Universitat Pompeu Fabra Barcelona

Click to show/hide PDF Layer

Lab 1

Pablo Arias*, Ricard Borrell Pol†, Sergi Laut Turón‡, Pavel Pratyush§, Daniel Santos-Oliván#

Instructions

The first Lab session consists of 3 problems with several questions each. You must compile your answers to the exercises in a report, explaining how you solved the problem and answering the questions. This lab assignment is worth 15% of the total grade of the subject, and the deadline for it is April 25th at 23:59h.

Each group must submit a compressed file named lab1_TxGyy.zip, where TxGyy is your group identifier. A .tar or a .tgz file is also accepted (e.g., lab1_T1G1.zip or lab1_T2G21.zip). The compressed file **must** contain three folders (1_pi, 2_sort, and 3_primes) and **all** the requested files with the following structure. Additional files will not be considered and will not be penalized.

```
lab1_TxGyy.zip ─┬─ 1_pi ─┬─ pi_seq.c
                │        ├─ pi_par.c
                │        └─ pi_task.c
                ├─ 2_sort ─┬─ main.c
                │          ├─ sort.c
                │          └─ sort.h
                └─ 3_primes ── primes.c
```

A sample file named lab1_TxGyy.zip containing the reference codes has been published in Aula Global. You need to create your job scripts to perform your tests according to what is asked. The Makefiles should not be modified unless the exercise says so. Focus on the code and the work requested for each exercise.

If you have any questions, please post them on the lab class forum in Aula Global. However, do not post your code in the Forum. If you have other questions regarding the assignment that you consider cannot be posted in the Forum (e.g., personal matters or code), please contact the lab responsible person.

Criteria

The codes will be tested and evaluated on the same cluster where you work. The maximum grade on each part will only be given to these exercises that solve in the most specific way and tackle all the functionalities and work requested. All the following criteria will be applied while reviewing your labs in the cluster.

* pablo.arias@upf.edu.
† ricard.borrell@upf.edu.
‡ sergi.laut@upf.edu.
§ pratyush@upf.edu.
# daniel.santos@upf.edu.

Exercises that will not be evaluated:

- A code that does not compile.
- A code giving wrong results.
- A code that does not adhere to all the input/output requests.
- lab1_TxGyy.zip delivered files not structured or named as described previously.

Exercises with penalty:

- A code with warnings in the compilation.

1. Numerical integration

The following integral computes the value of Pi:

$$\int_0^1 \frac{4}{1+x^2}\,dx = \pi. \tag{1}$$

This can be approximated as the sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi \tag{2}$$

Where each rectangle has a width Δx and height $F(x_i)$ at the middle of interval i.

Write a code in C that implements this computation. Name this code $pi\_seq.c$. The executable must accept N as argument. The only output must be formatted as follows:

```
printf("\nPi with %i steps is %.15lf in %lf seconds\n", N, pi, runtime);
```

where N is the number of rectangles, pi is the computation result, and $runtime$ is the computation elapsed wall time.

Note: The elapsed time must only include the computation of pi. Do not include any other parts of the code.

Report Questions 1 — Pi (50%)

1. Test different values of N and check its effect on the results. Compare them with the real value (up to 11 significant digits). Explain what you observe.
2. Parallelize the code using **ONLY** the $\#pragma\ omp\ parallel$ directive. Name this code $pi\_par.c$. You can modify or add anything you need to the code. Make sure the results are the same for different numbers of OpenMP threads. Explain how you have parallelized it.
3. Parallelize the code using $tasks$. Name this code $pi\_task.c$. Use a divide-and-conquer strategy. That is, create tasks that split the domain recursively. Explain how you have done it. To avoid stack overflow due to excessive task creation, you will have to modify the code so that the tasks compute at least 1024 steps. Therefore, this code must accept another argument, M, which is the minimum number of steps in a task.
4. Compare the three versions times with $N = 1024^3 = 1073741824$. For the case of $pi\_task.c$, use the $M = 1024^2$. Explain the results.

2. Sorting

The sorting problem is a well-studied problem in algorithmics. In this exercise, you will have to parallelize using OpenMP the "insertion sorting algorithm" implemented with this definition:

```
int sort_openmp(int *array, int n)
```

A sequential insertion sort is already implemented in the function `sort()` (this implementation does not need to be modified). Use this implementation as a reference to implement `sort_openmp()`. In order to do so, consider the following:

- The number of threads is defined in the macro `_NUM_THREADS`. With this, we specify the number of running threads with the clause `num_threads(_NUM_THREADS)`. Note the value of the macro is set at compile time using the Makefile. Remember to request enough CPUs (1 thread/CPU) with your jobs.
- A new array named `tmp_array` will be created, and each thread will only read from the positions of its chunk from the array and save the sorted values in `tmp_array`. For example, with 2 threads and an array of 100 positions, thread 0 will sort the elements of `array[0-49]` and save them in `tmp_array[0-49]` and thread 1 will sort elements of `array[50-99]` and save them in `tmp_array[50-99]`.
- We will not use the OpenMP `for` clause; instead, we will use `#pragma omp parallel`, the thread `ID`, and the chunk size (calculated by us) to divide the for loop among the threads in a static scheduler-like fashion.
- Once the sorting phase has finished and we have saved the values in `tmp_array`, we will start a sequential phase where we will go through `tmp_array` to save the final sorted version in `array`. We will create a data structure named `head_i[_NUM_THREADS]` where we will save the first position of each thread. To illustrate the process here, there is a step-by-step example:

Assume a case for 2 threads and an array of size 6. Each thread has its part sorted in `tmp_array`, and we want to save the final result in an array. At iteration 0, the content of `head_i` is the index of the beginning of each part of each thread.

i	0					
head_i	0	3				
array	4	7	1	9	0	3
tmp_array	1	4	7	0	3	9

We compare elements at positions 0 and 3. Because the value 0 is smaller than value 1, we increment the index where 0 was. Now `head_i[1]` = 4 and we have updated the element `array[0]`.

i	0					
head_i	0	4				
array	0	7	1	9	0	3
tmp_array	1	4	7	0	3	9

Next, we follow the indices of `head_i` and compare 1 and 3. Because 1 is smaller, we save it in `array` and update the indices.

i	1					
head_i	1	4				
array	0	1	1	9	0	3
tmp_array	1	4	7	0	3	9

The process continues with the same logic.

i	2					
head_i	1	5				
array	0	1	3	9	0	3
tmp_array	1	4	7	0	3	9

i	3					
head_i	2	5				
array	0	1	3	4	0	3
tmp_array	1	4	7	0	3	9

i	4					
head_i	3	5				
array	0	1	3	4	7	3
tmp_array	1	4	7	0	3	9

At this point, we must take into account that once the `head_i` pointer has passed the last element of its part, do not start using elements of the values corresponding to the next thread. For example, if we compare elements `tmp_array[3]` and `tmp_array[5]`, we will compare values 0 and 9 from thread 1. Just create a condition

that detects when an index has reached the end. The last step will then be completed, and the array will contain the final sorted set of values.

i	5					
head_i	3	6				
array	0	1	3	4	7	9
tmp_array	1	4	7	0	3	9

Note: Make sure the resulting array is properly sorted. Codes with wrong results will not be evaluated.

Report Questions 2 — Sort 30%

1. Explain the strategies you have followed in order to parallelize the algorithms. Which dependencies between iterations did you find?
2. Plot the speedup for an array of 200.000 elements and 1, 2, 4, 8, and 16 cores for a strong scaling test. Plot the ideal speedup in the figures and use a logarithmic scale to print the results. Discuss the results and the gain.

3. Prime numbers

In this exercise, we will work with the prime number code we used in Seminar 1. We provide a variation implemented using an outer `for` loop. For each integer I, it simply checks whether any smaller J evenly divides it. The total amount of work for a given N is thus roughly proportional to $1/2 * N^2$.

In the code, there are three functions:

```
int prime_default ( int n );
int prime_static ( int n );
int prime_dynamic ( int n );
```

Parallelize the first with default scheduling.

Parallelize the second using static scheduling. Test different chunk sizes.

Parallelize the third using dynamic scheduling. Test different chunk sizes.

You do not need to modify the code. You only have to add directives.

Report Questions 3 — Primes 20%

1 Run the code with 1, 2, and 4 threads. Which scheduling technique scales the best?

2 Why? Discuss the results.