PARAL·LELISME: LABORATORI 2

# 

Pol Casacuberta Gil i Salvador Grimalt BInimelis

Quadrimestre 1, curs 22-23

Paral·lelisme, grup 21

[**Questionnaire**](#_yh0cynwswn6v) **3**

[Day 1: Parallel regions and implicit tasks](#_x8u1y8a8qc45) 3

[Day 2: Explicit tasks](#_e27bo7u8rqzt) 8

[Overheads part 1](#_mpfquz6zt673) 13

[Overheads part 2](#_cffdvtq088tz) 16

# 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"?**

It will be printed two times because the “#pragma omp parallel” instruction creates two threads by default.

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

By changing the value of the variable with this command: OMP\_NUM\_THREADS=4 ./1.hello

2.hello.c:

1. **Is the execution of the program correct? (i.e., prints a sequence of "(Thid) Hello (Thid) world!" being Thid the thread identifier). If not, add a data sharing clause to make it correct?**

Adding private(id) makes the program correct, so the id variable won’t have data races

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

Lines will not always be printed in the same order. Messages appear like this because that’s the nature of parallel computing, we never know when a process will finish first and sometimes a thread printed “Hello” while a different thread is printing “Hello” so the first thread doesn’t have time to print “world” so it does so, afterwards

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 a parallel region the output is 1, inside a parallel region and without any method to supersede the number of threads environment variable the output is 8.
2. **Indicate the two alternatives to supersede the number of threads that is specified by the  
      
   OMP NUM THREADS environment variable.**One alternative is to state the number of threads with num\_threads(N) for example:#pragma omp parallel num\_threads(4). This statement will be the number of threads use within the scope of the pragma.

The other way to do it is by the instruction omp\_set\_num\_threads(N). This instruction will be the number of threads used if within parallel region and no #pragma omp parallel num\_threads(N) is used, which has priority. If outside parallel region, then threads will be 1.

1. **Which is the lifespan for each way of defining the number of threads to be used?**  
   The lifespan of num\_threads is within the scope of the pragma where it’s used in. With omp\_set\_num\_threads(N) it will be used until the variable is overwritten.

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)**

After first parallel region (shared): it seems that the value changes between 30 and 31, this could be due to data races, the processes read the value in tmp as 0 and add to that their thread number and write it to x.

After the second parallel (private): the value turns out to be 5 which makes sense because the variable is private inside its thread which means it won’t be updated in the outside scope of the program since they are completely different variables.

After the third parallel (firstprivate): the value is 5 again, for the same reason we mentioned. The difference is in the previous example, if we were to print out the values during each thread we would see all 0’s, that's due to x not being initialized at any point.

After the fourth parallel (reduction): It seems like the sum of all process numbers is done properly thats due to the fact that

5.datarace.c

1. **Should this program always return a correct result? Reason either your positive or negative answer.**It should not always return a correct result, because there are dataraces on the maxvalue by multiple threads, so they overwrite the maxvalue.
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.**

One solution would be to add reduction(max:maxvalue) so that all threads accumulate values to that variable but creates private copies of that variable so that there aren’t any data races, and finally it ensures that the shared variable is safely updated with all the thread's partial values.

Another possible solution would be to add #pragma omp critical above the if-clause so that the threads won’t access the same variable simultaneously, this solution however causes a big hit to the performance of the program.

1. **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).**

| #include <stdio.h> #include <unistd.h> #include <omp.h> #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=3;  omp\_set\_num\_threads(8);  #pragma omp parallel private(i) reduction(max:maxvalue)  {   int id = omp\_get\_thread\_num();  int howmany = omp\_get\_num\_threads();  int n\_each = N/howmany;   for (i=id\*n\_each; i < (id+1)\*n\_each; i+=howmany) {  if (vector[i] > maxvalue)  {   sleep(1); // this is just to force problems  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; } |
| --- |
|  |

6.datarace.c

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

No it shouldn’t that’s because all threads can execute the instruction countmax++ simultaneously which can lead to dataraces.

1. **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. (Es pot fer servir critical)**

A possible solution would be to add a #pragma omp atomic so that countmax is not accessed simultaneously on different processors, but that wouldn’t be the most optimal solution.

Another possible solution, and a better one at that, would be to add “reduction(+:countmax)” so that each thread accumulates their respective countmax in a private variable, and then it updates safely the total sum of countmax.

## Day 2: Explicit tasks

**1.single.c**

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

It causes the different threads to execute without waiting for each other, while thread 0 executes the first iteration, thread 1 executes the next one and so on for all the threads.

1. **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?**

Because the nowait means that the threads do not have to wait for each other and the iterations can be distributed between threads (#pragma omp parallel is called outside the loop and #pragma omp single nowait is called inside the loop so every available thread will execute it independently.

**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?**

No parallelism is being created (#pragma omp parallel). Only one thread is creating tasks and only this one will execute them.

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

We use #pragma omp parallel before the loop. We use the #pragma omp single directive so that one thread executes every task.

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

Specifies that variables must be independent between threads, and that the variable has to be initialised with the value of the variable that exists before the parallel construct.

It produces a datarace on it because all threads are taking the same variable.

**3.taskloop.c**

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

Loop1: the distributing thread divides the number of iterations between the grainsize. The distributing iterations are consecutive. We have 12 iterations and the value of the grainsize is 4, so the distributing thread will create 3 tasks of 4 iterations.

Loop 2: number of iterations is divided by the number of tasks specified. These ones are consecutively exexcuted. Here we have 12 iterations and the number of threads is 4, so the distributing thread will create 4 tasks of 3 iterations.

1. **Change the value for grainsize or num\_tasks to 5. How many iterations is now each thread executing? How is the number of iterations decided in each case?**

In the first loop, each thread executes 6 iterations because the grainsize clause divides the iterations between the value of grainsize, but the grainsize can be bigger if the rest of the division is different to 0.

In the second loop, threads execute 2 or 3 iterations. The number of tasks is fixed to 5, so there will always be five tasks, unless there are more tasks than iterations. Now we have the same number of tasks as iterations. As we have 12 iterations of 5 tasks, he will have 2 tasks of 3 iterations and 3 tasks of 2 iterations.

1. **Can grainsize and num\_tasks be used at the same time in the same loop?**

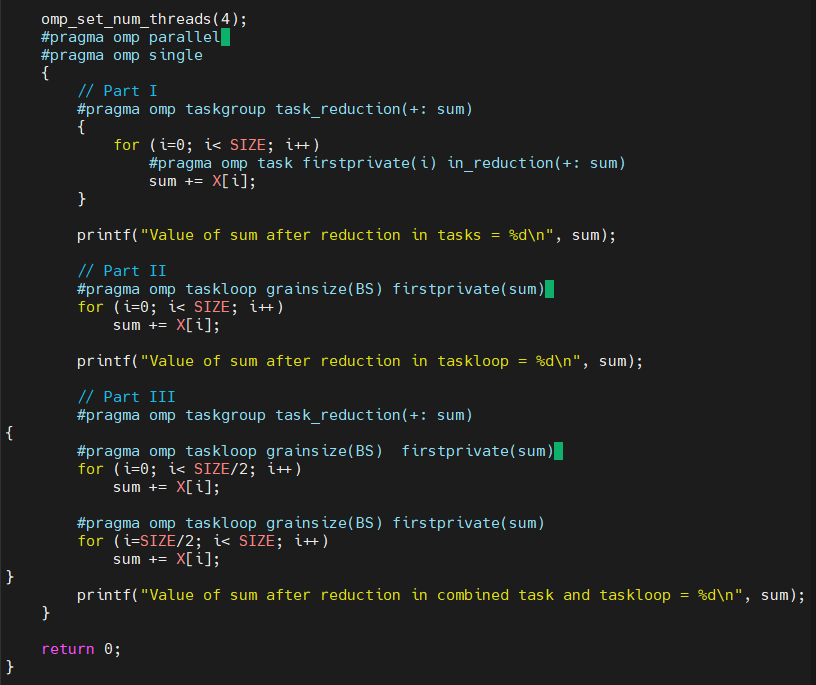
Yes. In this case loop is going to be divided into as many tasks specified with the num\_task clause and these tasks will be as big as the grainsize clause says. This happens in case that number of iterations is bigger than num\_tasks and grainsize. Otherwise they would get the value of the number of iterations.

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

The nogroup clause removes the implicit taskgroup made in the construct, so each thread takes the first task found.

**4.reduction.c**

1. **Complete the parallelization 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.**



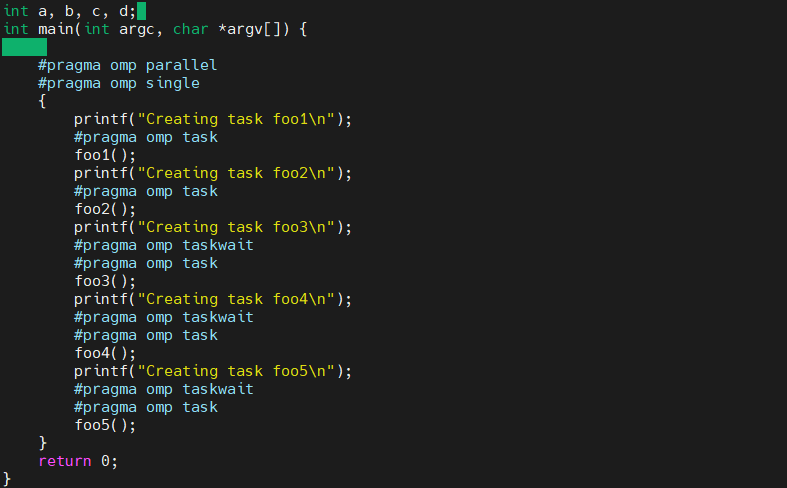
**5.synchtasks.c**

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

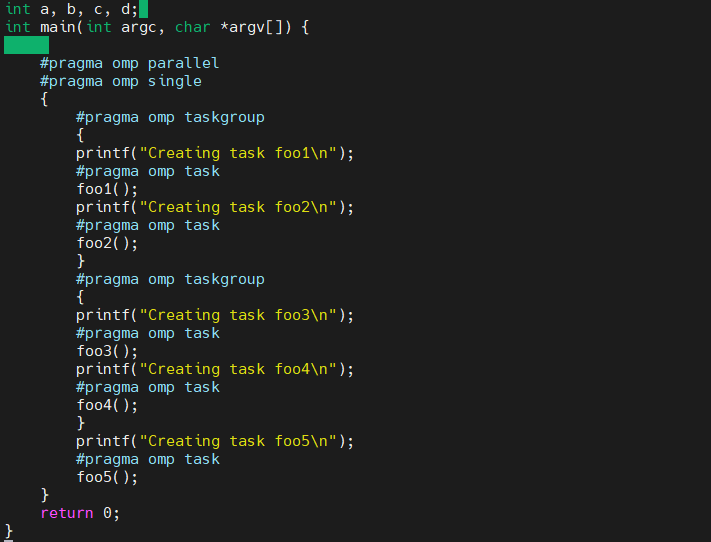
A, B and C are independent. Task D depends on A and B, so it starts when they finish. Task E depends on D and C, so it waits until both finish to be executed.



1. **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.**



1. **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.**



## 

## Overheads part 1

| par2107@boada-6:~/lab2/overheads$ cat pi\_omp\_critical-100000000-1-boada-11.txt  Total overhead when executed with 100000000 iterations on 1 threads: 1860542.0000 microseconds  par2107@boada-6:~/lab2/overheads$ cat pi\_omp\_atomic-100000000-1-boada-11.txt  Total overhead when executed with 100000000 iterations on 1 threads: 12314.0000 microseconds  par2107@boada-6:~/lab2/overheads$ cat pi\_omp\_sumlocal-100000000-1-boada-11.txt  Total overhead when executed with 100000000 iterations on 1 threads: -4451.0000 microseconds  par2107@boada-6:~/lab2/overheads$ cat pi\_omp\_reduction-100000000-1-boada-11.txt  Total overhead when executed with 100000000 iterations on 1 threads: -1294.0000 microseconds  par2107@boada-7:~/lab2/overheads$ cat pi\_omp\_critical-100000000-4-boada-12.txt  Total overhead when executed with 100000000 iterations on 4 threads: 49152843.0000 microseconds  par2107@boada-7:~/lab2/overheads$ cat pi\_omp\_atomic-100000000-4-boada-11.txt  Total overhead when executed with 100000000 iterations on 4 threads: 7382548.7500 microseconds  par2107@boada-7:~/lab2/overheads$ cat pi\_omp\_sumlocal-100000000-4-boada-11.txt  Total overhead when executed with 100000000 iterations on 4 threads: 11255.7500 microseconds  par2107@boada-7:~/lab2/overheads$ cat pi\_omp\_reduction-100000000-4-boada-11.txt  Total overhead when executed with 100000000 iterations on 4 threads: 4378.2500 microseconds   par2107@boada-7:~/lab2/overheads$ cat pi\_omp\_reduction-100000000-8-boada-11.txt  Total overhead when executed with 100000000 iterations on 8 threads: 17996.5000 microseconds  par2107@boada-7:~/lab2/overheads$ cat pi\_omp\_sumlocal-100000000-8-boada-11.txt  Total overhead when executed with 100000000 iterations on 8 threads: 14555.1250 microseconds  par2107@boada-7:~/lab2/overheads$ cat pi\_omp\_atomic-100000000-8-boada-11.txt  Total overhead when executed with 100000000 iterations on 8 threads: 9629231.5000 microseconds  par2107@boada-7:~/lab2/overheads$ cat pi\_omp\_critical-100000000-8-boada-11.txt  Total overhead when executed with 100000000 iterations on 8 threads: 41924440.7500 microseconds  par2107@boada-7:~/lab2/overheads$ cat pi\_sequential-100000000-8-boada-11.txt  Wall clock execution time = 1.246910095 seconds  par2107@boada-7:~/lab2/overheads$ cat pi\_sequential-100000000-4-boada-11.txt  Wall clock execution time = 1.247421980 seconds  par2107@boada-7:~/lab2/overheads$ cat pi\_sequential-100000000-1-boada-11.txt  Wall clock execution time = 1.245892048 seconds |
| --- |

|  | 1 Thread | 4 Threads | 8 Threads |
| --- | --- | --- | --- |
| Critical | 1860542.0000 microseconds | 49152843.0000 microseconds | 41924440.7500 microseconds |
| Atomic | 12314.0000 microseconds | 7382548.7500 microseconds | 9629231.5000 microseconds |
| Sumlocal | 4451.0000 microseconds | 11255.7500 microseconds | 14555.1250 microseconds |
| Reduction | 1294.0000 microseconds | 4378.2500 microseconds | 17996.5000 microseconds |

**1. If executed with only 1 thread and 100.000.000 iterations, do you notice any major overhead in the**

**execution time caused by the use of the different synchronisation mechanisms? You can compare**

**with the baseline execution time of the sequential version in pi sequential.c.**

Foremost, let’s analyse the instructions of each program that will cause overheads. In the pi\_omp\_critical.c program, we will have to perform a #pragma omp critical at each iteration of the loop, which is costly. In pi\_omp\_atomic.c we will have to perform an #pragma omp atomic at each iteration which is costly but not as much. With pi\_omp\_sumlocal and pi\_omp\_reduction we will have a #pragma omp critical at the end of the loop so that they can synchronize.

The overhead that is most notable through the execution is the pi critical version and the atomic version, this is due to the system calls that the program has to do, this causes the program to behave sequentially. Whilst, sumlocal and reduction have much less overhead.

**2. If executed with 4 and 8 threads and the same number of iterations, do the 4 programs benefit**

**from the use of several processors in the same way? Can you guess the reason for this behaviour?**

Not all programs benefit equally from the increase in the number of threads, this is most likely due to the overheads especially with the pi\_omp\_critical.c and with a little less overhead the pi\_omp\_atomic.c also these programs are executed sequentially due to the critical and atomic instructions.

On the other hand, the pi\_omp\_reduction and pi\_omp\_sumlocal.c benefit greatly from an increase in threads, this is because the threads only wait unlike the other two programs until the for loop is over when they have to do the reduction of the separate private results of each thread, this only has a single critical call, making it much faster.

## Overheads part 2

Task creation and synchronization

Now we’ll be using the pi\_omp\_tasks.c file. This one computes the difference between executing the program sequentially or parallely using a certain number of threads. This difference is printed in microseconds, and this is the overhead of the parallelization.

Observing the results we see that overhead does not depend on the number of tasks, because it’s always quite similar.

