PAR Laboratory Assignment
Lab 2: Brief tutorial on OpenMP programming model

E. Ayguadé, J. R. Herrero, P. Martínez-Ferrer,
J. Morillo, J. Tubella and G. Utrera Spring
2021-22

Maria Montalvo Falcón (par4214)
Victor Pla Sanchis (par4219)
Group: 42
Date: 24/03/22
Course: 2021-2022 Q2

Maria Montalvo Falcón - par4214
Victor Pla Sanchís - par4219
2021-2022 Q2
Group: 42

INDEX

Maria Montalvo Falcón - par4214
Victor Pla Sanchís - par4219
2021-2022 Q2
Group: 42

1 INTRODUCTION

In the following two sessions of laboratory two, we are going to be introduced to the clauses and directives of the interface of OpenMP.
As we are going to parallelise programs with this interface, we have to become familiar with this multiprocessing programming step by step.
In the next document, we will analyse different cases with our proposed solutions in order to understand how parallel programming works and how to solve the problems such as: data race condition, deadclock, starvation, among others with the clauses at our reach.
After all this introduction to OpenMP interface, we will have to analyse correctly what are the overheads in terms of parallel, task and synchronisation mechanisms. We will see that there is a cost to be paid in front of parallelising a program that has to be taken into account.

2 DAY ONE

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"?*
   Two times because the action it's executed by two threads (value of threads by default on a parallel region).
2. *Without changing the program, how to make it to print 4 times the "Hello World!" message?*
   Changing an environment variable called OMP_NUM_THREADS to 4 as we can see on Figure 2.1.

```
par4219@boada-1:~/lab2/openmp/Day1$ export OMP_NUM_THREADS=4
par4219@boada-1:~/lab2/openmp/Day1$ ./1.hello
Hello world!
Hello world!
Hello world!
Hello world!
```

Figure 2.1 Change on OMP_NUM_THREADS environment variable

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?*

   In this case, we can see on Figure 2.2 that the identifiers of the threads are not the same for the string Hello and the string world!

```
par4219@boada-1:~/lab2/openmp/Day1$ ./2.hello
(4) Hello (0) world!
(0) Hello (0) world!
(2) Hello (2) world!
(3) Hello (2) world!
(6) Hello (6) world!
(1) Hello (1) world!
(5) Hello (5) world!
(7) Hello (7) world!
```

Figure 2.2 Result of executing 2.hello.c

In order to solve this problem, we have added a critical region. With this, we can see that the result is corrected. See Figure 2.3 and 2.4.

```
par4219@boada-1:~/lab2/openmp/Day1$ ./2.hello
(0) Hello (0) world!
(3) Hello (3) world!
(2) Hello (2) world!
(6) Hello (6) world!
(1) Hello (1) world!
(5) Hello (5) world!
(4) Hello (4) world!
(7) Hello (7) world!
```

Figure 2.3 Result of executing 2.hello.c after modifications

```c
#include <stdio.h>
#include <omp.h>

/* Execute with ./2.hello                                  */
/* Q1: Is the execution of the program correct? Add a      */
/*     data sharing clause to make it correct              */
/* Q2: Are the lines always printed in the same order?     */
/*     Why the messages sometimes appear intermixed?       */

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

    return 0;
}
```

Figure 2.4 Changes made on 2.hello.c; adding data sharing clause

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

No, the threads are not given the result always in the same order. The messages appear inter-mixed because they execute and give the result as soon as they "arrive" to the point on the program.

3. how many.c:

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

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

Outside the parallel region: 1

Inside the parallel region: up to 4 as a consequence of a modification in the code.

```
par4219@boada-1:~/lab2/openmp/Day1$ OMP_NUM_THREADS=8
par4219@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 2.5 Get num threads result

2. *Indicate the two alternatives to supersede the number of threads that is specified by the OMP NUM THREADS environment variable.*
   One option is writing the following line:
   omp_set_num_threads(N) // with N equal to the number of threads
   The other option is typing the following pragma line:
   #pragma omp parallel ... num_threads(N) // with N equal to the number of threads

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

   For a number of threads set during all the program or until it is overwritten, we can use the method omp_set_num_threads(N) otherwise, if you just want to use a specific number of threads in a specific part of the code, you can use #pragma omp parallel ... num_threads(N) that executes just the pragma region with the threads specified.

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 value: this value should be 120 as we can see on Figure 2.6. However, after several executions, we have seen that sometimes the value given it's not correct. It could be because more than one thread is accessing variable x at the same time and they are overwriting the other's result (data race condition). The shared clause indicates that there are one or more variables shared, but not indicate the restrictions or order to access them.
   - Private: always show us an incorrect value of x outside the pragma omp parallel because every thread is creating their private x initialised at 0 but they are not modifying the real variable x at the end.
   - Firstprivate: as the private one, never showed us a correct value of x. That's because the firstprivate clause works exactly as the private one. Every thread uses its own variable x that is never used at the end of the pragma. There's just a difference, that every private variable x now is initialised with the value 5 (the shared value of x).
   - Reduction: in all the executions of the program printed the value 125 that is the correct one for the variable x. That's because the reduction clause allows us to use private variables, but in the end of the clause every thread knows that his own private variable is the subject of a reduction operation. That ends up with the correct sum of each private x variable saved on the extern x variable of every thread.

```
par4219@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
```

Figure 2.6 4.data_sharing results

5.datarace.c

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

   This program is not always returning the same value. The reason why it is happening is because of a problem with data race condition. During the execution of the program, the threads are overwriting the variable maxvalue and at the end, although unlikely, the program is returning a wrong answer.

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.*

Alternative 1:

```
#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 (maxvalue)
    if (vector[i] > maxvalue)
       maxvalue = vector[i];
    }
  }
```

Figure 2.7 Alternative 1 with critical

With critical clause on maxvalue, we are assuring that only one thread is accessing maxvalue at a time in the if region, so we can avoid the problem of data race condition.

Even this solution gives us the correct answer, critical value increases the time of execution a lot because of the blocking and unblocking section. This is because this blockage it's controlled by software.

Alternative 2:

```
#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 write
       maxvalue = vector[i];
    }
    }
  }
```

Figure 2.8 Alternative 2 with atomic write

As a second alternative, we have proposed the use of atomic clause. In this case, the clause works similarly to critical one, but more efficiently as the control of the blocking sections are made by hardware.

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.*

   With the use of the clause *for*, the work done in the loop inside a parallel region will be divided among threads.

   However, we have distributed the work "manually" as shown on Figure 2.9.

```
int offset = N >> 3; // N / 8
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*offset; i < offset * (id + 1); i++) {
    if (vector[i] > maxvalue) {
       #pragma omp atomic write
       maxvalue = vector[i];
    }
    }
  }
```

Figure 2.9 Division of work per thread

6.datarace.c

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

   The program is not always returning a correct answer as we can see on Figure 2.10. The reason why this is happening is because threads are accessing the variable countmax sometimes at the same time and without control. This results in a data race condition problem.

```
par4219@boada-1:~/lab2/openmp/Day1$ ./6.datarace
Program executed correctly - maxvalue=15 found 3 times
par4219@boada-1:~/lab2/openmp/Day1$ ./6.datarace
Program executed correctly - maxvalue=15 found 3 times
par4219@boada-1:~/lab2/openmp/Day1$ ./6.datarace
Program executed correctly - maxvalue=15 found 3 times
par4219@boada-1:~/lab2/openmp/Day1$ ./6.datarace
Program executed correctly - maxvalue=15 found 3 times
par4219@boada-1:~/lab2/openmp/Day1$ ./6.datarace
Sorry, something went wrong - incorrect maxvalue=15 found 1 times
par4219@boada-1:~/lab2/openmp/Day1$ ./6.datarace
Program executed correctly - maxvalue=15 found 3 times
```

Figure 2.10 Result of executing 6.data_race

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.*

   Alternative 1:

   As a first alternative, we have opted to add the clause atomic. As we have said, it restricted the access to a variable in a section in order to control the reads and the writes. We have chosen this clause because it is more efficient than critical one.

```
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 found %d times\n", maxv
    else printf("Sorry, something went wrong - incorrect maxvalue=%d found %d tim

    return 0;
}
```

Figure 2.11 Alternative one with atomic solution

Alternative 2:

As our second alternative, we have added the reduction clause of the for in the countmax value. In this case the reduction makes the threads counting locally the value of the contmax variable but adding their responses at the end of the region in order to synchronise the solution. This is a good and efficient option to keep the benefits of parallelising.

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

Figure 2.12 Alternative two with reduction solution

7.datarace.c

1. *Is this program executing correctly? If not, explain why it is not providing the correct result for one or the two variables (countmax and maxvalue)*

   No, the program does never give the correct answer as each thread have their own variable for maxvalue and countmax. At the end of the program they are summing up all of them values giving a wrong response.

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

   In order to synchronise all the threads, we have made the variables countmax and maxvalue shared and we have controlled the access to them with atomic clause.

```
#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)
    {
    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) {
           #pragma omp atomic write
           maxvalue = vector[i];
           #pragma omp atomic write
           countmax = 1;
        }
    }
    }
```

Figure 2.13 Syncronization to 7.data_race

8.barrier.c

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

We have predicted that in fact, we can't do any prediction as we don't know the order in which the threads will be executed and the time that the processor will be executing each thread or when they are going to be pushed out.
Moreover, we think that if the second printf have a higher separation in time, the threads will print in order, but in this case, as the distance is so low, we can't say the same.

```
par4219@boada-1:~/lab2/openmp/Day1$ ./8.barrier
(2) going to sleep for 8000 milliseconds ...
(0) going to sleep for 2000 milliseconds ...
(1) going to sleep for 5000 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!
(3) We are all awake!
(2) We are all awake!
(1) We are all awake!
```

Figure 2.14 Proved of execution of program 8.barrier

3 DAY TWO

Day 2: explicit tasks

1.single.c

1. *What is the nowait clause doing when associated to single?*
   The nowait clause associated with single overrides the implicit barrier of single clause. It means, that the rest of the threads are not waiting for the thread executing the single block.
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?*

   With nowait clause, the printf is made by a group of threads because of the omp parallel region, so, in other words, they are all participating in the execution of the loop and each iteration is randomly assigned to a thread because of the single nowait clause. The code is executed in bursts because there is a sleep(1) after the printf as a kind of "barrier".

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?*

   In 2.fibtasks.c program, there's no specification for executing the program in parallel. In other words, it is not instantiated the clause "#pragma omp parallel".
2. *Modify the code so that tasks are executed in parallel and each iteration of the while loop is executed only once.*

   We have modified the code in order to make the region of the task parallel and executed each task by only one thread.

```
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 3.1 Code modified on 2.fibtasks

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

   The firstprivate(p) clause is generating a private variable p for each thread to execute the processwork(p) with p initialised with the value of the original variable at the moment the task is encountered. In Figure 3.2, we can see that commenting the clause implies the program to be executed by only one thread. It is because without the clause, the explicit tasks' creation it's not being invoked.

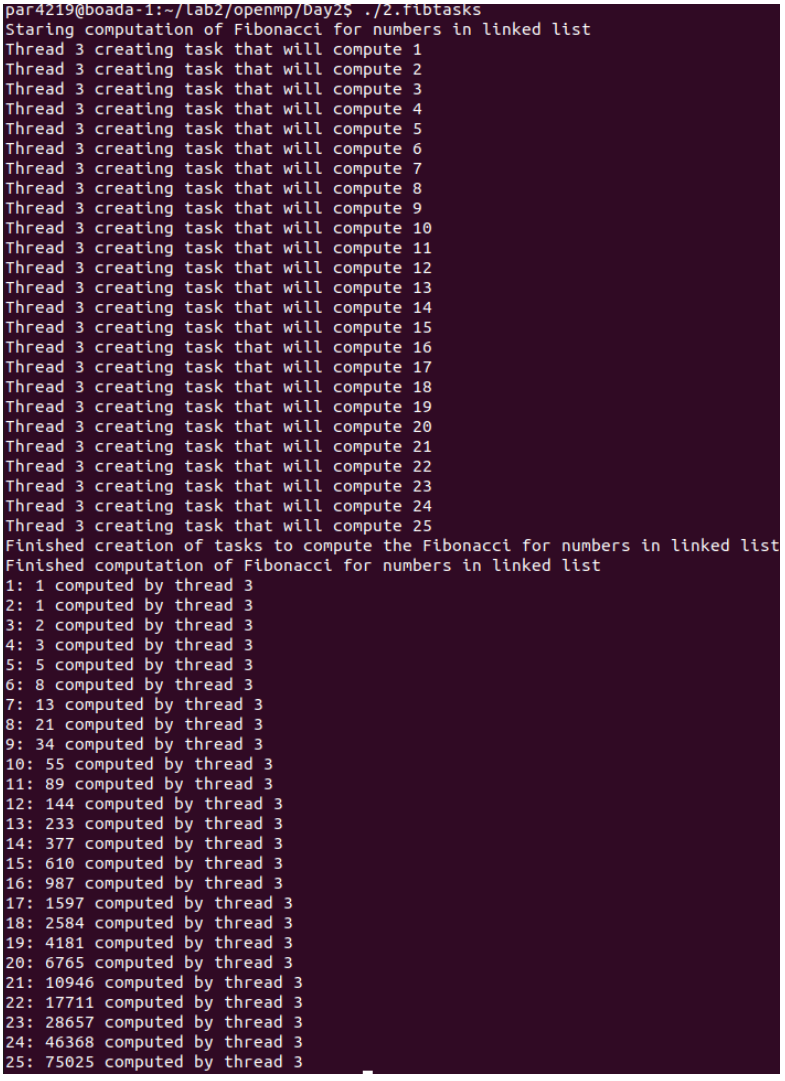

Figure 3.2 Output after commenting #pragma omp task firstprivate(p) clause

3.taskloop.c

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

   The clause #pragma omp taskloop grainsize(value) of the first loop is creating *N/grainsize* tasks. In our code N is equal to 12 and grainsize is equal to 4, so there are 3 tasks of 4 iterations for the first loop.

   On the other loop, the tasks creation is made by the clause #pragma omp taskloop num_tasks(value). It means that there are created 4 tasks of 3 iterations *4 = 12/grainsize*.

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?*

   If we change the grainsize to 5, the threads are executing 6 iterations, which means that they are doing 2 tasks of grainsize equal to 6. We can see that in the first loop, the grainsize has been increased to made an integer value of tasks.

   In the second loop, we have seen that the grainsize is equal to 2 or 3 but the number of tasks will always be equal to 5. That means that there are 3 tasks of 2 iterations and 2 tasks of 3 iterations in order to have the total iterations of the code, in this case 12 with only 5 tasks. Moreover, as there are more tasks than threads, some threads will execute more than one task.

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

   The grainsize clause and num_tasks clause are mutually exclusive and may not appear on the same taskloop directive, as we can see in the compilation error of Figure 3.4.

```
par4219@boada-1:~/lab2/openmp/Day2$ make
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 clauses
        #pragma omp taskloop grainsize(VALUE) num_tasks(VALUE) // nogroup
                                              ^

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

Figure 3.3 Compilation error when setting grainsize and num_tasks at the same time

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

   With the clause we have seen that each thread is executing the tasks as they get it, so there is no taskgroup in loop1.

   The nogroup clause removes the implicit taskgroup region that encloses the taskloop construct.

```
par4219@boada-1:~/lab2/openmp/Day2$ ./3.taskloop
Thread 0 distributing 12 iterations with grainsize(5) ...
Thread 0 distributing 12 iterations with num_tasks(5) ...
Loop 1: (1) gets iteration 0
Loop 2: (0) gets iteration 10
Loop 2: (0) gets iteration 11
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 2: (1) gets iteration 0
Loop 2: (1) gets iteration 1
Loop 2: (1) gets iteration 2
Loop 2: (1) gets iteration 3
Loop 2: (1) gets iteration 4
Loop 2: (1) gets iteration 5
Loop 2: (1) gets iteration 6
Loop 2: (1) gets iteration 7
Loop 2: (0) gets iteration 8
Loop 2: (0) gets iteration 9
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
```

Figure 3.4 Execution with nogroup

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.*

   - Part I: has not been modified as was the one that works correctly.
   - Part II: the loop was divided into tasks of granularity BS but each thread was overwritten the value of sum giving a wrong answer. We have added firstprivate(sum) clause in order each thread remains the value of the previous sum and the work locally with a private sum.
   - Part III: In this part the code was divided into two subloops. In order to synchronise the result of both loops, the clause pragma omp taskgroup task_reduction(+:sum) was added. Moreover, before both subloops we have added in the first one the clause grainsize(BS) and in the second one the clause firstprivate(sum).

```
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;
"4.reduction.c" 58L, 1434C written
par4219@boada-1:~/lab2/openmp/Day2$ make
icc 4.reduction.c -Wall -g -O3 -fno-inline -fopenmp -fopenmp -o 4.reduction
par4219@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 3.5 Code modified of 4.reduction

5.synchtasks.c

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

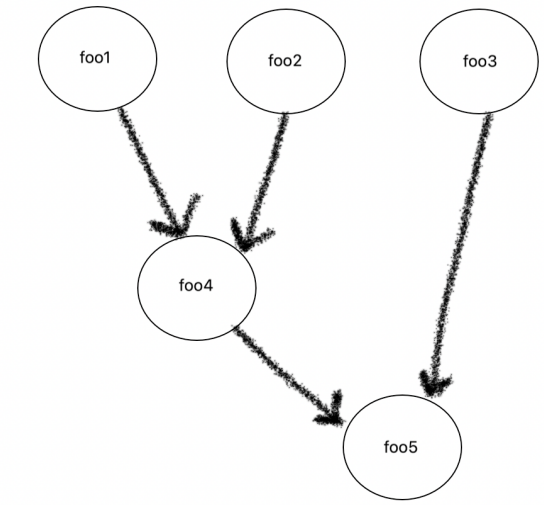

Figure 3.6 TDG of synchtasks

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.*

   We create a task with the clause #pragma omp task for each foo and add the clause #pragma omp taskwait to the tasks that have dependencies.

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

        printf("Creating task foo3\n");
        #pragma omp task
        foo3();

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

Figure 3.7 Program with taskwait

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 part we have grouped the tasks that could be executed at the same time. We have distinguished foo1, foo2, foo3 for the first group, foo4 for the second, and foo5 for the third as show in the Figure 3.8.

```
int a, b, c, d;
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();

                printf("Creating task foo3\n");
                #pragma omp task
                foo3();
        }

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

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

Figure 3.8 Program with taskgroup

4 OBSERVING OVERHEADS AND CONCLUSIONS

Overhead in terms of parallel

When we want to parallelise a program there are 4 ways to manage the validity of our critical variables that depend on the correctness of the program. That clauses are:

1. #pragma omp critical
2. #pragma omp atomic
3. #pragma omp sumlocal
4. #pragma omp reduction

We have executed the pi-omp program to see how overheads behave in all the cases, always with 1.000.000.000 iterations and with one, four, and eight threads.

```
par4219@boada-1:~/lab2/overheads$ cat pi_omp_critical-1000000000-1-boada-2.txt
Total overhead when executed with 1000000000 iterations on 1 threads: 24862462.0000 microseconds
par4219@boada-1:~/lab2/overheads$ cat pi_omp_atomic-1000000000-1-boada-3.txt
Total overhead when executed with 1000000000 iterations on 1 threads: 6054.0000 microseconds
par4219@boada-1:~/lab2/overheads$ cat pi_omp_sumlocal-1000000000-1-boada-3.txt
Total overhead when executed with 1000000000 iterations on 1 threads: 5700.0000 microseconds
par4219@boada-1:~/lab2/overheads$ cat pi_omp_reduction-1000000000-1-boada-3.txt
Total overhead when executed with 1000000000 iterations on 1 threads: 363492.0000 microseconds
```

Figure 4.1 Execution with 1 thread

```
par4219@boada-1:~/lab2/overheads$ cat pi_omp_critical-1000000000-4-boada-2.txt
Total overhead when executed with 1000000000 iterations on 4 threads: 372657384.0000 microseconds
par4219@boada-1:~/lab2/overheads$ cat pi_omp_atomic-1000000000-4-boada-2.txt
Total overhead when executed with 1000000000 iterations on 4 threads: 52081901.5000 microseconds
par4219@boada-1:~/lab2/overheads$ cat pi_omp_sumlocal-1000000000-4-boada-2.txt
Total overhead when executed with 1000000000 iterations on 4 threads: 52471.2500 microseconds
par4219@boada-1:~/lab2/overheads$ cat pi_omp_reduction-1000000000-4-boada-2.txt
Total overhead when executed with 1000000000 iterations on 4 threads: 52591.5000 microseconds
```

Figure 4.2 Execution with 4 threads

```
par4219@boada-1:~/lab2/overheads$ cat pi_omp_critical-1000000000-8-boada-3.txt
Total overhead when executed with 1000000000 iterations on 8 threads: 340245084.8750 microseconds
par4219@boada-1:~/lab2/overheads$ cat pi_omp_atomic-1000000000-8-boada-4.txt
Total overhead when executed with 1000000000 iterations on 8 threads: 54623375.5000 microseconds
par4219@boada-1:~/lab2/overheads$ cat pi_omp_sumlocal-1000000000-8-boada-2.txt
Total overhead when executed with 1000000000 iterations on 8 threads: 119088.5000 microseconds
par4219@boada-1:~/lab2/overheads$ cat pi_omp_reduction-1000000000-8-boada-3.txt
Total overhead when executed with 1000000000 iterations on 8 threads: 157346.6250 microseconds
```

Figure 4.3 Execution with 8 threads

PRAGMA	1 THREAD	4 THREADS	8 THREADS	S(4)	S(8)
Critical	24,862462	372,657384	340,245084	0,06671667614	0,0730722152
Atomic	0,006054	52,081901	54,623375	0,0001162399967	0,0001108316723
Sumlocal	0,0057	0,05247125	0,1190885	0,1086309169	0,04786356365
Reduction	0,363492	0,052915	0,157346625	6,869356515	2,310135346

Table 4.1 Table summarising executions with 1, 4 and 8 threads

With the results of Table 4.1 summarising the executions for different threads, we have seen that both critical and atomic clauses are increasing the time execution and making the program more inefficient when adding threads. It makes sense as these clauses are blocking and unblocking the resources to make threads work sequentially. In addition, we can see, as we have commented previously, that the overhead added when critical clause is used is even more that with the atomic one; also, as we have said, it is because the critical clause it's implemented by software while the atomic one it's performed by hardware.

In sumlocal program, a private and local sum is used to divide the work within threads but the clause critical is required to sum to the global variable; this is why we have not seen any improvement on sumlocal speedup and even a worsening in time.

The only clause that really implies an improvement when adding threads is reduction. In the case with 4 threads we have seen a speedup of 6.8 with respect to the version executed only with one thread. In any case, the speedup with 8 threads is lower than the one with 4 threads; that means that adding more threads is making a deterioration in the execution time as a consequence of time overhead.

Overhead in terms of thread creation

Another point to take into consideration when parallelising a program is the overhead in the creation of threads. We will execute the program pi_omp_parallel.c with 1 iteration from 1 thread to 24 threads. The output of the program lets us see how the overhead behaves in 2 ways, total overhead of the program and average overhead per thread.

As we see on Figure 4.4, generally the total overhead increases by adding more threads, meanwhile, the overhead per thread decreases. That means that the increase in overhead is shared between the number of threads, and every thread is assuming a lower proportion of overhead. So we can describe the curve of overhead in terms of task creation as a diminishing marginal growth in overhead per thread. Even if the ratio per thread is smaller, the increase in the total overhead is due to the creation and synchronisation of the number of threads, and makes sense that when adding new threads, the total overhead increases.

We can conclude that initially there is a big fixed overhead to pay in terms of thread creation and a variable overhead per thread that in this case, is a marginally decreasing increase.

```
par4219@boada-1:~/lab2/overheads$ cat pi_omp_parallel-1-24-boada-2.txt
All overheads expressed in microseconds
Nthr    Overhead        Overhead per thread
2       2.0110          1.0055
3       1.4075          0.4692
4       1.6432          0.4108
5       1.8118          0.3624
6       1.9190          0.3198
7       1.8944          0.2706
8       2.1242          0.2655
9       2.1803          0.2423
10      2.3096          0.2310
11      2.2730          0.2066
12      2.4747          0.2062
13      2.7328          0.2102
14      3.2848          0.2346
15      2.7768          0.1851
16      3.1965          0.1998
17      2.9882          0.1758
18      3.5772          0.1987
19      2.9910          0.1574
20      3.7304          0.1865
21      3.1246          0.1488
22      3.7216          0.1692
23      3.1920          0.1388
24      3.3782          0.1408
```

Figure 4.4 Execution of pi_omp_parallel.c

Overhead in terms of synchronisation mechanisms

Finally, we will see how the overhead behaves on parallel synchronisation mechanisms, or in other words, in the creation and synchronisation of tasks. For that, we will execute pi_omp_tasks.c program with 1 iteration from 1 thread to 24 as we did recently. In this case, we are going to analyse the evolution on total overhead and on overhead per task.

In that case, we can clearly see on Figure 4.5 how the total overhead increases fastly, meanwhile, the overhead per task remains almost constant. We can deduce that the increase of total overhead time is produced for other factors independently of the number of tasks.

```
par4219@boada-1:~/lab2/overheads$ cat pi_omp_tasks-1-10-boada-2.txt
All overheads expressed in microseconds
Ntasks  Overhead        Overhead per task
2       3.7580          1.8790
4       7.7812          1.9453
6       11.5633         1.9272
8       15.2125         1.9016
10      19.0343         1.9034
12      22.7392         1.8949
14      26.9704         1.9265
16      31.2224         1.9514
18      35.0872         1.9493
20      38.8423         1.9421
22      42.8746         1.9488
24      46.7495         1.9479
26      50.4836         1.9417
28      54.2900         1.9389
30      58.1307         1.9377
32      61.9235         1.9351
34      65.3999         1.9235
36      69.1633         1.9212
38      73.1750         1.9257
40      77.0469         1.9262
42      80.2827         1.9115
44      83.9962         1.9090
46      87.8196         1.9091
48      91.8680         1.9139
50      95.3727         1.9075
52      99.0822         1.9054
54      103.2700                1.9124
56      107.0318                1.9113
58      110.7075                1.9087
60      114.6128                1.9102
62      118.2078                1.9066
64      121.8856                1.9045
```

Figure 4.5 Execution of pi_omp_tasks.c

Conclusions

In this second laboratory we have analysed different aspects to take into account when parallelising a program. Apparently, what one might consider a good idea in the first instance, could become a worse performance of a program without taking into account different overheads that may appear.In other words, more threads do not always imply less time execution.

Moreover, we have analysed some clauses that would be useful along the course and weigh them up with their cons and pros. We have also seen how OpenMP clauses works in different codes and why or why not them are correct or making a better performance as we have seen in the case of critical or atomic clauses.