Parallelism

Computadors – Grau en Ciència i Enginyeria de Dades – 2020-2021 Q2

Facultat d'Informàtica de Barcelona

As discussed in Theory lectures, parallel codes significantly improve the performance of sequential codes. Nevertheless, implementations may be quite different depending on the parallel programing models employed or even development decisions.

This Lab session aims at getting experience on parallelizing code. 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).

First part: introduction to OpenMP

Exercise 1

*Develop a code called "hello_world1.cpp" as the one shown in the theory slides. This code has to show as many "Hello World" messages as number of threads you have setup, including the ID of every thread in the hello message, as indicated as well in the slides. To configure the number of threads (**we want to use 10 threads**) check how to use the two approaches shown in the lectures: 1) environment variable ("OMP_NUM_THREADS"); and 2) "num_threads(n)" clause of the "#pragma parallel" directive.*

Let's compile the code with the command line:

```
#> g++  -o  hello_world1  hello_world1.cpp  -fopenmp
```

The last flag indicates the compiler it has to enable handling OpenMP directives and by default include the OpenMP library (i.e. "libgomp"). **NOTE:** in the Lab hosts with OpenSUSE you can use g++ or g++-5 compiler. In the Ubuntu virtual machines you can directly use the g++ compiler. If you develop with C language you can compile directly with gcc. Finally, if you are working on Windows, you can compile with the command line "#> cl /openmp hello_world1.cpp".

You can also check the OpenMP version supported by your compiler with this command:

```
#> g++  -fopenmp  -E  -dM  - </dev/null  |  grep  -i  openmp
```

For example, 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.

Exercise 2

Copy the previous source code file and name it "hello_world2.cpp". Modify the code to introduce the following loop:

```
for (i=0; i<10; i++) {
    //Start critical section
    a=i;
    sleep(1);   //this function blocks the thread execution for 1 second
    printf("Hello world %d vs %d by %d\n", i, a, omp_get_thread_num());
    //End critical section
}
```

In the first implementation put the loop inside the parallel code, but do not use any additional OpenMP directive to manage it. Then, perform several modifications to: a) introduce the "#pragma omp for" directive on top of the for; b) add the "private(a)" clause in the original first "parallel" directive; c) change the "private(a)" clause to "shared(a)" clause in the original first "parallel" directive; d) add the "#pragma omp critical" directive inside the for loop to protect the critical section.

Execute several times every single modification (a, b, c, and d), and write down in the "answers.txt" file your findings from the output, differences among repetitions of the same experiment and differences among the different experiments.

NOTE: in Windows the "sleep" function is written as "Sleep" and the parameter indicates miliseconds instead of seconds.

Exercise 3

Copy the first code file and name it "hello_world3.cpp". Modify the code to introduce the following lines in the parallel code:

```
//Start critical section
a = omp_get_thread_num();
sleep(1);
printf("Hello world %d \n", a);
//End critical section
printf("I am done %d\n", omp_get_thread_num());
```

Launch the code. Afterwards, introduce a "barrier" directive, as shown in the theory slides, just after the end of the critical section and before the final "printf". Run multiple times the experiments and write down in "answers.txt" your findings about the usage of the "barrier" directive.

Second part: development of introductory parallel codes

Exercise 4

Making Sums

Here we present different examples of OpenMP code that will run in a multi-threading setting. What will be the value of ans at the end of each program?

• *Code A.*

```
int main()
{
	int ans=0, x=0;
	#pragma omp parallel num_threads(4) shared(ans, x)
	{
		#pragma omp single
		x += 1;
		#pragma omp critical
		{
			ans += x;
		}
	}
	printf("The total is %d.\n", ans);
}
```

• *Code B.*

```
int main()
{
	int ans=0;
	#pragma omp parallel num_threads(2)
	{
		#pragma omp parallel num_threads(2)
		{
			#pragma omp parallel num_threads(2)
			{
				#pragma omp atomic
				ans += 1
			}
		}
	}
	printf("The total is %d.\n", ans);
}
```

• *Code C.*

```
int main()
{
	int ans=0;
	#pragma omp parallel num_threads(2)
	{
		int x=0;
		#pragma omp parallel num_threads(2) private(x)
		{
			#pragma omp atomic
			x += 1;
			#pragma omp atomic
			ans+=x;
		}
	}
	printf("The total is %d.\n", ans);
}
```

• *Code D.*

```
int main()
{
	int ans=0, x=0;
	#pragma omp parallel num_threads(3) shared(ans) private(x)
	{
		x += 2;
		#pragma omp single
		{
			ans += x;
		}
		#pragma omp atomic
		ans += 1;
	}
	printf("The total is %d.\n", ans);
}
```

Third part: development of parallel codes

Select, develop and analyse one of the following codes to better understand the advantages of parallel codes.

Exercise 5.a

Develop a code called "bubblesort.cpp". It accepts an input parameter that indicates the number of threads used in the parallel sections. The code comprises two main parts. The first one initializes an array of 1.000.000 integers. You can put random values or decremental values from the iterator of the loop. The second part sorts the array using the bubble sort approach. You can check the implementation given in Wikipedia, on the following link:
https://en.wikipedia.org/wiki/Bubble_sort#Pseudocode_implementation

Run the experiment with 1, 2, 4, 8, [16, and 32] threads. Depending on the number of cores you have in the machine going above 8 threads will not actually shown any improvements. This is why we only suggest to try 16 and 32. Executions with 16 and 32 should be safe, but not much useful. Measure the time of every experiment (e.g. "#>/usr/bin/time ./program") and write down the measurements and findings in the "answers.txt" file. ***NOTE:*** *there are several ways to protect the shared variables. We suggest you to run experiments with different approaches to measure the impact on execution time.*

Exercise 5.b

*Develop a program called "matrixmul.cpp" that implements a matrix multiplication and prints the result. That is, C = A * B, where each of them is a matrix of [size][size] and randomly initialized. The type of data is "float". HINT: the algorithm of the multiplication should be something like:*

```
for (i = 0; i < SIZE; i++) {
	for (j = 0; j < SIZE; j++) {
		c[i][j] = 0.0;
		for (k = 0; k < SIZE; k++)
			c[i][j] = c[i][j] + a[i][k] * b[k][j];
	}
}
```

Run the experiment with 1, 2, 4, 8, 16, and 32 threads. Measure the time of every experiment and write down the measurements and findings in the "answers.txt" file. ***NOTE:*** *there are several ways to protect the shared variables. We suggest you to run experiments with different approaches to measure the impact on execution time.*

Upload the Deliverable

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

```
#tar czvf session10.tar.gz answers.txt Makefile *.cpp
```

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