    else printf("Sorry, something went wrong - incorrect maxval

    return 0;
}
```

In our second (5.3) version we have decided to add a critical to keep only a thread on the region. There is a problem that can happen to the maxvalue. It is that two threads exclude the correct max value, so to solve it we would add a second condition (the same of the if) to make sure that runs properly.

3. Alternative N/threads, we create a taskloop manually:

```
#define N 1 << 20
int vector[N]={0, 0, 0, 1, 2, 3, 4, 5, 6, 7, 15, 14, 13, 12, 11

int main()
{
    int i, maxvalue=0;

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

    int chunk = N/howmany;

    for (i=id; i < N; i+=chunk) {
        #pragma omp task
        for (int j = i; j < N; ++j){
            if (vector[i] > maxvalue)
               maxvalue = vector[i];
        }
    }
    }

    if (maxvalue==15)
         printf("Program executed correctly - maxvalue=%d found
    else printf("Sorry, something went wrong - incorrect maxval

    return 0;
}
```

(Figure 5.4)

In this part (5.4) we create the chunk variable to make possible the N/threads. Later we have created a taskloop manually to make correct the access of each chunk and obtain the proper result.

6.datarace.c

1. Is the program executing correctly? Why?
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.

```c
#define N 1 << 20
int vector[N]={0, 0, 0, 1, 2, 3, 4, 5, 6, 7, 15, 14, 13, 12, 11, 10, 9, 8, 15, 15};

int main()
{
    int i, countmax = 0;
    int maxvalue = 15;

    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)
            countmax++;
        }
    }

    if (countmax==3)
         printf("Program executed correctly - maxvalue=%d found %d times\n", maxvalue, count
    else printf("Sorry, something went wrong - incorrect maxvalue=%d found %d times\n", maxv

    return 0;
}
```

(6.1 v1:6.datarace.c code)

1. This program is not executing properly because it can develop some problems with data race in the countmax variable, which count the number of times that the maximum appears, so to solve the problem we have decided to make these solutions that are shown below.

2. Two alternative solutions: (Figure 6.1 and 6.2)

```c
#define N 1 << 20
int vector[N]={0, 0, 0, 1, 2, 3, 4, 5, 6, 7, 15, 14, 13, 1

int main()
{
    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();

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

    if (countmax==3)
         printf("Program executed correctly - maxvalue=%d
    else printf("Sorry, something went wrong - incorrect m

    return 0;
}
```

In our first solution (6.1) we have added a reduction to the variable countmax so when all the threads finish we will make the join with the operand sum. Also we privatize the countmax variable because it is made by default.

```c
#define N 1 << 20
int vector[N]={0, 0, 0, 1, 2, 3, 4, 5, 6, 7, 15, 14, 13, 1

int main()
{
    int i, countmax = 0;
    int maxvalue = 15;

    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++;
        }
    }

    if (countmax==3)
         printf("Program executed correctly - maxvalue=%d
    else printf("Sorry, something went wrong - incorrect ma

    return 0;
}
```

In our second solution (6.2), keeping in mind that we can't use criticals for this version, we have decided to use the atomic to the operation of sum of the countmax, so we solve the problem of the conflict of the threads in this part instead of giving a possible bad result.

7.datarace.c

1. Is the program executing correctly? If not, explain why it is not providing the correct result for one or the two variables (`countmax` and `maxvalue`)
2. Write a correct way to synchronise the execution of implicit tasks (threads) for this program.

```c
#define N 1 << 20
int vector[N]={0, 0, 0, 1, 2, 3, 4, 5, 6, 7, 15, 14, 13, 12, 11, 10, 9, 8, 15, 15};

int main()
{
    int i, maxvalue=0;
    int countmax = 0;

    omp_set_num_threads(8);
    #pragma omp parallel private(i) reduction(+: countmax) 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) countmax++;
        if (vector[i] > maxvalue) {
           maxvalue = vector[i];
           countmax = 1;
        }
    }
    }

    if ((maxvalue==15) && (countmax==3))
         printf("Program executed correctly - maxvalue=%d found %d times\n", maxvalue
    else printf("Sorry, something went wrong - maxvalue=%d found %d times\n", maxvalu

    return 0;
```

(7.1 v1: 7.datarace.c code)

```
par2311@boada-1:~/lab2/openmp/Day1$ ./7.datarace
Sorry, something went wrong - maxvalue=15 found 9 times
```

(Figure 7.2)

1. As we observe in the image 7.2 the program is wrong. It is because we can't make a reduction of both variables. We can't obtain the maxvalue and the countmax at the same time so that's why one of the variables is wrong. It is because of the reduction and the modification of the variables with the different threads at the same time.

2. Correct way to synchronize:

```c
#define N 1 << 20
int vector[N]={0, 0, 0, 1, 2, 3, 4, 5, 6, 7, 15, 14, 13, 12, 11, 10, 9, 8, 15, 15};

int main()
{
    int i, maxvalue=0;
    int countmax = 0;

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

        //privatizamos con variables el maximo valor i el
        //contador de maximos i las anteriores creadas
        //se quedarian como shared
        int maxvalueP = 0;
        int countmaxP = 0;

        for (i=id; i < N; i+=howmany) {
            if (vector[i]==maxvalueP) countmaxP++;
            if (vector[i] > maxvalueP) {
               maxvalueP = vector[i];
               countmaxP = 1;
            }
        }
        //una vez acaaba el bucle falta recoger los valores
        //adoptados por las variables privadas para mostrar
        //el resultado correcto en las privadas
        #pragma omp critical
        {
            printf("thread %i in critical\n", id);
            if (maxvalueP >= maxvalue){
                maxvalue = maxvalueP;
                countmax = countmaxP;
            }
            else if (maxvalueP == maxvalue){
                countmax += countmaxP;
            }
        }
    }

    if ((maxvalue==15) && (countmax==3))
         printf("Program executed correctly - maxvalue=%d found %d times\n", maxva
    else printf("Sorry, something went wrong - maxvalue=%d found %d times\n", maxv
```

(Figure 7.3)

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?

```c
int main ()
{
    int myid;
    #pragma omp parallel private(myid) num_threads(4)
    {
        int sleeptime;

    myid=omp_get_thread_num();
        sleeptime=(2+myid*3)*1000;

    printf("(%d) going to sleep for %d milliseconds ...\n",myid,sleeptime);
    usleep(sleeptime);
    printf("(%d) wakes up and enters barrier ...\n",myid);
    #pragma omp barrier
    printf("(%d) We are all awake!\n",myid);
    }
    return 0;
}
```

1. In this case we can predict the first and the second printf because the threads will show us their sleeptime in order. But when they arrive at the barrier we can't predict how the printf would be because it executes in a different way. When the threads got out of the barrier they didn't have an order so that's the reason why we don't know the execution order.

```
par2311@boada-1:~/lab2/openmp/Day1$ ./8.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 ...
(3) wakes up and enters barrier ...
(2) wakes up and enters barrier ...
(0) We are all awake!
(2) We are all awake!
(1) We are all awake!
(3) We are all awake!
par2311@boada-1:~/lab2/openmp/Day1$ ./8.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 ...
(3) wakes up and enters barrier ...
(2) wakes up and enters barrier ...
(1) We are all awake!
(3) We are all awake!
(0) We are all awake!
(2) We are all awake!
par2311@boada-1:~/lab2/openmp/Day1$
```

(Figure 8.1)

In this image 8.1 we can appreciate the different reasons explained above.

Observing overheads:

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 observe 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`.
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?

Take note of all the results that you obtain and reach your conclusions about the overheads associated with these `OpenMP` constructs. Can you quantify (in microseconds) the cost of each individual synchronisation operation (`critical` or `atomic`) that is used?. You will have to write your conclusions in the appropriate section of the deliverable for this second laboratory assignment.

Executions of the different programs:

```
par2311@boada-1:~/lab2/overheads$ ./pi_omp_critical 100000000 1
Total overhead when executed with 100000000 iterations on 1 threads: 2494074.0000 microseconds
par2311@boada-1:~/lab2/overheads$ ./pi_omp_critical 100000000 4
Total overhead when executed with 100000000 iterations on 4 threads: 149281287.5000 microseconds
par2311@boada-1:~/lab2/overheads$ ./pi_omp_critical 100000000 8
Total overhead when executed with 100000000 iterations on 8 threads: 280050764.6250 microseconds
```

(pi_omp_critical <num_steps> <n_threads>)

```
par2311@boada-1:~/lab2/overheads$ ./pi_omp_atomic 100000000 1
Total overhead when executed with 100000000 iterations on 1 threads: -4147.0000 microseconds
par2311@boada-1:~/lab2/overheads$ ./pi_omp_atomic 100000000 4
Total overhead when executed with 100000000 iterations on 4 threads: 1762808.0000 microseconds
par2311@boada-1:~/lab2/overheads$ ./pi_omp_atomic 100000000 8
Total overhead when executed with 100000000 iterations on 8 threads: 1993082.2500 microseconds
```

(pi_omp_atomic <num_steps> <n_threads>)
For the first execution we did it again and we obtained 2.319 ms.

```
par2311@boada-1:~/lab2/overheads$ ./pi_omp_sumlocal 100000000 1
Total overhead when executed with 100000000 iterations on 1 threads: -1223.0000 microseconds
par2311@boada-1:~/lab2/overheads$ ./pi_omp_sumlocal 100000000 4
Total overhead when executed with 100000000 iterations on 4 threads: 530945.0000 microseconds
par2311@boada-1:~/lab2/overheads$ ./pi_omp_sumlocal 100000000 8
Total overhead when executed with 100000000 iterations on 8 threads: 775762.7500 microseconds
```

(pi_omp_sumlocal <num_steps> <n_threads>)
For the first execution we did it again and we obtained 2.592 ms.

```
par2311@boada-1:~/lab2/overheads$ ./pi_omp_reduction 100000000 1
Total overhead when executed with 100000000 iterations on 1 threads: -16839.0000 microseconds
par2311@boada-1:~/lab2/overheads$ ./pi_omp_reduction 100000000 4
Total overhead when executed with 100000000 iterations on 4 threads: 543733.7500 microseconds
par2311@boada-1:~/lab2/overheads$ ./pi_omp_reduction 100000000 8
Total overhead when executed with 100000000 iterations on 8 threads: 791185.1250 microseconds
```

(pi_omp_reduction <num_steps> <n_threads>)
For the first execution we did it again and we obtained 188 ms.

100000000 iterations	Critical (microS)	Atomic (microS)	Sumlocal (microS)	Reduction (microS)
1 thread	2.494.074	2.319	2.592	188
4 threads	149.281.287,5	1.762.808	530.945	543.733,75
8 threads	280.050.764,625	1.993.082,25	775.762,75	791.185,125

As we can appreciate on the table the sumlocal program is the fastest one when we apply 4 and 8 threads, but when it is just one the reduction in this case is faster. Moreover if we increase the time, the next program in general would be the reduction. Later on we should add the atomic method, but in this case we can see that with one thread it goes faster than reduction. Finally as the slowest method is the critical one, as the table shows, we can clearly see that with a higher number of threads the time increases a quite.

Day2: Explicit tasks

1.single.c

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

```
par2320@boada-1:~/lab2/openmp/Day2$ ./1.single
Thread 0 executing instance 0 of single
Thread 1 executing instance 1 of single
Thread 2 executing instance 2 of single
Thread 3 executing instance 3 of single
Thread 1 executing instance 4 of single
Thread 0 executing instance 5 of single
Thread 2 executing instance 6 of single
Thread 3 executing instance 7 of single
Thread 0 executing instance 8 of single
Thread 3 executing instance 11 of single
Thread 1 executing instance 10 of single
Thread 2 executing instance 9 of single
Thread 2 executing instance 13 of single
Thread 1 executing instance 12 of single
Thread 3 executing instance 14 of single
Thread 0 executing instance 15 of single
Thread 2 executing instance 16 of single
Thread 0 executing instance 19 of single
Thread 3 executing instance 18 of single
Thread 1 executing instance 17 of single
```

(Image 1.1)

```c
int main()
{
    int i;

    omp_set_num_threads(4);
    #pragma omp parallel private(i)
    for (i=0; i<N; i++) {
        #pragma omp single nowait
        {
            printf("Thread %d executing instance %d of single\n", omp_get_thread_num(), i);
            sleep(1);
        }
    }

    return 0;
}
```

(Image 1.2)

1. The nowait clause nullifies the implicit barrier that omp creates for the single clause. In other words, tasks can be executed without having to wait for others. As it is shown in image 1.1 instances 0 to 8 follow the natural order. Nevertheless, the following instances do not respect the clause's implicit barrier.

2. This is due to the sleep(1); line in code shown in image 1.2 .

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?
2. Modify the code so that tasks are executed in parallel and each iteration of the while loop is executed only once.
3. What is the `firstprivate(p)` clause doing? Comment it and execute again. What is happening with the execution? Why?

```
par2320@boada-1:~/lab2/openmp/Day2$ ./2.fibtasks
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
Thread 0 creating task that will compute 16
Thread 0 creating task that will compute 17
Thread 0 creating task that will compute 18
Thread 0 creating task that will compute 19
Thread 0 creating task that will compute 20
Thread 0 creating task that will compute 21
Thread 0 creating task that will compute 22
Thread 0 creating task that will compute 23
Thread 0 creating task that will compute 24
Thread 0 creating task that will compute 25
Finished creation of tasks to compute the Fibonacci for numbers in linked list
Finished computation of Fibonacci for numbers in linked list
1: 1 computed by thread 0
2: 1 computed by thread 0
3: 2 computed by thread 0
4: 3 computed by thread 0
5: 5 computed by thread 0
6: 8 computed by thread 0
7: 13 computed by thread 0
8: 21 computed by thread 0
9: 34 computed by thread 0
10: 55 computed by thread 0
11: 89 computed by thread 0
12: 144 computed by thread 0
13: 233 computed by thread 0
14: 377 computed by thread 0
15: 610 computed by thread 0
16: 987 computed by thread 0
17: 1597 computed by thread 0
18: 2584 computed by thread 0
19: 4181 computed by thread 0
20: 6765 computed by thread 0
21: 10946 computed by thread 0
22: 17711 computed by thread 0
23: 28657 computed by thread 0
24: 46368 computed by thread 0
25: 75025 computed by thread 0
```

```c
int fib(int n) {
   int x, y;
   if (n < 3) {
       return(1);
   } else {
      x = fib(n - 1);
      y = fib(n - 2);
      return (x + y);
   }
}

void processwork(struct node* p)
{
   int n;
   n = p->data;
   p->fibdata += fib(n);
   p->threadnum = omp_get_thread_num();
}

struct node* init_list(int nelems) {
    int i;
    struct node *head, *p1, *p2;

    p1 = malloc(sizeof(struct node));
    head = p1;
    p1->data = 1;
    p1->fibdata = 0;
    p1->threadnum = 0;
    for (i=2; i<=nelems; i++) {
       p2  = malloc(sizeof(struct node));
       p1->next = p2;
       p2->data = i;
       p2->fibdata = 0;
       p2->threadnum = 0;
       p1 = p2;
    }
    p1->next = NULL;
    return head;
}

struct node *p;

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;

     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;
     }
```

1. It's because there is only the a call, this is "#pragma omp task firstprivate(p) and it only develops tasks. Moreover only a thread is executed so that's the reason why the development seems to be sequential.

2. Code modified to execute in parallel and for each iteration of the while loop is executed only once:

```c
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
                processwork(p);
              p = p->next;
            }
            printf("Finished creation of tasks to compute the Fibonacci for numbers i

            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;
}
```

(Image 2.1)

3. The first private clause is making private the variable p and for the value default obtains the value of the global variable declared.

   What we can appreciate in the image 2.1 is the addition of the pragma parallel and single to make possible the parallelization of the code, it solves the problem as the exercicis demands. In the execution 2.2 we can appreciate that the thread different threads take randomly their tasks to do as it shows, so now the program works in parallel.

(Figure 2.2)

```
Thread 0 creating task that will compute 16
Thread 0 creating task that will compute 17
Thread 0 creating task that will compute 18
Thread 0 creating task that will compute 19
Thread 0 creating task that will compute 20
Thread 0 creating task that will compute 21
Thread 0 creating task that will compute 22
Thread 0 creating task that will compute 23
Thread 0 creating task that will compute 24
Thread 0 creating task that will compute 25
Finished creation of tasks to compute the Fib
Finished computation of Fibonacci for numbers
1: 1 computed by thread 3
2: 1 computed by thread 0
3: 2 computed by thread 0
4: 3 computed by thread 1
5: 5 computed by thread 0
6: 8 computed by thread 0
7: 13 computed by thread 0
8: 21 computed by thread 0
9: 34 computed by thread 0
10: 55 computed by thread 0
11: 89 computed by thread 0
12: 144 computed by thread 0
13: 233 computed by thread 0
14: 377 computed by thread 0
15: 610 computed by thread 2
16: 987 computed by thread 0
17: 1597 computed by thread 5
18: 2584 computed by thread 2
19: 4181 computed by thread 4
20: 6765 computed by thread 4
21: 10946 computed by thread 0
22: 17711 computed by thread 0
23: 28657 computed by thread 0
24: 46368 computed by thread 0
25: 75025 computed by thread 0
```

3.taskloop.c

1. Which iterations of the loops are executed by each thread for each task `grainsize` or `num_tasks` specified?
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?
3. Can `grainsize` and `num_tasks` be used at the same time in the same loop?
4. What is happening with the execution of tasks if the `nogroup` clause is uncommented in the first loop? Why?

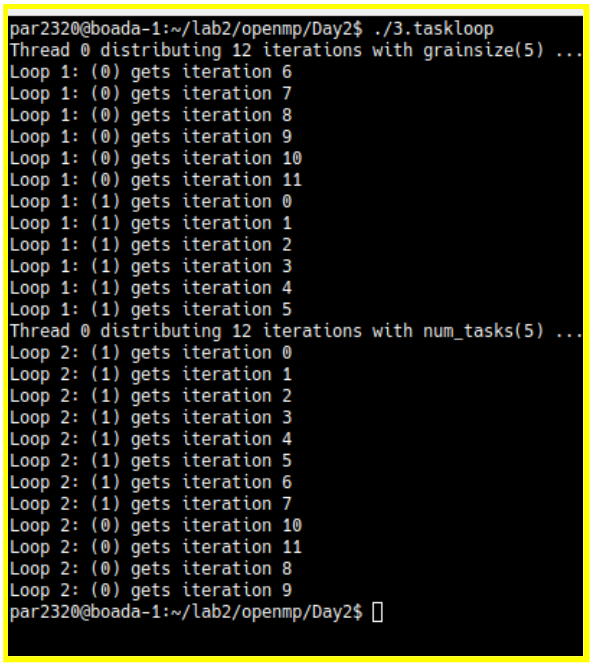
```
1.single  1.single.c  2.fibtasks  2.fibtasks.c  3.taskloop
par2320@boada-1:~/lab2/openmp/Day2$ ./3.taskloop
Thread 0 distributing 12 iterations with grainsize(4) ...
Loop 1: (0) gets iteration 8
Loop 1: (1) gets iteration 0
Loop 1: (1) gets iteration 1
Loop 1: (1) gets iteration 2
Loop 1: (1) gets iteration 3
Loop 1: (1) gets iteration 4
Loop 1: (1) gets iteration 5
Loop 1: (1) gets iteration 6
Loop 1: (1) gets iteration 7
Loop 1: (0) gets iteration 9
Loop 1: (0) gets iteration 10
Loop 1: (0) gets iteration 11
Thread 0 distributing 12 iterations with num_tasks(4) ...
Loop 2: (1) gets iteration 0
Loop 2: (1) gets iteration 1
Loop 2: (1) gets iteration 2
Loop 2: (0) gets iteration 9
Loop 2: (0) gets iteration 10
Loop 2: (0) gets iteration 11
Loop 2: (1) gets iteration 3
Loop 2: (1) gets iteration 4
Loop 2: (1) gets iteration 5
Loop 2: (2) gets iteration 6
Loop 2: (2) gets iteration 7
Loop 2: (2) gets iteration 8
par2320@boada-1:~/lab2/openmp/Day2$
```

(Image 3.1)

```c
#define VALUE 4

int main()
{
    int i;

    omp_set_num_threads(4);
    #pragma omp parallel
    #pragma omp single
    {
        printf("Thread %d distributing %d iterations with grainsize(%d) ...\n", omp_get_thread_num(), N, VALUE);
        #pragma omp taskloop grainsize(VALUE) // nogroup
        for (i=0; i < N; i++) {
                printf("Loop 1: (%d) gets iteration %d\n", omp_get_thread_num(), i);
        }

        printf("Thread %d distributing %d iterations with num_tasks(%d) ...\n", omp_get_thread_num(), N, VALUE);
        #pragma omp taskloop num_tasks(VALUE)
        for (i=0; i < N; i++) {
                printf("Loop 2: (%d) gets iteration %d\n", omp_get_thread_num(), i);
        }
    }

    return 0;
}
```

(Image 3.2)

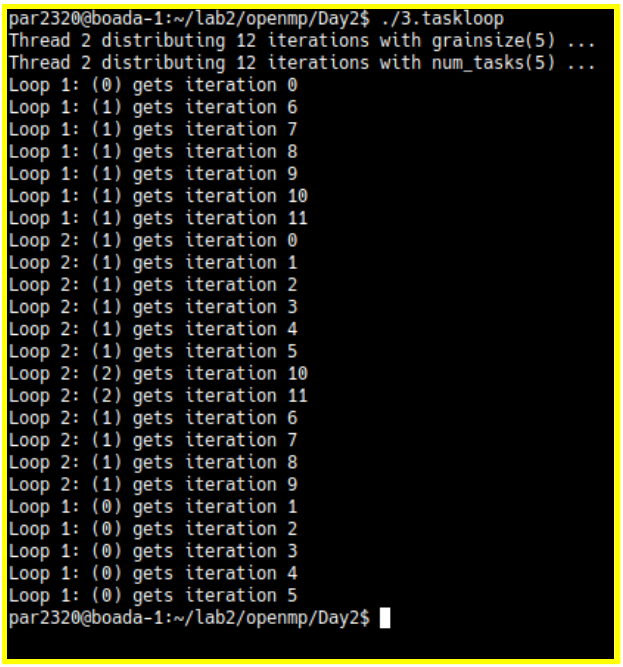
1. This information appears in image 3.1 the specified format in the code shown in image 3.2 . In Loop 1, 4 iterations are executed by thread (0) and 8 by thread (1). In Loop 2, 3 iterations are executed by thread (0), 6 by thread (1) and 3 by thread (2).



(Image 3.3)

2. Changing the VALUE variable from 4 to 5 produces the outcome shown in image 3.3 . In this case, the taskloop is a more coarse grained task so thread (2) does not get involved.

3.



(Image 3.4)

4. From image 3.4 we can observe that Loop 1 and Loop 2 tasks are not grouped for execution. In other words, it is not necessary to finish the first taskloop execution to execute tasks from the next one because it treats the grainsize taskloop as a next_task one.

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.

```
par2320@boada-1:~/lab2/openmp/Day2$ ./4.reduction
Value of sum after reduction in tasks = 33550336
Value of sum after reduction in taskloop = 66286736
Value of sum after reduction in combined task and taskloop = 99678925
par2320@boada-1:~/lab2/openmp/Day2$
```

(Image 4.1)

```
// Part II
#pragma omp taskgroup task_reduction(+: sum)
{
#pragma omp taskloop grainsize(BS)
for (i=0; i< SIZE; i++)
    sum += X[i];
}
printf("Value of sum after reduction in taskloop = %d\n", sum);
```

(Image 4.2)

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

#pragma omp taskloop 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);
```

(Image 4.3)

As seen in image 4.2 and 4.3 we need to parallelize both parts to obtain the same result. To solve it we privatize the sum, with a different variable than the shared one, in the sum of the loop and finally we sum the private with the shared one but having an atomic to avoid the manipulation of the others threads. It is shown in the image 4.4 .

```
    // Part II
int sumP = 0;
#pragma omp taskloop grainsize(BS)
for (i=0; i< SIZE; i++){
    sumP += X[i];
 }
 #pragma omp atomic
 sum += sumP;

    printf("Value of sum after reducti

    // Part III
int sumP1 = 0;
for (i=0; i< SIZE/2; i++){
    #pragma omp task firstprivate(i)
    sumP1 += X[i];
  }
  #pragma omp atomic
  sum += sumP1;

int sumP2 = 0;
#pragma omp taskloop grainsize(BS)
for (i=SIZE/2; i< SIZE; i++){
    sumP2 += X[i];
 }
 #pragma omp atomic
 sum += sumP2;
```

(Image 4.4 and the execution result)

```
par2311@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`.
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`.

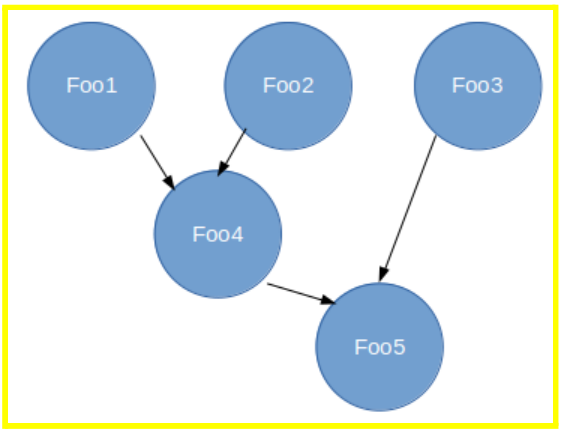
```
par2320@boada-1:~/lab2/openmp/Day2$ ./5.synchtasks
Creating task foo1
Creating task foo2
Creating task foo3
Starting function foo1
Creating task foo4
Creating task foo5
Starting function foo3
Terminating function foo1
Starting function foo2
Terminating function foo2
Starting function foo4
Terminating function foo3
Terminating function foo4
Starting function foo5
Terminating function foo5
par2320@boada-1:~/lab2/openmp/Day2$
```

(Image 5.1)

```c
int a, b, c, d;
int main(int argc, char *argv[]) {

    #pragma omp parallel
    #pragma omp single
    {
    printf("Creating task foo1\n");
    #pragma omp task depend(out:a)
    foo1();
    printf("Creating task foo2\n");
    #pragma omp task depend(out:b)
    foo2();
    printf("Creating task foo3\n");
    #pragma omp task depend(out:c)
    foo3();
    printf("Creating task foo4\n");
    #pragma omp task depend(in: a, b) depend(out:d)
    foo4();
    printf("Creating task foo5\n");
    #pragma omp task depend(in: c, d)
    foo5();
    }
    return 0;
}
```

(Image 5.2)

1. Image of the task dependence graph:



2. New version of the code (taskwaits) and the execution:

```c
int a, b, c, d;
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();
    printf("Creating task foo4\n");
    #pragma omp taskwait
    foo4();
    printf("Creating task foo5\n");
    #pragma omp taskwait
    foo5();
    }
    return 0;
}
```

```
par2311@boada-1:~/lab2/openmp/Day2$
Creating task foo1
Creating task foo2
Creating task foo3
Creating task foo4
Starting function foo1
Starting function foo3
Terminating function foo1
Starting function foo2
Terminating function foo2
Terminating function foo3
Starting function foo4
Terminating function foo4
Creating task foo5
Starting function foo5
Terminating function foo5
par2311@boada-1:~/lab2/openmp/Day2$
```

3. New version of the code (taskgroups) and the execution:

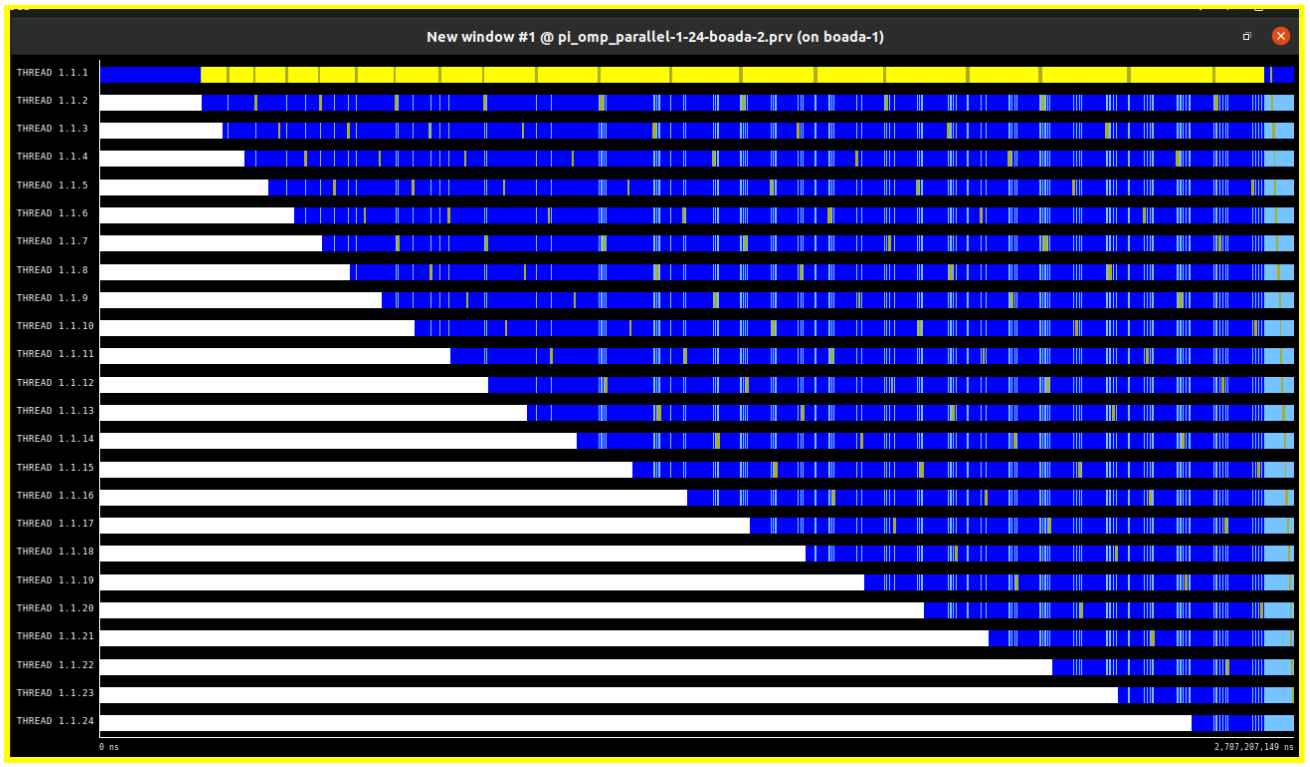
```c
    #pragma omp parallel
    #pragma omp single
    {
        #pragma omp taskgroup
        {
            printf("Creating task foo1\n");
            #pragma omp task
            foo1();
            printf("Creating task foo2\n");
            #pragma omp task
            foo2();
            printf("Creating task foo3\n");
            #pragma omp task
            foo3();
        }
        printf("Creating task foo4\n");
        #pragma omp task
        foo4();
        printf("Creating task foo5\n");
        #pragma omp task
        foo5();
    }
    return 0;
```

```
Creating task foo1
Creating task foo2
Creating task foo3
Starting function foo1
Starting function foo3
Terminating function foo1
Starting function foo2
Terminating function foo2
Terminating function foo3
Creating task foo4
Creating task foo5
Starting function foo4
Starting function foo5
Terminating function foo4
Terminating function foo5
```

Observing overheads

Thread creation and termination

2. "sbatch ./submit-omp.sh pi omp parallel 1 24", paraver view of the simulation



(Image 6.1)

3. "sbatch ./submit-omp.sh pi omp parallel 1 24", image of the .txt with the overheads and overheads per thread of the simulation (ms).

(Image 6.2)

All overheads expressed in microseconds

Nthr	Overhead	Overhead per thread
2	2.1734	1.0867
3	1.6063	0.5354
4	1.7443	0.4361
5	1.8247	0.3649
6	1.9870	0.3312
7	1.9747	0.2821
8	2.2798	0.2850
9	2.3738	0.2638
10	2.5136	0.2514
11	2.4719	0.2247
12	2.6149	0.2179
13	2.9019	0.2232
14	3.2785	0.2342
15	2.9346	0.1956
16	3.4497	0.2156
17	3.1762	0.1868
18	3.4280	0.1904
19	3.2660	0.1719
20	3.8891	0.1945
21	3.9759	0.1893
22	4.0247	0.1829
23	3.3962	0.1477
24	3.5521	0.1480

4. How does the overhead of creating/terminating threads vary with the number of threads used?

Overall overhead time increases gradually, while overhead time per thread reduces. This trend is expected as overhead time is directly dependent on the number of threads. But the reduction in overhead time per thread is explained because the majority is created by parallelization and not by thread creation.

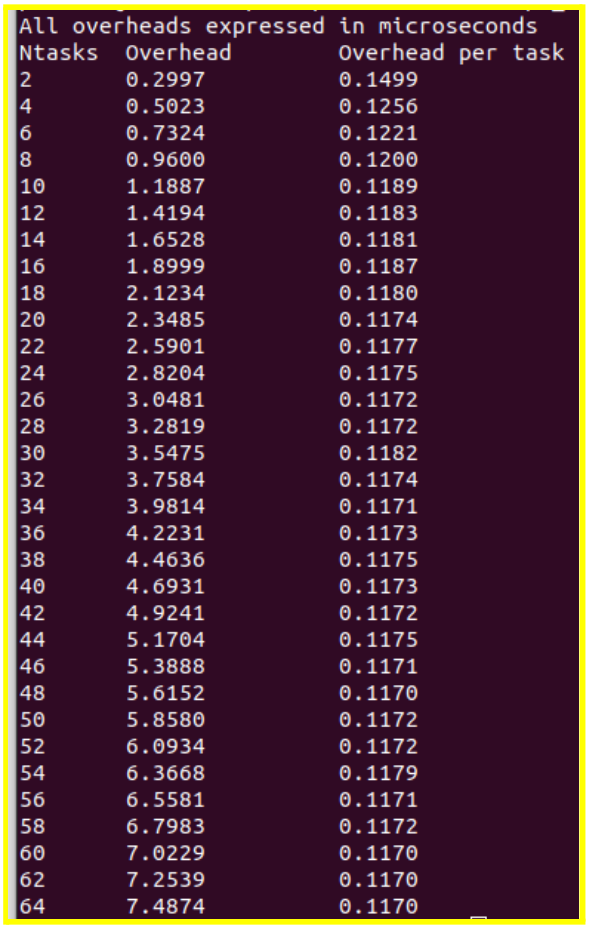
5. Which is the order of magnitude for the overhead of creating/terminating each individual thread in the parallel region?

The order of magnitude, as shown in the execution of “sbatch ./submit-omp.sh pi omp parallel 1 24” in image 6.2 is less than 0.1 ms.

Thread creation and synchronisation

1. "sbatch ./submit-omp.sh pi omp tasks 10 1", image of the .txt with the overheads and overheads per thread of the simulation (ms).

(Image 6.3)



All overheads expressed in microseconds

Ntasks	Overhead	Overhead per task
2	0.2997	0.1499
4	0.5023	0.1256
6	0.7324	0.1221
8	0.9600	0.1200
10	1.1887	0.1189
12	1.4194	0.1183
14	1.6528	0.1181
16	1.8999	0.1187
18	2.1234	0.1180
20	2.3485	0.1174
22	2.5901	0.1177
24	2.8204	0.1175
26	3.0481	0.1172
28	3.2819	0.1172
30	3.5475	0.1182
32	3.7584	0.1174
34	3.9814	0.1171
36	4.2231	0.1173
38	4.4636	0.1175
40	4.6931	0.1173
42	4.9241	0.1172
44	5.1704	0.1175
46	5.3888	0.1171
48	5.6152	0.1170
50	5.8580	0.1172
52	6.0934	0.1172
54	6.3668	0.1179
56	6.5581	0.1171
58	6.7983	0.1172
60	7.0229	0.1170
62	7.2539	0.1170
64	7.4874	0.1170

2. How does the overhead of creating/synchronising tasks vary with the number of tasks created?

   As shown in image 6.3, overall overhead time increases more than the previous execution and overhead time per task is more or less constant.

3. Which is the order of magnitude for the overhead of creating/synchronising each individual task?

   Now the order of magnitude in which overhead time increases has doubled from 0.1 ms to 0.2 ms approximately.

According to our data, thread and task parallelization have two significant differences. The first one is that overhead time per task is constant while overhead time per thread decreases, and the second one being that tasks parallelization has more impact in overhead time increase.