Parallelism

Computadors – Grau en Ciència i Enginyeria de Dades – 2021-2022 Q2

Facultat d'Informàtica de Barcelona

This Lab session aims at getting experience on parallelizing code and understanding its impact. In particular, we are going to play with the OpenMP approach, as well as analyse the behavior and the performance. In fact, the first part of the Lab Session introduces the basics of OpenMP code development. The second part aims at developing some codes that need to be parallelized.

We will perform some basic steps to get experience on developing and compiling codes parallelized with OpenMP. If you want to dive into details of OpenMP, here you may have a link to GNU Manual of the OpenMP Library (https://gcc.gnu.org/onlinedocs/libgomp) and the link to the official web site with specifications (http://www.openmp.org/specifications).

Introduction to OpenMP

Firstly, check the OpenMP version supported by your compiler with this command:

```
"#> gcc   -fopenmp  -E  -dM  - </dev/null   |   grep  -i  openmp"
```

For example, the compiler gcc 8.3.0 returns "#define _OPENMP 201511", as the OpenMP version of November (11) of 2015. "-E" means "preprocess only", and "-dM" means "dump preprocessor macros". By including "-fopenmp", we are sure to get the definition of _OPENMP. With "- </dev/null" we indicate the compiler that it should read an empty (/dev/null) program from its standard input.

In case OpenMP is not properly recognised, contact with your lecturer if you don't know how to solve this issue.

Exercise 1

*Develop a code called **"exercise1.c"** as the ones shown in the Theory slides. This code should show a given message (e.g. "I am the thread <ID>\n") as many times as the number of threads you have setup, including the ID of every thread. That is, a similar behaviour like the one shown in the Theory slides. Besides, we strongly suggest you declare a "char buf[256];" to print the messages, similar to the example of the Theory slides. Configure the number of threads **to use a total of 10 threads**. Validate you can correctly use the two approaches shown in the lectures, but use any of them in the final delivered source code file:*

1) update the environment variable ("OMP_NUM_THREADS") as shown in the Theory slides.

*2) use the directive "#pragma parallel **num_threads(n)**" directive, where "n" is the number of threads you want to use for this particular code section. We strongly suggest you can use an input parameter from the command line (i.e. "argv") to introduce the number of threads you want to use in this directive.*

***NOTE:** it is important to insert the braces (i.e. the "{" and "}" characters) as indicated in the slides. That is, the very next line after the directive and the very next line after the end of the parallelized code section.*

Let's compile the code with the command line:

```
“#> gcc  -o  exercise1  exercise1.c  -fopenmp”
```

The flag “-fopenmp” indicates to the compiler it has to enable handling OpenMP directives and, by default, it automatically links the OpenMP library (i.e. “libgomp”).

Exercise 2

*Copy the previous source code file and name it **“exercise2.c”**. Modify the code in order to compare the behaviour of the code depending on the declaration of “char buf[256];” is:*

*a) inside the parallelized piece of code (call it **“exercise2A.c”**)*
*b) outside (i.e. before) the parallelized code (call it **“exerciseB.c”**)*

*Execute multiple times the code (e.g. five times) and indicate in the **“answers.txt”** file whether you get the expected results (similar to the previous exercise) or incoherent ones (e.g. messages with repeated thread IDs). What is the reason you think the declaration of the variable has an impact on this behaviour? Argue your answer in the **“answers.txt”** file.*

Exercise 3

*Copy the source code file and name it **“exercise3.c”**. In this case, modify the code introduce a loop similar to the one shown in Theory slides:*

```
for (i=0; i<10; i++) {
   var = i;
   sleep(1);   //this function blocks the thread execution for 1 second
   printf("Msg: iterator %d; Var %d; by Thread %d\n", i, var, omp_get_thread_num());
}
```

and perform the following incremental variations in the code:

*a) in **“exercise3A.c”** introduce the loop, but with no “parallel” directive of OpenMP. You should see the loop executed by a single thread.*
*b) in **“exercise3B.c”** introduce the “parallel” directive, like the exercise 1.*
*c) in **“exercise3C.c”**, in addition to the previous “parallel” directive, introduce “#pragma omp for” directive on top of the for-loop (**NOTE:** remember you don't have to introduce any “{“ or “}” to delimit this directive).*

*Indicate in the **“answers.txt”** file the behaviour you have seen in the three executions.*

Now, keep doing the following incremental variations in the code:

*d) in **“exercise3D.c”**, add the “private(var)” clause in the “parallel” directive (i.e. “#omp pragma parallel private(var)”). This clause indicates there is a private copy of the “var” variable per thread.*
*e) in **“exercise3E.c”**, change the “private(a)” to “shared(a)” clause in the “parallel” directive (i.e. “#omp pragma parallel shared(var)”). This clause indicates there is a single and shared copy of the “var” variable for all threads in this code section. This is the default clause that parallel directives uses in case the developer don't introduce any variable specification.*
*f) in **“exercise3F.c”**, add the “#pragma omp critical” directive inside the for loop to protect the critical section (i.e. the code line the “var” is modified). NOTE: it is important to introduce the braces (i.e. “{“ and “}”) in the very next line after the directive and the very next line after the end of the critical section.*

Execute multiple times the code (e.g. five times) and check the output. You should see coherent messages in the codes of D and F sections.

Development of parallel codes

Use the following link to download the required files:

https://docencia.ac.upc.edu/FIB/GCED/COM/documents/Lab/S11/FilesS11.tar.gz

Exercise 5

Let's do a simple development modifying the attached ***"histogram.c"*** *file. This code reads from the stdin, calculates the number of times every character has been read (we assume the code reads ASCII characters) and print two outputs: stats to the stdout; the histogram to the stderr. We suggest you to execute the experiment with the following command line:*

```
./histogram   < input.txt    > stats.dat    2> histo.dat
```

In the code you will find some explanations of the purpose of every variable. You should develop parallelized code to keep track the total number of read characters, as well as the number of characters read per thread. At the end of the parallel code we strongly suggest to print the number of characters processed per thread to see the workload distribution among them. ***NOTE:*** *you can move the declaration of variables to the code line that you may consider.*

To validate your code, we have attached a text file, called "input.txt" with a total of 25659 characters. We additionally attach a file called "result.dat" with the histogram you should obtain.

Run multiple times the experiment with 1, 2, 4, 8, and 16 threads.

Understanding the time stats

Depending on the number of hardware threads and cores you have in the machine (i.e. virtual machine) it is likely you will not see any time improvement when you use more threads than hardware threads available in your cpu, because threads are colliding with each other to execute instructions.

Upload the Deliverable

To save the changes you can use the tar command as follows:

```
#tar czvf session11.tar.gz answers.txt Makefile *.c
```

Now go to RACO and upload this recently created file to the corresponding session slot.