POLYTECHNIC UNIVERSITY OF CATALONIA

Lab 2

Brief tutorial on OpenMP programming

DELIVERABLE

PAR

Authors

LIANGWEI DONG (PAR 4202)
DAVID LATORRE ROMERO (PAR 4211)

2022-23 Q1

Day 1: Parallel regions and implicit tasks

1.hello.c:

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

   We will see the message twice, because 2 threads will be created (that's because if we do not specify the number of threads to be created, then by default openMP creates the same number of threads as cores our system has), and the instruction printf is inside the #pragma omp parallel.

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

   We use the command "OMP_NUM_THREADS=4 ./1.hello" to execute the program and change the number of threads to 4. This way, the message will be printed 4 times.

2.hello.c:

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

   The execution is not correct because *id* is a shared variable and it causes dataraces. We have to add the *private(id)* clause after *#pragma omp parallel* to make the variable *id* private for each thread.

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, because there's no order of execution. The messages appears inter-mixed because sometimes a thread makes a print between prints of another thread. Because we are using private(id), we are able to compute the printf instructions in a total parallelized way, and therefore the lines will be printed according to which thread has executed a printf instruction before or after another.

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_num_threads* return when invoked outside and inside a parallel region?

Outside a parallel region it always returns 1 because there's only thread. Inside a parallel region it returns the number of threads defined.

In this case, we have defined the number of threads to 8, but on some occasions the code changes the number, and because the statement OMP_NUM_THREADS=8 is the one that has less priority, that's why we see different numbers.

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

   We can do it adding *num_threads(nthreads)* after *#pragma omp parallel* clause or calling the *omp_set_num_threads(nthreads)* function before the parallelism, being *nthreads* the number of threads.

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

   The lifespan for the first way (*num_threads(nthreads)*) is the parallel region defined with the *#pragma omp parallel* clause.

   The second way (*omp_set_num_threads(nthreads)*) modifies the OMP_NUM_THREADS variable so it keeps that value until being modified again or the program ends.

4.data_sharing.c:

1. 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? (Execute several times if necessary)

   Shared: The value of x is not the same for each execution because being x a shared variable, we have datarace, so the result is not correct. We expect x to be 496 (because the triangle number of 31 (0+1+2+3…+31) is 496), but we get different results each execution, typically low results.

   Private: We expect x to be 5 and it is correct for all the executions, because x is private for each parallel region and its value does not change, because at the end all private variables are destroyed, and so the real x variable is not modified.

   First-private: We expect x to be 5 and it is correct for all the executions for the same reason as **Private**.

   Reduction: We expect x to be 496 (triangle number of 31) and it is correct for all the executions because reduction avoids datarace on the variable x.

5.datarace.c:

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

   No, it should not. The fact that every thread can modify the variable maxvalue at the same time could cause that maxvalue doesn't end up having the maximum value of the vector, because there is datarace on 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 could be to write **#pragma omp critical** just before the conditional inside the loop. This way we make sure there is not datarace because there is only one thread at the same time checking the condition and writing to maxvalue. This solution is not effective at all, in fact, we end up doing the execution like the sequential way.
   - We could make a reduction on the variable **maxvalue** adding **reduction(max:maxvalue)** into the line of code that has **#pragma omp parallel private(i)**. This way we only do not have datarace, but we also can benefit more from the parallelism.

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).

```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=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();

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

Note that this program only works correctly when the size of the vector is divisible by the number of threads.

6.datarace.c:

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

   No, it should not, because every thread can write on the variable countmax at the same time, and therefore there's datarace on this variable. Imagine the case where the variable countmax has the value "1", and at the same time two threads increment the value of countmax. This will produce a value of "2" for countmax, which is incorrect, because the value should be "3".

2. Propose two alternative solutions to make it correct, without changing the structure of the program (just using directives or clauses) and never making use of critical. Explain why they make the execution correct.
   - Adding **#pragma omp atomic** just before countmax++. This way we make sure countmax is modified in a synchronized way. We can use atomic because we're only computing an addition on the variable. Also, atomic is faster than critical.
   - Making a reduction on the variable countmax, adding **reduction(+:sum)** on the line that has **#pragma omp parallel private(i)**. This way countmax acts as a private variable for every thread, and at the end it makes the addition for all the threads. Also, this way is more effective than using **atomic**, because we can get rid of a lot of synchronization wait times.

Day 2: explicit tasks

1.single.c:

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

   It makes the others threads that aren't executing that single clause keep executing other instructions at the same time, this way they don't have to wait for the thread that is running the single clause.

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?

The reason why all the threads contribute to the execution of the multiple instances of singles is because with “nowait” the remaining threads doesn’t have to wait for the current iteration of “single” to end, and therefore they can execute the next iterations at the same time.

The instances appear in burst because after the instruction “printf” there’s a sleep time of 1 second for the thread that is executing that iteration. That’s why they appear in bursts, because there’s a period of time where all the threads are in that sleep instruction.

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?

   The program is not being executed in parallel because we’re not writing **#pragma omp parallel** anywhere, and therefore the program is never gonna run with more than one thread.

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

```
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;
}
```

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

   If we comment it we get a Segmentation fault (core dumped).

   The firstprivate(p) clause is defining the variable p as private for every thread, and initializing it with the value that p has originally when the task is created. This way, every task has the correct node* to calculate the fibonacci sequence.

3.taskloop.c:

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

On the first loop we declare the taskloop with grainsize, which specifies the number of iterations per task. With that said, because the value of N is 12 and the value of VALUE is 4, on the first loop we do 4 iterations per task, so a thread will do 4 iterations on every task. Because the length of the vector is 12, (12/4=3), only 3 threads at maximum will execute iterations of the loop.

On the second loop, we declare the taskloop with num_tasks, which specifies the number of tasks to create for the execution of the loop. Because we create 4 tasks, the 4 threads at maximum will execute the iterations of the loop. Also, because the loop has length 12, (12/4=3), every thread will do three iterations when it executes a task.

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

Now, because (12/4) is not an exact division, at the first loop we are not gonna be able to execute 5 iterations on each task. What grainsize will do is create some tasks with a close number of iterations as 5. Observing the execution of this program, we see that it usually creates 2 tasks of 6 iterations.

On the other hand, on the second loop, although we can create 5 tasks for the execution of the loop, the number of iterations each task is gonna execute will not be the same for each of them.

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

No, the compiler logically does not allow that, because, on one hand, specifying the number of tasks to create also establishes the number of iterations per task, and on the other hand, specifying the number of iterations per task also establishes the number of tasks to create.

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

Uncommenting the nogroup clause, we override the implicit taskgroup associated with the taskloop construct, and therefore the two loops are executed at the same time.

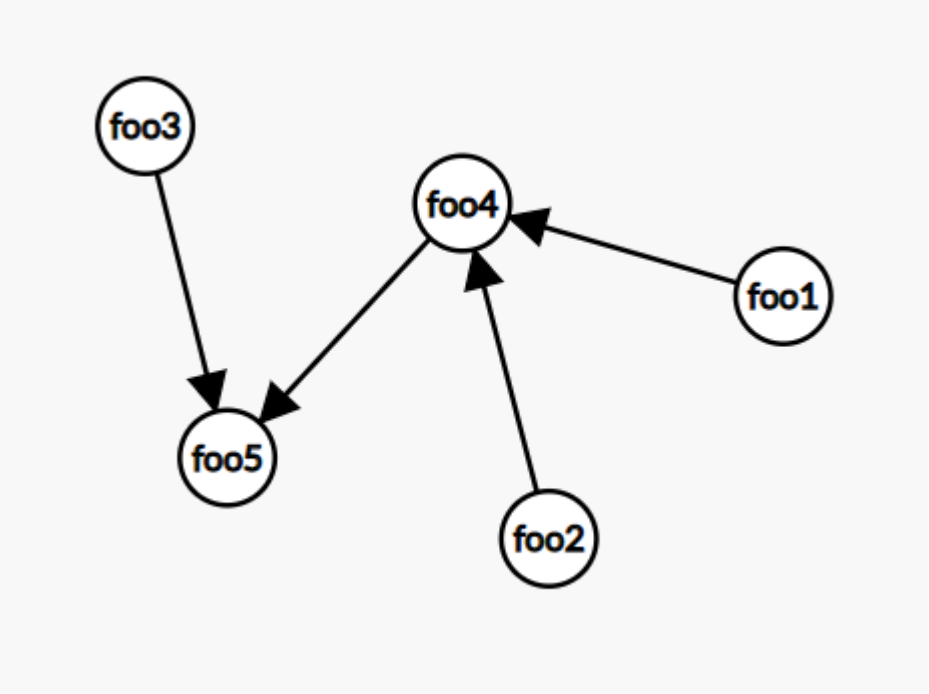
4.reduction.c:

1. Complete the parallelisation of the program so that the correct value for variable sum is returned in each printf statement. Note: in each part of the 3 parts of the program, all tasks generated should potentially execute in parallel.

```
    // 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) reduction(+: 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)
    {

        for (i=0; i< SIZE/2; i++)
            #pragma omp task firstprivate(i) in_reduction(+: sum)
            sum += X[i];

        #pragma omp taskloop grainsize(BS) in_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;
```

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.

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

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.

```
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;
}
```

Observing overheads

1. Synchronization overheads:

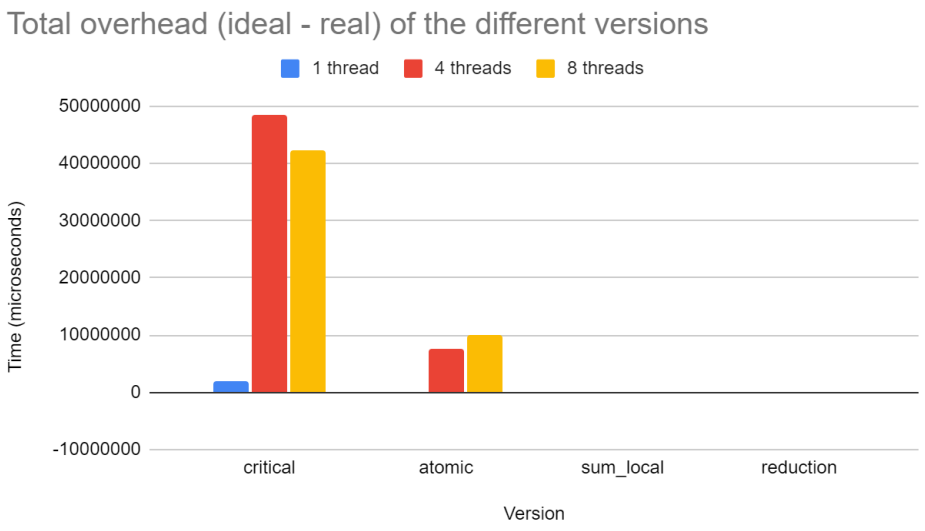
1. If executed with only 1 thread and 100.000.000 iterations, do you notice any major overhead in the execution time caused by the use of the different synchronisation mechanisms? You can compare with the baseline execution time of the sequential version inpisequential.c.

To respond at this questions, let's execute the 4 versions with the **submit-omp.sh** script, and visualize the data on a plot to have a better understanding.

The data we have collected (Total overhead (ideal - real) of the different versions in microseconds):

Version	1 thread	4 threads	8 threads
critical	1.861.740	48544431	42375656
atomic	-4429	7602126	10037907
sum_local	-4741	10542	14787
reduction	-2611	3883	15364

The plot:



The first thing that catches our attention in this plot is the big overhead (for both 1,4,8 threads) of the critical version in comparison with all the other ones, which is logical, because with pragma omp critical we are continuously forcing threads to wait when they want to update the variable sum.

We also observe a big overhead for the atomic version in comparison with the sum_local and the reduction one for the same reason, although with atomic we get smaller overheads than with critical because allowing only simple operations makes it more efficient.

It is also interesting to see (in the table), that while sum_local only experiences an increment of approx 4000 microseconds with 8 threads in comparison with 4 threads, the reduction version experiences a much bigger increment, making it to benefit less from 8 threads.

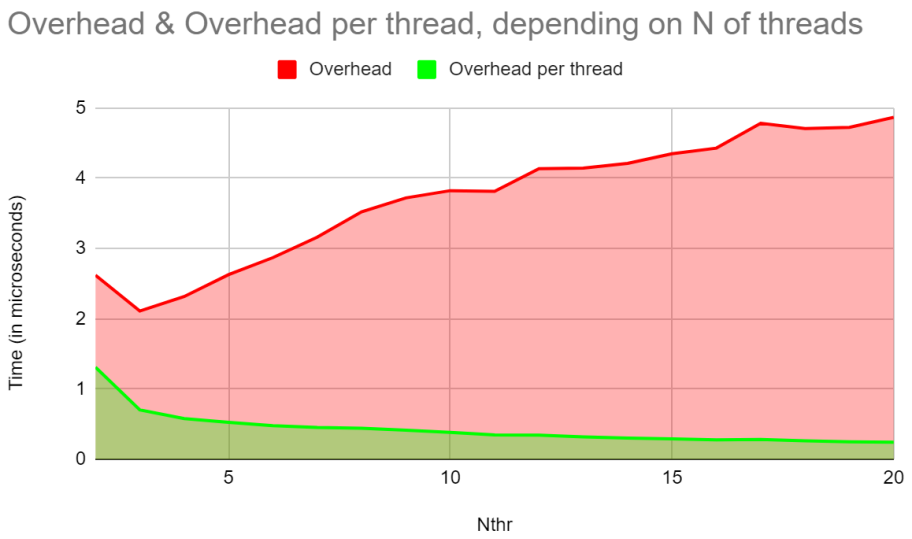
2. Thread creation and termination

1. How does the overhead of creating/terminating threads varies with the number of threads used? Which is the order of magnitude for the overhead of creating/terminating each individual thread in the parallel region?

To respond at this question, let's queue the command **sbatch submit-omp.sh pi_omp_parallel 1 20** to execute the program with 1 iteration and a maximum of 20 threads, and observe at the data it produces:

Nthr	Overhead	Overhead per thread
2	2,6194	1,3097
3	2,1096	0,7032
4	2,3154	0,5788
5	2,6272	0,5254
6	2,8693	0,4782
7	3,1606	0,4515
8	3,5219	0,4402
9	3,7196	0,4133
10	3,823	0,3823
11	3,8143	0,3468
12	4,1348	0,3446
13	4,1446	0,3188
14	4,2104	0,3007
15	4,3488	0,2899
16	4,4281	0,2768
17	4,7809	0,2812
18	4,708	0,2616
19	4,724	0,2486
20	4,8657	0,2433

Let's visualize this data with a plot:



On the plot we can see the relation of the overhead and overhead per thread depending on the number of parallel regions we create.

We can see a notable increasing tendency on the overall overhead as we create more parallel regions, which makes sense, because as more threads there are, more time it takes to create all of them, or synchronize them, etc.

On the other hand, the overhead per thread slowly diminishes as we create more threads.

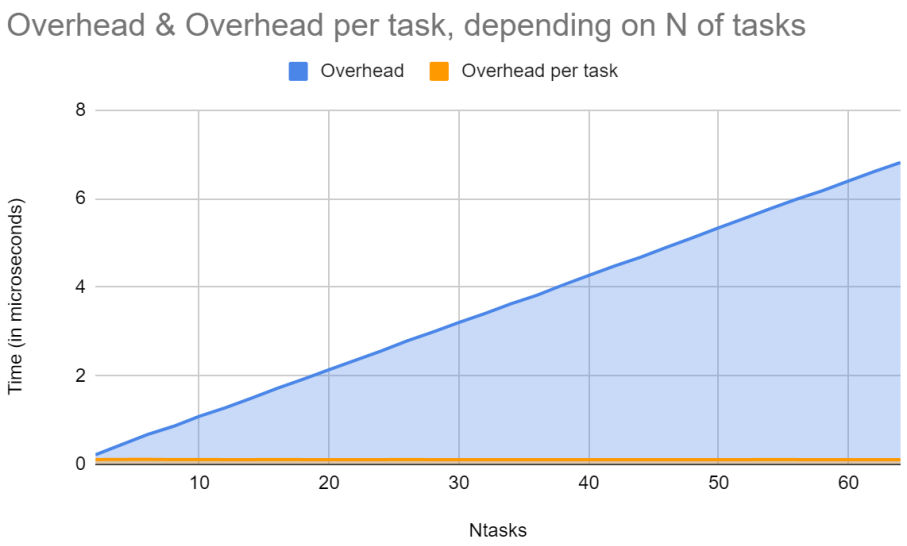
3. Task creation and synchronization

1. How does the overhead of creating/synchronising tasks varies with the number of tasks created? Which is the order of magnitude for the overhead of creating/synchronising each individual task?

To respond at this question, let's queue the command **sbatch submit-omp.sh pi_omp_tasks 10 1** to execute the program with 10 iterations and one thread, and observe at the data it produces:

Ntasks	Overhead	Overhead per task
2	0,2164	0,1082
4	0,444	0,111
6	0,6741	0,1124
8	0,8582	0,1073
10	1,0841	0,1084
12	1,2775	0,1065
14	1,4943	0,1067
16	1,7177	0,1074
18	1,9219	0,1068
20	2,1365	0,1068
22	2,3505	0,1068
24	2,5601	0,1067
26	2,7874	0,1072
28	2,9886	0,1067
30	3,2041	0,1068
32	3,4051	0,1064
34	3,6273	0,1067
36	3,8176	0,106
38	4,0492	0,1066
40	4,2685	0,1067
42	4,4805	0,1067
44	4,68	0,1064
46	4,902	0,1066
48	5,1197	0,1067
50	5,3405	0,1068
52	5,5551	0,1068
54	5,7755	0,107
56	5,9852	0,1069
58	6,1799	0,1066
60	6,401	0,1067
62	6,6126	0,1067
64	6,8143	0,1065

Like before, let's visualize the data with a plot:



On the plot we can see the relation of the overhead and overhead per thread depending on the number of tasks we create.

In comparison with the previous version, in which we analyzed the overhead depending on the number of parallel regions we create, now we can see a much more stable increasing tendency of the overall overhead as we create more tasks.

This is due to the fact that the overhead per task is almost constant, the same, no matter how many tasks we create.

This gives us an interesting conclusion, in which we can say that while the tendency of the overall overhead as we create both more tasks or threads is to increase, when we create tasks it increases with in a much more linear way, and on the other hand when we create threads the slope of this increasement slowly diminishes as we create more threads, because the overhead per thread also decreases.