PAR Laboratory

Assignment Lab 2:

Brief tutorial on OpenMP programming model

Guillem Dubé Quintín par3103

Ricard Guixaró Trancho par3108

Fall 2021-2022

Date: 20/10/2022

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Introduction

The purpose of this laboratory is to introduce the main constructs in the *OpenMP* extensions to the C programming language. We'll go through a set of different codes for the computation of number Pi in parallel. To do that we go into the */lab2/pi/* directory. There we have different versions of *pi-vx.c* (x being the version), whose objective is to parallelise the sequential code. To compile them we can use:

- **make-pi-vx-debug**
- **make-pi-vx-omp**

If we use ***make pi-vx-debug*** a binary file that prints which iterations are executed by each thread and the value of Pi that is computed will be created. It can be useful to understand what the program is doing. We execute this binary with ***./run-debug.sh pi-vx*** interactively.

On the other hand, if we use ***make pi-vx-omp*** we'll time the parallel execution and will need to use *submit-omp.sh* (command: ***sbatch submit-omp.sh pi-vx***). It has a larger input value.

Day 1:

- **1.hello.c:**

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

Two times, because is the thread number per default.

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

We need to set the number to 4 externally when running the command with (OMP_NUM_THREADS = 4 ./1.hello).

- **2.hello.c:**

    - *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's not. We need to make the variable id private, so that each thread stops overwriting the *id* variable.
We would need to use the private construct and put a clause after the declaration of variable id, with *#pragma omp parallel num_threads(8) private (id).*

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

The *id* printed by each thread has the value of the last modification, meaning that it will differ on each execution.

- **3.how_many.c:**

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

Returns the number of threads inside the parallel execution. When outside the parallel region, it will return 1, as only one thread will execute that part.

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

With “#pragma omp parallel num_threads(4)” we change that number of threads supposed to execute that part.

    - *Which is the lifespan for each way of defining the number of threads to be used?*

It will last until that line is executed (?).

- **4.data_sharing.c:**

    - *Which is the value of variable x after the execution of each parallel region with different datasharing attribute (shared, private, firstprivate and reduction)? Is that the value you would expect?*

The value of *shared* is x = 120, though it should differ each execution because all threads are overwriting themselves.

The clauses *private* and *firstprivate* are the same (x = 5) because all threads are writing using its own local copy of x, therefore, outside the parallel region, the variable will remain the same. The difference is that private has a default value of 0, and firstprivate uses that value before entering the parallel execution.

And for the *reduction*, each thread has a private x initialized to 0, that in the end will be computed into the global one. If we have 16 threads, the sum will be 0+1+2+...+15) + 5, which is 125.

- **5.datarace.c:**

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

No, actually it neves does. Because each one of the 8 threads read and write the same variable.

    - *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.*

We could use “#pragma omp atomic ++x;” and “#pragma omp critical ++x;”, so that region of the code is locked and the threads don’t overwrite themselves.

    - *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.*

- **6.datarace.c:**

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

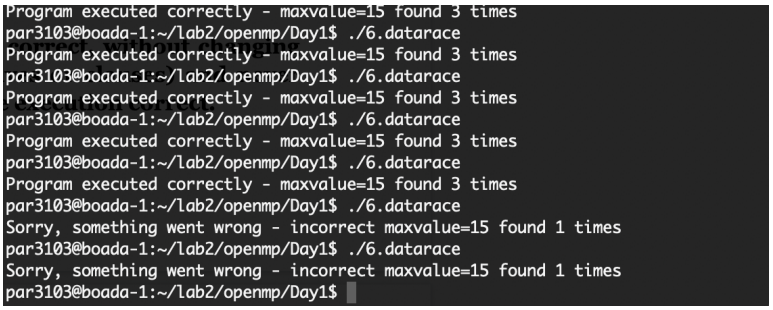

Figure 1: 6.datarace output.

If we execute the program several times we can see that it is executed incorrectly, that's because the variable countmax is being shared with other threads when we execute.

    - *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.*

We can use either atomic or doing a reduction of the variable countmax.
#pragma omp atomic before countmax++, or in the first declaration of the parallel region we add reduction(+: countmax).
The execution is now correct because now just one thread can access to the variable at a given time.

- **7.datarace.c:**
    - *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).*

Figure 2: 7.datarace output.

What we can see in this picture is that the execution is not correct.

This happens because of the reduction clauses.

The first accesses to the variables countmax and maxvalue are not protected by the reduction.

- *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) {
            #pragma omp atomic
            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, countmax);
    else printf("Sorry, something went wrong - maxvalue=%d found %d times\n", maxvalue, countmax);

    return 0;
}
```

Figure 3: 7.datarace.c modified code

```
par3103@boada-1:~/lab2/openmp/Day1$ ./7.datarace
Program executed correctly - maxvalue=15 found 3 times
par3103@boada-1:~/lab2/openmp/Day1$ ./7.datarace
Program executed correctly - maxvalue=15 found 3 times
par3103@boada-1:~/lab2/openmp/Day1$ ./7.datarace
Program executed correctly - maxvalue=15 found 3 times
par3103@boada-1:~/lab2/openmp/Day1$ ./7.datarace
Program executed correctly - maxvalue=15 found 3 times
par3103@boada-1:~/lab2/openmp/Day1$
```

Figure 4: 7.datarace output.

- **8.barrier.c:**

  - *Can you predict the sequence of printf in this program? Do threads exit from the #pragma omp barrier construct in any specific order?*

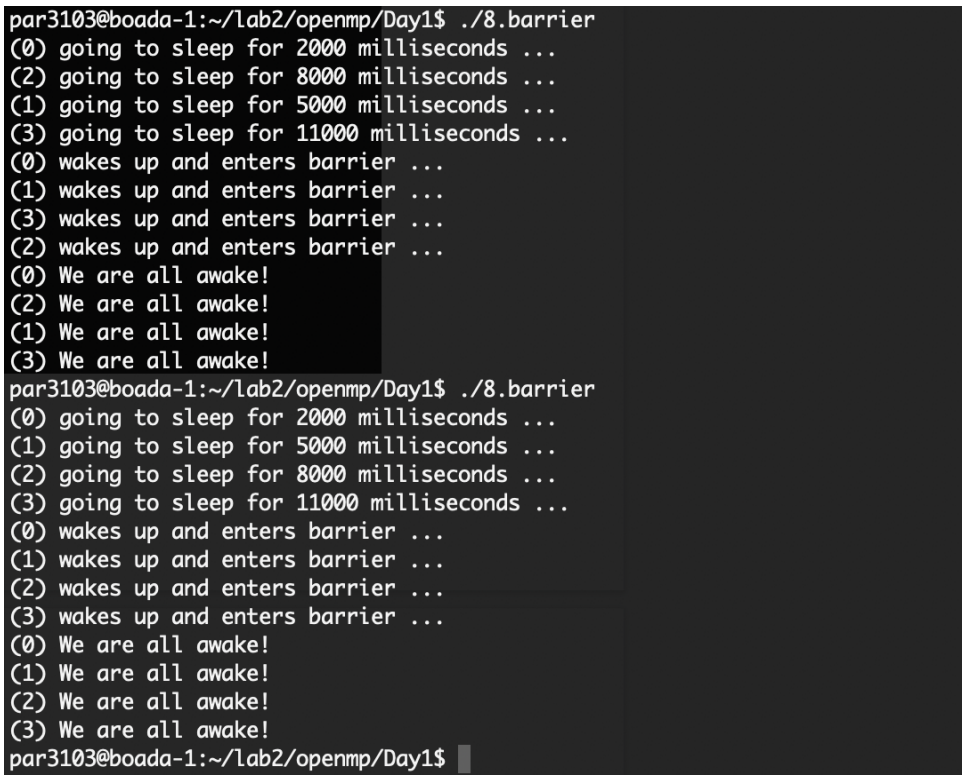

Figure 4: 8.barrier output.

As we can see they are all written 4 times, this is because of the *private(myid)* clause.

Threads don't exit in a particular order, they wait for themselves in the *pragma omp barrier* clause but it acts as a synchronization point.

Day 2:

- **1.single.c:**

    - *What is the nowait clause doing when associated to single?*

Pragma omp single does an automatic pragma omp barrier at the end of the loop, so with the ”nowait” clause, the loop is executed with 4 threads every iteration of the loop, instead of one executing and others waiting in the barrier.
The omp barrier directive identifies a synchronization point at which threads in a parallel region will not execute beyond the omp barrier until all other threads in the team complete all explicit tasks in the region.

    - *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?*

Using the *nowait* clause we avoid the barrier and all 4 threads execute the loop. If we remove it, only one thread at a time will execute the loop. They appear to be executed in bursts because when all 4 threads have done their loop iteration, they wait.

- **2.fibtasks.c:**

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

Because we are missing the clause of ***#pragma omp parallel*** and also ***#pragma omp single,*** so that's why we are not creating threads.

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

```
par3103@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 link
Finished computation of Fibonacci for numbers in linked list
1: 1 computed by thread 1
2: 1 computed by thread 1
3: 2 computed by thread 1
4: 3 computed by thread 2
5: 5 computed by thread 1
6: 8 computed by thread 2
7: 13 computed by thread 1
8: 21 computed by thread 2
9: 34 computed by thread 3
10: 55 computed by thread 2
11: 89 computed by thread 1
12: 144 computed by thread 1
13: 233 computed by thread 3
14: 377 computed by thread 1
15: 610 computed by thread 2
16: 987 computed by thread 3
17: 1597 computed by thread 1
18: 2584 computed by thread 2
19: 4181 computed by thread 3
20: 6765 computed by thread 1
21: 10946 computed by thread 2
22: 17711 computed by thread 3
23: 28657 computed by thread 1
24: 46368 computed by thread 0
25: 75025 computed by thread 0
par3103@boada-1:~/lab2/openmp/Day2$
```

Figure 5: 2.fibtasks output

```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_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;
}
```

Figure 6: Modified code

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

Is creating a private variable for each thread keeping the value of variable “p” with the initial value before the parallel execution. The execution ends with a segmentation fault, because all threads are accessing different values of the variable “p”, creating dependencies.

- **3.taskloop.c:**

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

If we use *grainsize,* the iterations of the loop are divided so that each thread has minimum *grainsize* iterations and maximum *2*grainsize* iterations.

num_tasks evenly distributes the iterations of the loop between the number of tasks specified. In the example, task0 executes iterations 0-2, task1 3-5, task2 6-8 and task0 again 9-11.

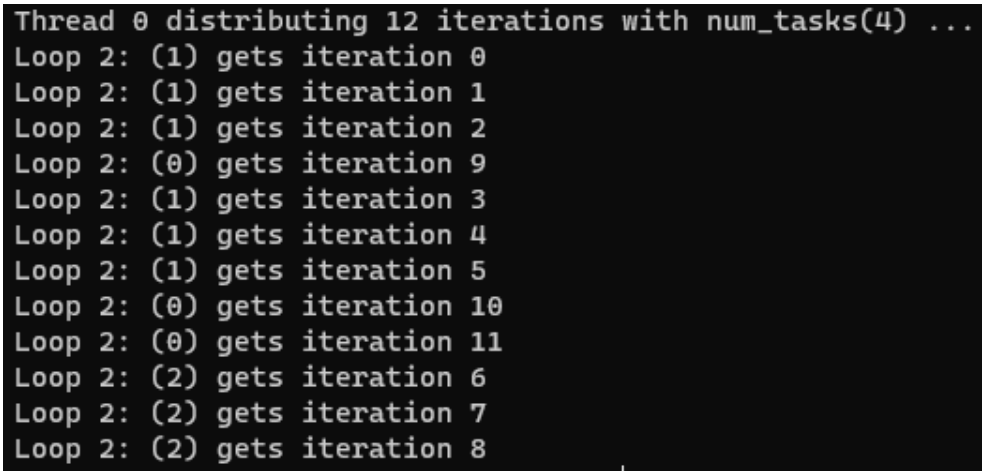

Figure 7: 3.taskloop output

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

```
par3103@boada-1:~/lab2/openmp/Day2$ ./3.taskloop
Thread 2 distributing 12 iterations with grainsize(5) ...
Loop 1: (2) gets iteration 6
Loop 1: (2) gets iteration 7
Loop 1: (2) gets iteration 8
Loop 1: (2) gets iteration 9
Loop 1: (2) gets iteration 10
Loop 1: (2) gets iteration 11
Loop 1: (0) gets iteration 0
Loop 1: (0) gets iteration 1
Loop 1: (0) gets iteration 2
Loop 1: (0) gets iteration 3
Loop 1: (0) gets iteration 4
Loop 1: (0) gets iteration 5
Thread 2 distributing 12 iterations with num_tasks(5) ...
Loop 2: (0) gets iteration 0
Loop 2: (0) gets iteration 1
Loop 2: (0) gets iteration 2
Loop 2: (1) gets iteration 3
Loop 2: (1) gets iteration 4
Loop 2: (1) gets iteration 5
Loop 2: (2) gets iteration 10
Loop 2: (2) gets iteration 11
Loop 2: (1) gets iteration 6
Loop 2: (1) gets iteration 7
Loop 2: (2) gets iteration 8
Loop 2: (2) gets iteration 9
```

Figure 8: 3.taskloop (modified grain size) output

In the first case for grain size, we have that every thread executes minimum 5 iterations, and in the second case for num_tasks we have that it divides the iterations in the number we tell him to do it, so in our case it will be 5, so it will execute 12 iterations divided in 5 tasks.

- *Can grainsize and num tasks be used at the same time in the same loop?*

```
par3103@boada-1:~/lab2/openmp/Day2$ make 3.taskloop
icc 3.taskloop.c -Wall -g -O3 -fno-inline -fopenmp -fopenmp -o 3.taskloop
3.taskloop.c(28): error: directive cannot contain both grainsize and num_tasks clau
ses
       #pragma omp taskloop num_tasks(VALUE) grainsize(VALUE) // nogroup
                                             ^

compilation aborted for 3.taskloop.c (code 2)
Makefile:17: recipe for target '3.taskloop' failed
make: *** [3.taskloop] Error 2
```

Figure 9: 3.taskloop output

No, they can’t be used in the same loop, the error message from the output above is quite clear.

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

nogroup basically removes the implicit *taskgroup* of the taskloop. Meaning that the tasks of the loop2 will be created and therefore loop1 and loop2 tasks will be mixed.

Threads execute loop 2 and loop 1 at the same time, because taskloop creates an automatic barrier at the end of the loop, so with the “nogrup” clause, what we see is that threads don’t wait in loop 1.

```
par3108@boada-1:~/lab2/openmp/Day2$ ./3.taskloop
Thread 2 distributing 12 iterations with grainsize(5) ...
Thread 2 distributing 12 iterations with num_tasks(5) ...
Loop 1: (0) gets iteration 0
Loop 2: (2) gets iteration 10
Loop 2: (2) gets iteration 11
Loop 1: (0) gets iteration 1
Loop 1: (0) gets iteration 2
Loop 2: (2) gets iteration 8
Loop 2: (2) gets iteration 9
Loop 2: (2) gets iteration 6
Loop 2: (2) gets iteration 7
Loop 2: (2) gets iteration 3
Loop 2: (2) gets iteration 4
Loop 2: (2) gets iteration 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: (1) gets iteration 10
Loop 1: (1) gets iteration 11
Loop 1: (0) gets iteration 3
Loop 1: (0) gets iteration 4
Loop 1: (0) gets iteration 5
Loop 2: (3) gets iteration 0
Loop 2: (3) gets iteration 1
Loop 2: (3) gets iteration 2
```

Figure 10: 3.taskloop (uncommented nogroup) output

- **4.reduction.c:**

    - *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.

```
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);
    sum = 0;
        // Part II
    #pragma omp taskloop grainsize(BS) reduction(+: sum)
    for (i=0; i< SIZE; i++)
        sum += X[i];
        printf("Value of sum after reduction in taskloop = %d\n", sum);
    sum = 0;
        // Part III
    #pragma omp taskgroup task_reduction(+: sum)
    {
    for (i=0; i< SIZE/2; i++)
        #pragma omp task firstprivate(i) in_reduction(+: sum)
        sum += X[i];
    }
    #pragma omp taskloop grainsize(BS) reduction(+: 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;
}
```

```
par3108@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
```

Figure 11: 4.reduction output after parallelisation.

- **5.synchtasks.c:**

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

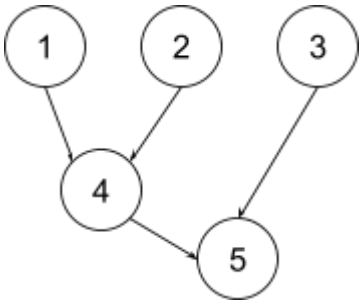

Figure 8: 5.synchtasks task dependence graph

1, 2 and 3 will run in parallel, but 4 will have to wait until 1 and 2 have finished, and then 5 will follow.

- *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 foo3\n");
        #pragma omp taskwait
        #pragma omp task
        foo3();
        printf("Creating task foo4\n");
        #pragma omp task
        foo4();
        printf("Creating task foo5\n");
        #pragma omp taskwait
        #pragma omp task
        foo5();
    }
```

```
    return 0;
}
```

```
par3108@boada-1:~/lab2/openmp/Day2$ ./5.synchtasks
Creating task foo1
Creating task foo2
Creating task foo3
Starting function foo2
Starting function foo1
Terminating function foo1
Terminating function foo2
Creating task foo4
Creating task foo5
Starting function foo3
Starting function foo4
Terminating function foo4
Terminating function foo3
Starting function foo5
Terminating function foo5
par3108@boada-1:~/lab2/openmp/Day2$
```

Figure 12: 5.synchtasks (modified) output

- *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.*

```
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 foo3\n");
                #pragma omp taskwait
                #pragma omp task
                foo3();
                printf("Creating task foo4\n");
                #pragma omp task
                foo4();
        }
        printf("Creating task foo5\n");
        #pragma omp task depend(in: c, d)
        foo5();
```

```
    }
    return 0;
}
```

```
par3108@boada-1:~/lab2/openmp/Day2$ ./5.synchtasks
Creating task foo1
Creating task foo2
Starting function foo2
Starting function foo1
Terminating function foo2
Terminating function foo1
Creating task foo3
Creating task foo4
Starting function foo4
Starting function foo3
Terminating function foo4
Terminating function foo3
Creating task foo5
Starting function foo5
Terminating function foo5
par3108@boada-1:~/lab2/openmp/Day2$ D
```

Figure 13: 5.synchtasks (modified) output

2.3 Observing overheads

2.3.1 Thread creation and termination

In the directory lab2/overheads we can find some versions of code:

- pi_omp_critical.c equivalent to pi-v4
- pi_omp_atomic.c equivalent to pi-v5
- pi_omp_sumlocal.c equivalent to pi-v6
- pi_omp_reduction.c equivalent to pi-v7

We can compile the 4 programs with *submit-omp.sh* script.

pi_omp_parallel.c computes the time difference (overhead) between the sequential and the parallel execution when using a certain number of threads.

```
par3108@boada-1:~/lab2/overheads$ cat pi_omp_parallel-1-24-boada-2.txt
All overheads expressed in microseconds
Nthr    Overhead        Overhead per thread
2       2.1523          1.0761
3       1.6712          0.5571
4       1.8279          0.4570
5       1.9424          0.3885
6       2.0230          0.3372
7       2.0721          0.2960
8       2.2709          0.2839
9       2.3311          0.2590
10      2.4353          0.2435
11      2.4373          0.2216
12      2.6169          0.2181
13      2.8799          0.2215
14      3.7516          0.2680
15      2.8800          0.1920
16      3.5398          0.2212
17      3.1745          0.1867
18      4.4509          0.2473
19      3.1988          0.1684
20      3.7078          0.1854
21      3.2676          0.1556
22      3.6226          0.1647
23      3.3072          0.1438
24      3.9921          0.1663
```

Figure 14: pi_omp_parallel output (1 iteration and 24 threads)

In the figure above we see that the overhead grows alongside the number of threads in a linear way. But the overhead per thread does not increase though, instead, it gets smaller as we use more threads. We can deduce then, that as we use more threads, the time difference between the sequential and the parallel execution increases, but as we are using more threads, that overhead is split in more parts.

```
par3108@boada-1:~/lab2/overheads$ cat pi_omp_atomic-1-24-boada-2.txt
Total overhead when executed with 1 iterations on 24 threads: 5501.0000 microseconds
par3108@boada-1:~/lab2/overheads$
par3108@boada-1:~/lab2/overheads$ cat pi_omp_critical-1-24-boada-2.txt
Total overhead when executed with 1 iterations on 24 threads: 5335.9583 microseconds
par3108@boada-1:~/lab2/overheads$
par3108@boada-1:~/lab2/overheads$ cat pi_omp_sumlocal-1-24-boada-2.txt
Total overhead when executed with 1 iterations on 24 threads: 4415.9583 microseconds
par3108@boada-1:~/lab2/overheads$
par3108@boada-1:~/lab2/overheads$ cat pi_omp_reduction-1-24-boada-2.txt
Total overhead when executed with 1 iterations on 24 threads: 4545.0000 microseconds
```

Figure 15: output for each version with 1 iteration and 24 threads.

```
par3108@boada-1:~/lab2/overheads$ cat pi_omp_atomic-1-16-boada-2.txt
Total overhead when executed with 1 iterations on 16 threads: 3982.0000 microseconds
par3108@boada-1:~/lab2/overheads$
par3108@boada-1:~/lab2/overheads$ cat pi_omp_critical-1-16-boada-2.txt
Total overhead when executed with 1 iterations on 16 threads: 3841.0000 microseconds
par3108@boada-1:~/lab2/overheads$
par3108@boada-1:~/lab2/overheads$ cat pi_omp_sumlocal-1-16-boada-2.txt
Total overhead when executed with 1 iterations on 16 threads: 7852.0000 microseconds
par3108@boada-1:~/lab2/overheads$
par3108@boada-1:~/lab2/overheads$ cat pi_omp_reduction-1-16-boada-2.txt
Total overhead when executed with 1 iterations on 16 threads: 4115.0000 microseconds
```

Figure 16: output for each version with 1 iteration and 16 threads.

```
par3108@boada-1:~/lab2/overheads$ cat pi_omp_atomic-1-8-boada-2.txt
Total overhead when executed with 1 iterations on 8 threads: 3230.0000 microseconds
par3108@boada-1:~/lab2/overheads$
par3108@boada-1:~/lab2/overheads$ cat pi_omp_critical-1-8-boada-2.txt
Total overhead when executed with 1 iterations on 8 threads: 3873.0000 microseconds
par3108@boada-1:~/lab2/overheads$
par3108@boada-1:~/lab2/overheads$ cat pi_omp_sumlocal-1-8-boada-2.txt
Total overhead when executed with 1 iterations on 8 threads: 3151.0000 microseconds
par3108@boada-1:~/lab2/overheads$
par3108@boada-1:~/lab2/overheads$ cat pi_omp_reduction-1-8-boada-2.txt
Total overhead when executed with 1 iterations on 8 threads: 4431.8750 microseconds
```

Figure 17: output for each version with 1 iteration and 8 threads.

```
par3108@boada-1:~/lab2/overheads$ cat pi_omp_atomic-1-4-boada-2.txt
Total overhead when executed with 1 iterations on 4 threads: 2925.0000 microseconds
par3108@boada-1:~/lab2/overheads$
par3108@boada-1:~/lab2/overheads$ cat pi_omp_critical-1-4-boada-2.txt
Total overhead when executed with 1 iterations on 4 threads: 2821.0000 microseconds
par3108@boada-1:~/lab2/overheads$
par3108@boada-1:~/lab2/overheads$ cat pi_omp_sumlocal-1-4-boada-2.txt
Total overhead when executed with 1 iterations on 4 threads: 2711.0000 microseconds
par3108@boada-1:~/lab2/overheads$
par3108@boada-1:~/lab2/overheads$ cat pi_omp_reduction-1-4-boada-2.txt
Total overhead when executed with 1 iterations on 4 threads: 2792.7500 microseconds
```

Figure 18: output for each version with 1 iteration and 4 threads.

```
par3108@boada-1:~/lab2/overheads$ cat pi_omp_atomic-1-1-boada-2.txt
Total overhead when executed with 1 iterations on 1 threads: 2500.0000 microseconds
par3108@boada-1:~/lab2/overheads$
par3108@boada-1:~/lab2/overheads$ cat pi_omp_critical-1-1-boada-2.txt
Total overhead when executed with 1 iterations on 1 threads: 2510.0000 microseconds
par3108@boada-1:~/lab2/overheads$
par3108@boada-1:~/lab2/overheads$ cat pi_omp_sumlocal-1-1-boada-2.txt
Total overhead when executed with 1 iterations on 1 threads: 2784.0000 microseconds
par3108@boada-1:~/lab2/overheads$
par3108@boada-1:~/lab2/overheads$ cat pi_omp_reduction-1-1-boada-2.txt
Total overhead when executed with 1 iterations on 1 threads: 2572.0000 microseconds
```

Figure 19: output for each version with 1 iteration and 1 threads.

Table to represent and compare all results obtained.

Version	**1 Thread** (ms)	**4 Threads** (ms)	**8 Threads** (ms)	**16 Threads** (ms)	**24 Threads** (ms)
critical	2500.0000	2925.0000	3230.0000	3982.0000	5335.9583
atomic	2510.0000	2821.0000	3873.0000	3841.0000	5501.0000
sumlocal	2784.0000	2711.0000	3151.0000	7852.0000	4415.9583
reduction	2572.0000	2792.7500	4431.8750	4115.0000	4545.0000

Due to the fact that we executed with just one iteration, the numbers don't show how as we use more threads and increase the number of threads, both *sumlocal* and *reduction* are far superior in rendiment terms than atomic and critical.

- The *critical* version is slow because there can only be one thread simultaneously as the instructions are protected.
- *Atomic* improves the times from the *critical* version because only the read and write instructions are blocked.
- The *sumlocal* and *reduction* versions produce better times because no region of code or instructions are protected, so all threads will compute simultaneously until the end, when they do the reduction.

2.3.2 Task creation and synchronisation

pi_omp_tasks.c measures the difference between the sequential execution and the version that creates the tasks.

```
par3108@boada-1:~/lab2/overheads$ cat pi_omp_tasks-10-1-boada-2.txt
All overheads expressed in microseconds
Ntasks  Overhead        Overhead per task
2       0.1742          0.0871
4       0.4830          0.1207
6       0.7294          0.1216
8       0.9665          0.1208
10      1.1829          0.1183
12      1.4182          0.1182
14      1.6497          0.1178
16      1.8875          0.1180
18      2.1224          0.1179
20      2.3462          0.1173
22      2.5868          0.1176
24      2.8128          0.1172
26      3.0541          0.1175
28      3.2736          0.1169
30      3.5222          0.1174
32      3.7553          0.1174
34      3.9958          0.1175
36      4.2267          0.1174
38      4.4430          0.1169
40      4.6932          0.1173
42      4.9212          0.1172
44      5.1598          0.1173
46      5.3897          0.1172
48      5.6377          0.1175
50      5.8527          0.1171
52      6.0961          0.1172
54      6.3395          0.1174
56      6.5278          0.1166
58      6.8124          0.1175
60      6.9965          0.1166
62      7.2515          0.1170
64      7.4975          0.1171
```

Figure 20: pi_omp_tasks output (10 iteration and 1 threads)

As *Figure 20* shows, the overhead per task is constant (~0.11ms) and therefore independent of the number of tasks.

Conclusion:

The cost of overheads is an important concept to have in mind while programming parallel programs, one thing that can occur is accumulating a lot of overheads so the sequential program is the same or better than the one we are parallelizing.

We have made an analisis to see which program accumulates more overheads and understand why it is so important when making parallelism and what it does with different types of clauses.

With the results we obtained we can see that the overheads are bigger when we use more threads while executing different programs.