Lab2: OpenMP programming model

and analysis of overheads

par2103; Joan Sánchez García, Marc Domínguez de la Rocha

Fall 2017-2018 7/11/17

Optional

Version	Execution time	Speed-Up
v0 (Sequential)	0.789199996 seconds	1
v1	0.829151416 seconds	0.9518
v2	0.823044346 seconds	0.9588
v3	0.216566546 seconds	3.6441
v4	16.487510697 seconds	0.0478
v5	7.992015004 seconds	0.0987
v6	0.211069823 seconds	3.7390
v7	0.209028284 seconds	3.7755
v8	0.210493538 seconds	3.7492
v9	7.869410455 seconds	0.1003
v10	0.208294766 seconds	3.7888
v11	0.215906106 seconds	3.6553
v12	0.213290923 seconds	3.7001
v13	0.233169721 seconds	3.3846
v14	0.809189714 seconds	0.9753
v15	0.404229823 seconds	1.9523
v16	123.736600215 seconds	0.0063
v17	0.250713630 seconds	3.1478

In the ***v1*** version, although we define the parallel clause, we don't look the loop control variables, so in each iteration, each thread will change the value and the result and the number of iterations will be unknown, increasing the execution time.

In ***v2*** version, we protect the loop control variables, but now all the threads execute the whole loop because we define it parallel but we don't split it in parts. So all threads are doing the same amount of work instead of sharing it.

In ***v3*** version, although we should protect the sum variable against data race conditions, we split the loop work between the threads making that each thread executes just a portion of the total work. With this tweak, we improve the execution time considerably.
v4 version provides a correct result, but the critical clause makes that only one thread at a time can execute that code region. So for more threads that can be executing our code at the same time, if just one can be modifying one of its parts we lost all the parallel power.

v5 version replaces the critical clause with an atomic clause. The atomic clause ensures that a code region is accessed atomically, avoiding the possibility of multiple, simultaneous reading and writing threads, but don't block them, so all other threads can be executing the rest of the code. This improves the v4 approach.

v6 version makes use of the reduction clause, which makes a private copy of the sum variable for each thread and combine them all at the end of the execution,, allowing that more than one thread can be executing all the code at the same time. This version improves by far the execution time and provides a correct result.

v7 is the same as v6 just changing the iterations distribution by the schedule static clause. This clause breaks the iteration space in chunks of approximately size N/4 (4 is the number of threads we are using). So thread 0 will execute from iteration 0 to N/4-1. Speed Up is nearly the same.

v8 is the same as v7 just intervalling the iterations. Same result as the version before.

v9 is a recession. If in v7 and v8 we gained in speed up by distributing the loop weigh equitably among all the threads, with the dynamic schedule we lost it. This type of distribution make threads grab chunks of 1 iterations (as we have not defined a larger N) until all iterations have been executed, but with no established order.

v10 works similar to v7. It's a dynamic assignation but the size of the chunks it's big enough as to each thread could perform a sequence of iterations consecutively. For example, a thread can perform iteration 0,1,2,3,4 as if it was a unique one, because the thread grabs them together. It improves the execution time.

v11 works similar to v10, but the size of the chunks decreases as the threads grab iterations (at least of size N). Execution time is the same as v10.

v12 is the same as v11 just adding a single clause to a region of the code. This region will be executed by just one processor, but we can't assure that it will be executed when all threads have finished the loop, that's why the result is incorrect. As it is just a line with the single clause, execution time is not affected.

v13 just moves the reduction clause to ensure that the code affected by the single one gets the correct result. Execution time is more or less the same.

v14 computes the problem using tasks. As we define that the computational region is parallel and that each loop will be splitted in tasks, each thread replicate all the tasks, so we are creating all of them for each thread. Moreover, the taskwait make us wait until all of them have finished. The execution time increases.

In ***v15*** we correct the v14 problem making that the task definition is only performed by just one thread, but as we define the whole loop as one task, only two processors can be working (one in loop 1 and the other in loop 2). So we obtain a better execution time than the sequential code, but could be higher if all threads could work.

v16 presents a correct code but with a very small task granularity. As we define a new task for each iteration of each loop we are creating N tasks, with the overhead that entails.

v17 in contrast to v16, makes use of the taskloop clause, which generate a task for a certain number of iterations. Some iterations are joined into the same task so the granularity improves.

To sum up, we have seen that not all parallel implementations improve our execution time because not all of them take care of data race conditions, the waitings and the parts of the code that must be accessed at the same time by more than just one thread.
Then when we were parallelizing our code we need to have in mind that we don't need to make our code as small that executing it will take longer time, we need to break it in parts big enough as to not increase the execution time and small enough as we can take advantage of the parallelization process.

Part I: OpenMP questionnaire

A) Basics

1.hello.c

1. *How many times will you see the "**Hello world!**" message if the program is executed with "**./1.hello**"?*
   As many as threads we have assigned.

2. *Without changing the program, how to make it print 4 times the "**Hello World!**" message?*
   Assigning just 4 threads to our OpenMP program. This can be done executing the program this way:

   OMP_NUM_THREADS=4 ./1.hello

   Where OMP_NUM_THREADS is an environment variable modified this time.

2.hello.c

1. *Is the execution of the program correct? Which data sharing clause should be added to make it correct?*
   No, because the "id" variable is **shared** by default and gets rewritten every execution due to race conditions.
   We can correct it adding the **firsprivate**(id) clause.

2. *Are the lines always printed in the same order? Could the messages appear intermixed?*
   No, because in concurrency (in this configuration) we don't have control over the execution order, which also corresponds with the second question.

3.hello.c

1. *How many "**Hello world …**" lines are printed on the screen?*
   Sixteen.

2. *If the if(0) clause is commented in the last parallel directive, how many "**Hello world …**" lines are printed on the screen?*
   Between 1 and 4 (*n%4 + 1 = {1, 2, 3, 4}*).

4.data_sharing.c

1. *Which is the value of variable **x** after the execution of each parallel region with different data-sharing attribute?*
   For **shared**, the value of x is the number of OMP_NUM_THREADS (supposing that there are no race conditions). For **private**, the value of "x" is unknown: "x" is not initialized inside the scope of each thread, so we don't know which value will result. For **firstprivate**, the value of "x" will be 1, because "x" is initialized to 0, and each thread sums one to it in its local scope, but as the printf is outside the threads scope, it will show the global "x" variable, which is always 0.

2. *What needs to be changed/added/removed in the first directive to ensure that the value after the first parallel is always 8?*
   We ensure that there are no race conditions above our parallel region by doing a **reduction** to the sum operation over "x":

   #pragma omp parallel reduction (+: x)

5.parallel.c

1. *How many messages the program prints? Which iterations is each thread executing?*
   We can't know it, it's random in both questions due to the "i" variable is shared.

2. *What needs to be changed in the directive to ensure that each thread executes the appropriate iterations?*
   Adding the **firstprivate**(i) directive.

6.datarace.c

1. *Is the program always executing correctly?*
   No. As *x* is shared among all the threads and all of them increase its result, data race conditions can make that when one of them uses the *x* variable, it wasn't updated yet, so it is working with an outdated value.

2. *Add two alternative directives to make it correct. Which are these directives?*
   We can use the atomic or critical directive to solve the data race condition problems.

7.barrier.c

1. *Can you predict the sequence of messages in this program? DO threads exit from the barrier in any specific order?*
   Not at all, we can't predict the order in the sleeping process either in the barrier exit

but in the awaking process it's always the same sequence.

B) Worksharing

1.for.c

1. *How many iterations from the first loop are executed by each thread?*
   Each thread executes two iterations of the first loop.

2. *How many iterations from the second loop are executed by each thread?*
   Each thread executes two iterations except for the first three threads, which executed three iterations each one (to assume the rest of the load)

3. *Which directive should be added so that the first printf is executed only once by the first thread that finds it?*
   Adding the "single" directive we can ensure that only one thread will execute the code following it.

```
omp_set_num_threads(8);
#pragma omp parallel
{
   #pragma omp single
   printf("Going to distribute iterations in first loop ...\n");
   #pragma omp for
   for (i=0; i < N; i++) {
         int id=omp_get_thread_num();
         printf("(%d) gets iteration %d\n",id,i);
   }
}
```

2.schedule.c

1. *Which iterations of the loops are executed by each thread for each* ***schedule*** *kind?*
   For the *Loop1,* which is of **static** kind, we have an equitative load distribution, which means that every thread will execute *N/num_threads* iterations.

   For the *Loop2* which is of **static interleaved** kind, so for 12 iterations with 3 threads we have this distribution:

	First Assignation	Second Assignation
t0	0, 1	6, 7
t1	2, 3	8, 9
t2	4, 5	10, 11

For the *Loop3* which is of **dynamic** kind, the assignation of workload to each thread is done dynamically, which means it depends on each execution.

3.nowait.c

1. *How does the sequence of **printf** change if the **nowait** clause is removed from the first **for** directive?*
   Then the sequence of **printf** is about first all the bunch of "Loop 1: …" prints and right after all the "Loop 2: …" prints, without any order within every bunch of prints.

2. *If the **nowait** clause is removed in the second **for** directive, will you observe any difference?*
   Not at all, every thread that ends the first loop will just go ahead and do its workload in the second loop.

4.collapse.c

1. *Which iterations of the loops are executed by each thread when the collapse clause is used?*
   Each thread (from 0-7) executes at least 3 iterations, except of the thread 0, that must deal with one extra iteration because **25%8 == 1**.

2. *Is the execution correct if we remove the collapse clause?*
   Add the appropriate clause to make it correct.
   No.
   *#pragma omp for **ordered***

5.ordered.c

1. *How can you avoid the intermixing of **printf** messages from the two loops?*

By removing the **nowait** from the first loop.

2. *How can you ensure that a thread always executes two consecutive iterations in order during the execution of the first loop?*
   By doing this in both loops (change the schedule and add ordered).
   ```
   #pragma omp for schedule(static, 2) ordered
   for (i=0; i < N; i++) {
       int id=omp_get_thread_num();
       #pragma omp ordered
       printf("Loop 1 - (%d) gets iteration %d\n",id,i);
   }
   ```

6.doacross.c

1. *In which order are the "Outside" and "Inside" messages printed?*
   The Outside randomly, but the inside in order of i. This happens thanks to the #pragma omp ordered depend(sink: i-2), that defines the wait point for the completion of computation in iteration i-2.

2. *In which order are the iterations in the second loop nest executed?*
   If we imagine the iterations as a 5x5 matrix, each space depends on the one on the left (i, j-1) and the one above (i-1, j). So to be able to compute an iteration we must wait to be finished the one with i = i-1 and j = j and i = 1 and j = j-1.
   Starting with 1 1 (which is the one with no active dependences), the two next that can be performed are 1 2 and 2 1, following the direction of the inner loop. Once 1 2 is performed, 1 3 and 2 2 will not have dependences, and so on. So we have to imagine this as a queue: when an iteration is performed, the two which depended from it get into the queue.

3. *What would happen if you remove the invocation of **sleep(1)**. Execute several times to answer in the general case.*
   The order we had is not respected, the fastests thread execute one of the two possible iterations with no dependencies, thus altering the result

C) Tasks

1.serial.c

1. *Is the code printing what you expect? Is it executing in parallel?*
   Yes, but is not executing in parallel, we have just created a **task** but not initialized a parallel region or something like that.

2.parallel.c

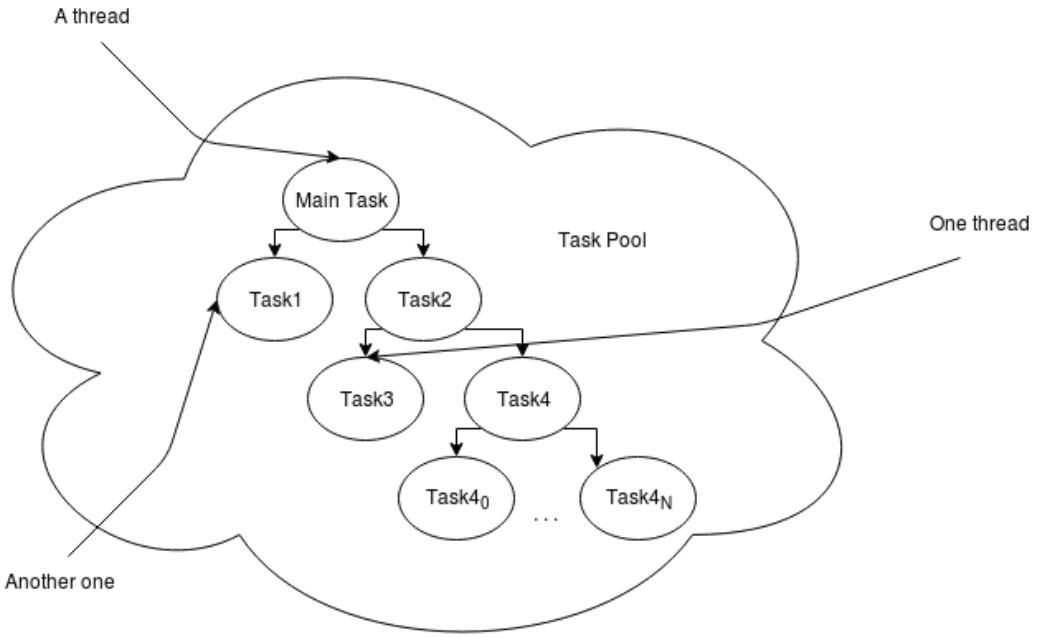
1. *Is the code printing what you expected? Is it executing in parallel? What is wrong with it?*
   No. The code is executing in parallel but without any control above the workflow or the workload, so each thread is creating tasks, which will be created one time per thread, working with the same list, clobbering data previously other thread could have calculated, in conclusion, miscalculating.

2. *Add a directive to make its execution correct.*
   Add #pragma omp single just after the parallel directive.
   Then only one thread will create the tasks.

3. *What would happen if you remove the firstprivate clause in the task directive? And if you ALSO remove the first private clause in the parallel directive? Why are they redundant?*
   The code works fine, just as expected.
   Then a wrong behaviour will issue, by default in the *OpenMP* model, all variables are shared, so all the threads will be working with the *p* pointer, which will broke the semantics of our loop. The two *firstprivate* define the same, an already initialized private variable *p* for each thread, that is why they are redundant.

3.taskloop.c

1. *Execute the program several times and make sure you are able to explain when each thread in the threads team is actually contributing to the execution of work (tasks) generated in the taskloop.*
   We know that each task will be created by a thread, it could be the same or not, exactly the same when we are talking about who will execute those tasks.
   Due to the *#pragma omp single* directive, one thread will create tasks T1 and T2, lately, threads in the team will decide to execute tasks T1 a T2 (or the same thread), T1 is simply executed, but T2 is the creation of two more tasks, T3 and T4.
   Analogously to T1 and T2, the team will take part into the execution, T3 is executed and also T4 but again this will result in a thread creation burst.



A thread per task, they can be the same or not.

Part II: Parallelization overheads

1.-Which is the order of magnitude for the overhead associated with a parallel region (fork and join) in OpenMP? Is it constant? Reason the answer based on the results reported by the pi omp parallel.c code.

The order of magnitude is in microseconds, and no, it is not constant. This is due to the reservation and initialization of dynamic memory (plus the time to take other resources a thread needs) for each thread. As much threads we have, more time will take to reserve and initialize them, so in add to a constant time we have a variable time proportionally direct to the number of threads.

2.-Which is the order of magnitude for the overhead associated with the creation of a task and its synchronization at taskwait in OpenMP? Is it constant? Reason the answer based on the results reported by the pi omp tasks.c code.
The order of magnitude is 10^{-7} seconds and in this case it's constant, since the creation of a task is just taking a 'job' and putting it into a pool of tasks. The cost of doing the assignment of the code (which is a pointer to a code region) is constant. For the execution with one unique thread, there's no cost on synchronizations due to the code is being executing sequentially (we got only one thread).

3.-Which is the order of magnitude for the overhead associated with the execution of critical regions in OpenMP? How is this overhead decomposed? How and why does the overhead associated with critical increase with the number of processors? Identify at least three reasons that justify the observed performance degradation. Base your answers on the execution times reported by the pi omp.c and pi omp critical.c programs and their Paraver execution traces.

The order of magnitude is in microseconds. The overhead in this case is the sum of threads creation (**fork and join**), the **synchronization** between these threads, and the mutual exclusion (**I/O**). Increasing the number of threads implies increasing the threads you got to create, the synchronization and also the number of threads that will be waiting on a mutual exclusion zone, because it's just one **critical** for all that threads, they will be just idle.
So as we defined the overhead, composed by the thread creation, synchronization and waiting on a mutual exclusion zone, this are three reasons that justify the observed performance degradation.

4.-Which is the order of magnitude for the overhead associated with the execution of atomic memory accesses in OpenMP? How and why does the overhead associated with atomic increase with the number of processors? Reason the answers based on the execution times reported by the pi omp.c and pi omp atomic.c programs.

The overhead associated with *atomic* is proportional to the number of threads to execute, but in fact is less than the overhead produced by *critical* statements. With *Paraver* we can see that using *atomic* will reduce significantly the time to do the operations due to its assembly traduction to an instruction that allows us to do simple operation assuring that is executed sequentially, that means, that no one will interfere in such operations.

5.-In the presence of false sharing (as it happens in pi omp sumvector.c), which is the additional average time for each individual access to memory that you observe? What is causing this increase in the memory access time? Reason the answers based on the execution times reported by the pi omp sumvector.c and pi omp padding.c programs. Explain how padding is done in pi omp padding.c.

As we use caches, we have to maintain a certain memory coherence, so every time a thread modifies memory addresses stored in the same cache line as the rest of threads, every cache must make a flush (what literally means setting all the validity bits to zero).
This will have a big repercussion because a fast access to a cache will result in a slow access to principal memory.

The *padding.c* solves that problem, as before we were using a vector of doubles where elements of different threads took place in the same cache line (of different caches), now we assure that elements of different threads are in different cache lines, we improve the time to access memory significantly respect to the *pi_omp_sumvector.c* giving up some additional space per thread.

6.-Write down a table (or draw a plot) showing the execution times for the different versions of the Pi computation that we provide to you in this laboratory assignment (session 3) when executed with 100.000.000 iterations. and the speed–up achieved with respect to the execution of the serial version pi seq.c. For each version and number of threads, how many executions have you performed?

Version	Execution Time (1 threads)	Execution time (8 threads)	Speed-Up
pi_seq	0.790917s	-	1
pi_omp	0.788966s	0.119681s	6.5922
pi_omp_atomic	1.469643s	7.422147s	0.1980
pi_omp_critical	1.793926s	21.249610s	0.0844
pi_omp_padding	0.791591s	0.114421s	6.9182
pi_omp_sumvector	0.792127s	0.598915s	1.3226