Arquitectura de ordenadores. Practice 4.

Pablo Cuesta Sierra, Álvaro Zamanillo Sáez
(group 1392, team 04)

November 24, 2021

Contents

1 Exercise 1: Basic OpenMP programs **2**

2 Exercise 2: Parallelize the dot product **2**

3 Exercise 3: Parallel matrix multiplication **4**

1 Exercise 1: Basic OpenMP programs

1.1 *Is it possible run more threads than cores on the system? Does it make sense to do it?*

It is possible to do so: if we first execute omp1 without arguments to see the available cores and then we execute it again with a number of threads higher than the amount of cores, we will see that the execution is carried out without problems. Using more threads than cores may make no sense when the goal is simply to parrallelize a single task because the threads will interrupt each other. However, if there are several tasks to be run at the same time and any of them may provoke the thread to go asleep, then using more threads could help to take advantage of the time that particular thread is asleep.

1.2 *How many threads should you use on the computers in the lab? And in the cluster? And on your own team?*

For a general situation, the number of threads to be used is the one determined by the function *omp_get_max_threads()*. Indeed this value used by default.

1.3 *Modify the omp1.c program to use the three ways to choose the number of threads and try to guess the priority among them*

The priority order is: OMP_NUM_THREADS < `omp_set_num_threads()` < #pragma omp parallel num_threads(numthr)

1.4 *How does OpenMP behave when we declare a private variable?*

Each thread created will create its own (new) variable.

1.5 *What happens to the value of a private variable when the parallel region starts executing?*

The variable is not necessarily initialized when allocated in each thread stack, although when executing `omp2` in the lab computers, it is initialized as if it were a *firstprivate* variable (with the value of the variable previous to the parallel section). Furthermore, it is private to each thread so its value will only change because of instructions performed by that specific thread.

1.6 *What happens to the value of a private variable at the end of the parallel region?*

After the parallel region, the master threads resumes execution and the value of the variable is the one from before the parallel execution. Privates variable are local to the threads and the parallel region, so they are deleted when the threads terminate.

1.7 *Does the same happen with public variables?*

Public variables are shared by all threads (no copies of the variable are created for each thread); threfore, after the parallel region, the value will be determined by the code executed in the parallel region (and, perhaps, by the order of execution of the several threads).

2 Exercise 2: Parallelize the dot product

2.1 *Run the serial version and understand what the result should be for different vector sizes.*

As we are calculating the dot product of two vectors of size M where all the components are 1, the expected output is M.

2.2 *Run the parallelized code with the openmp pragma and answer the following questions in the document*

The parallel version is not working due to the fact that variable sum is being shared (default mode). So we have severals threads writing in the same variable at the same time which produces a data race.

2.3 *Modify the code and name the program pescalar_par2. This version should give the correct result using the appropriate pragma:*

The modification needed is adding the pragma clause just before the calculation. Both *pragmas* solve the data race. Nonetheless, the way they achieved so is different. *Critial* pragma implies the use of a

mutex so only one thread can access the block at a time (which is quite expensive in execution time) whereas *atomic* ensures that the whole calculation (line 42) is executed as a single operation. The result is exactly the same as we are enclosing just a sentence of code, however, the second option is considerably faster.

```
...
#pragma omp parallel for
for(k=0;k<M;k++)
{
    #pragma omp atomic
        sum = sum + A[k]*B[k];
}
...
```

2.4 *Modify the code and name the resulting program pescalar_par3. This version should give the correct result using the appropriate pragma:*

If we use *reduction* we get better results than the ones obtained with *atomic*. This is the appropriate solution in this case, because it is the optimized version meant for this specific task.

2.5 *Run time analysis*

We have written a script to test this (*scr2_threshold.sh*). As can be seen in the following figure (fig. 1), the parallel version (in the lab computers) is very inconsistent. However, the first value that meets the requirements is $T = 1200000$

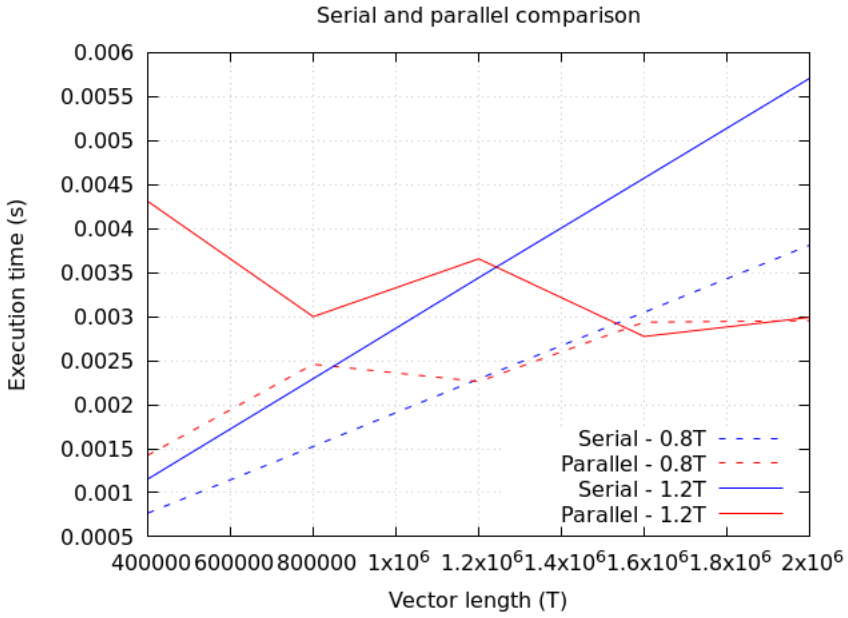

Figure 1: Search for threshold in the lab computers

3 Exercise 3: Parallel matrix multiplication

We have sampled times for the different programs using matrices of size 2000, which yield high enough times to make a difference between the different implementations. Here are the results:

threads	1	2	3	4
serial	106.778	101.143	101.286	99.7248
loop1	118.864	45.8971	274.153	282.774
loop2	106.161	51.9164	55.1282	53.3591
loop3	103.47	51.9108	51.613	51.1451

Figure 2: Times (seconds) for the requested tests

threads	1	2	3	4
serial	1	0.947231	0.948568	0.933947
loop1	1.11319	0.429838	2.56751	2.64825
loop2	0.994225	0.48621	0.516289	0.499722
loop3	0.969019	0.486157	0.483368	0.478987

Figure 3: Ratios for the requested tests

3.1 *Which of the three versions performs the worst? Why? Which of the three versions performs better? Why?*

The worst performing version is the one that parallelizes the innermost loop, because it is forced to "fork" itself N^2 times, which causes the execution to be a lot slower. The best performing one, as could be expected, is the one that parallelizes the outermost loop, it has to separate into different threads only once (the *multiplication_loop2.c* one "forks" N times).

3.2 *Based on the results, do you think fine-grained (innermost loop) or coarse-grained (outermost loop) parallelization is preferable in other algorithms?*

Based on this result, coarse-grained parallelization is generally preferable, as it avoids killing and creating threads, which slows the execution.