Lab 3
Thread-level parallelism: Parallelization via OpenMP and asynchronous scheduling of digital photo processing

Objectives:

- Learn how to parallelize an application on a centralized memory parallel machine through *threads using* the *shared variable* style.
- Study the OpenMP API and apply different parallelism strategies in its application.
- Study the C++ API for asynchronous programming to exploit functional parallelism.
- Apply methods and techniques specific to this subject to estimate the maximum gains and efficiency of the parallelization process.
- Apply all of the above to a problem of sufficient complexity and size.

Development:
In this practice, you will have to parallelize, using OpenMP and asynchronous programming, the solution to a given problem to take advantage of the different cores available to each practice computer. You will therefore parallelize for a multiprocessor system (centralized memory parallel machine), in which all cores in the same package see the same memory, i.e. a pointer on one core is the same pointer for the rest of the microprocessor cores.

Task 0.1 OpenMP pre-training:
Look at the following C program where two vectors of *floats* are added using OpenMP to parallelize the calculation.

```
#include <omp.h>
#define N 1000
#define CHUNKSIZE 100

main(int argc, char *argv[]) {

    int i, chunk;
    float a[N], b[N], c[N];

    /* We initialize the vectors */
    for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;
    chunk = CHUNKSIZE;

    #pragma omp parallel shared(a,b,c,chunk) private(i)
    {
  #pragma omp for schedule(dynamic,chunk) nowait
  for (i=0; i < N; i++)
  c[i] = a[i] + b[i];
    } /* end of parallel region */
}
```

Check the OpenMP documentation (API, tutorials, this link: https://computing.llnl.gov/tutorials/openMP/, etc...) and respond:

0.1.1 What is the **`chunk`** variable used for?

0.1.2 Fully explains the *pragma* :

```
#pragma omp parallel shared(a,b,c,chunk) private(i)
```

Why and what is **`shared(a,b,c,chunk)`** used for in this program?

Why is the variable `i` labeled as **`private`** in the pragma?

0.1.3 What is the **`schedule for`**? What other possibilities are there?

0.1.4 What times and other performance metrics can we measure in parallelized code sections with OpenMP?

Task 0.2: Pre-training std::async

Look at the following c++ program where two functions are called with std::async:

```
#include <iostream>
#include <future>
#include <chrono>

int task(int id, int millis) {
    std::this_thread::sleep_for(std::chrono::milliseconds(millis));
    std::cout<<"Task"<<id<<" completed"<<std::endl;
    return id;
}

int main() {
    auto start = std::chrono::high_resolution_clock::now();
    std::future<int> task1 = std::async(std::launch::async, task, 1, 2000);
    std::future<int> task2 = std::async(std::launch::async, task, 2, 3000);

    task1.wait();
    int taskId = task2.get();

    auto end = std::chrono::high_resolution_clock::now();
    auto elapsed = std::chrono::duration_cast<std::chrono::milliseconds>(end-start);

    std::cout<<"Completed in:"<<elapsed.count()<<"ms"<<std::endl;
}
```

To compile it use g++ and link to the pthreads library with `-lpthread`.

Review the std::async documentation and answer the following questions:

0.2.1 What is the `std::launch::async` parameter used for?

0.2.2 Calculate the time the program takes with `std::launch::async` and `std::launch::deferred`. What is the reason for the time difference?

0.2.3 What is the difference between the `wait` and `get` methods of `std::future`?

0.2.4 What advantages does `std::async` offer over `std::thread`?

Task 0.3: Pre-training std::vector

Look at the following c++ program where an stl vector is initialized and filled with values:

```
#include <vector>
#include <iostream>
#include <chrono>

int main() {
        // default initialization and add elements with push_back
        auto start = std::chrono::high_resolution_clock::now();

        std::vector<float> v1;
        for (int i = 0; i < 10000; i++)
                v1.push_back(i);

        auto end = std::chrono::high_resolution_clock::now();
```

```
                auto elapsed = std::chrono::duration_cast<std::chrono::milliseconds>(end-
                start);
                std::cout<<"Default initialization:"<<elapsed.count()<<"ms"<<std::endl;

                // initialized with required size and add elements with direct access
                start = std::chrono::high_resolution_clock::now();

                std::vector<float> v2(10000);
                for (int i = 0; i < 10000; i++)
                        v1[i] = i;
                end = std::chrono::high_resolution_clock::now();

                elapsed = std::chrono::duration_cast<std::chrono::milliseconds>(end-start);
                std::cout<<"Initialization with size:"<<elapsed.count()<<"ms"<<std::endl;
        }
```

Answer the following questions:

0.3.1: Which of the two ways of initializing the vector and filling it is more efficient? Why?

0.3.2: Could a problem occur when parallelizing the two for loops? Why?

Task 1: Study of the OpenMP API [Compulsory individual part (25% of the grade)].

The OpenMP API and its use with GNU GCC should be studied, checking the correct operation of some of the examples available on the Internet (e.g. https://lsi.ugr.es/jmantas/ppr/ayuda/omp_ayuda.php).

Each member of the group* will have to make a small tutorial as a "OpenMP parallelism recipe book" with **different examples of the application** of OpenMP parallelism in different **software structures**.

Task 2: Parallelization of digital photo development

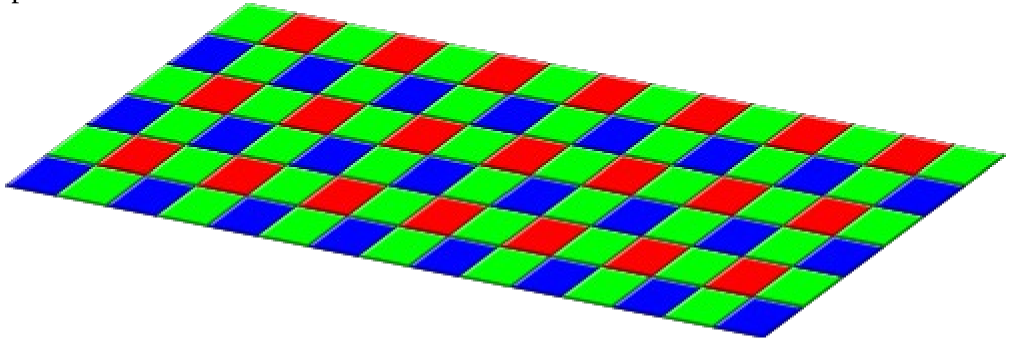
All the groups of each turn of practices must undertake the parallelization of the problem posed. To do so, the sequential solution provided will be analyzed following the indications of the statement and the teachers. Subsequently, it will be transformed, using OpenMP and asynchronous programming, to incorporate parallelism at thread level.

Problem:

The problem posed is the digital development of photographs.

In traditional analog photography, the camera captures the image through the lens and projects it onto a photosensitive film. This film is called a "negative" because an image of inverted intensity is captured. Once the negatives are captured, a chemical process is used to develop the captured image on photographic paper, which makes the paper react to the exposure to the light projected through the negative.

In digital photography, the process of capturing and developing photographs is similar conceptually but very different in practice. In the case of a digital camera, the light coming through the lens is projected onto a digital sensor known as a CCD. This digital sensor is an array of pixels that produce an electrical intensity directly proportional to the intensity of the light received. The electrical intensity of each of these pixels is read by the camera driver to generate a digital image that is stored on the memory card. This process, which in principle seems simple, becomes more complicated when you want to obtain a color image, since the pixels of the sensor only react to the light intensity and not to the wavelength (color). To solve this problem, the so-called Bayer filter is used, which consists of placing filters of the three primary colors, red, green and blue, alternately on each of the pixels of the sensor:



As can be seen, there is a higher proportion of green pixels than red and blue pixels. This is because the human eye is more sensitive to the color green. Not all cameras use the same pattern, not even the same primary colors, but the most common is the one shown in the image above and is known as RGGB or Bayer pattern.

Therefore, the image that comes out of the camera sensor is a two-dimensional matrix, of the same size in pixels as the sensor, with the RGB color intensities interleaved following the Bayer pattern. This image is known as a RAW (raw) image and is the equivalent of the negative in analog photography. This RAW image contains the intensities recorded by each of the pixels of the sensor without any additional processing. To obtain the final image, a series of processes are applied to this RAW image.

Usually, when we take a picture with a digital camera, what we receive on the memory card is the image already processed in JPEG or TIFF format. Most mid-high end cameras (even some cell phones or GoPro) provide the possibility to save both the JPEG processed image and the RAW image. Working with the RAW image provides a number of advantages as it gives more freedom to the photographer to edit, being able to obtain a higher quality end result. The JPEG image can also be edited with photo editors such as Photoshop or GIMP, however, this already processed image is compressed and in a much lower dynamic range than the RAW image, 8 bits per pixel (256 gray levels) of the JPEG versus up to 16 bits per pixel (65536 gray levels) of the RAW image. This allows more aggressive filters to be applied without losing image quality and reducing the occurrence of artifacts such as banding.

The processes applied to the RAW image depend on the camera driver or digital development software used (Lightroom, Darktable, RawTherapee, ...) but generally include the following steps:

- **Debayer or demosaicing**: this is the process by which red, green and blue pixels are separated into individual channels and the missing pixels are interpolated to obtain the RGB image.
- **White balance**: is the process by which the color temperature is adjusted so that white colors appear white and not reddish or bluish.
- **Gamma correction**: the image obtained from the sensor is given by the sensitivity of the pixels that react linearly to light intensity. However, our eyes do not react in this way, they are less sensitive to darker or brighter areas so that the linear image output from the sensor is perceived by the human eye as a flat image without contrast. This process therefore converts the linear image to a non-linear image more similar to how the human eye perceives light.

- **Sharpening**: the resulting image from the sensor usually lacks sharpness since the intensity of a pixel is usually affected by the intensity received by neighboring pixels due to the Bayer pattern. Therefore, a process that increases the sharpness of the final image is usually applied.

Task 2.1: Analyze the code and identify the different processes
In this task you will have to analyze the code using the strategies seen in the previous practice to identify the different processes that are applied on the RAW image and elaborate a dependency diagram.
Identifies if some processes can be executed in parallel and the dependencies between them.

Task 2.2: Analyze the code of each process
In this task you must analyze the code corresponding to each process identified in task 3.1. Make a flow diagram of each one and identify possible parallelizable points.

Task 2.3 Parallelization
Having identified the parallelizable parts of the code, it is now time to apply parallelism with OpenMP and asynchronous programming. Run tests parallelizing different parts and observe the performance gain.
Answer the following questions:

- Is better performance obtained by parallelizing as many parts as possible?
- Is performance degraded by parallelizing certain parts?
- If so, to what do you think this degradation is due?

Objectives:

- Obtain a parallel version with OpenMP and std::async of the provided sequential solution
- Analyze and exploit in all cases the different possible types of parallelism perceived in the given sequential solution.
- Perform sufficient tests to estimate performance values of the parallel version versus the sequential version.

Steps:

a) Analyze the provided sequential solution to ensure understanding of the problem. Pay attention to details seen in previous practices that are important for designing the parallel solution: variables, problem size, etc.
b) Make a graphical representation in the form of a dependency diagram of the operation of the given sequential solution.
c) Using OpenMP and std::async and in view of the above details, design a thread-parallel solution starting from the sequential version
d) Test your parallel version for different cases and configurations by taking timing and gain measurements.
e) Make a graphical representation in the form of a control-flow graph of the operation of the parallel solution based on OpenMP and std::async.
f) Comment on the following aspects of both the sequential and parallel solution as indicated:

About the implemented code

- Discuss the most interesting code portions of both the sequential and parallel solutions implemented:
    - Aspects that define the size of the problem.
    - Code control structures of special interest in the solution to the problem.
    - It is important to justify in as much **detail as** possible the changes that have been made with respect to the sequential version in order to parallelize the solution.
    - OpenMP or std::async instructions and blocks used for code parallelization.

On the exploited parallelism

- **Reviewing the documentation for "*Unit 3. Parallel Computing*".** State the following with respect to your practice:
    - Types of parallelism used.
    - Parallel programming mode.
    - What communication alternatives (explicit or implicit) your program employs.
    - Parallel programming style used.
    - Type of parallel program structure

On the results of the tests performed and their context

- Characterization of the parallel machine on which the program runs (e.g. number of compute nodes, cache system, memory type, etc.).
  On Linux use the command: `cat /proc/cpuinfo` to access this information or `lscpu`.
- What does the word `ht` in the output of the above command invocation mean? Does it appear on your practice computer?
- Calculate the following (with associated graphs and brief explanation):
    - Gain in speed (*speed-up*) as a function of the number of computation units (*threads* in this case) and as a function of the parameters that modify the **size of the problem** (e.g. dimensions of a matrix, number of iterations considered, etc. ....).
    - Comment on the results in a reasoned manner. What is the theoretical maximum speed gain?
      Note: 2D or 3D graphics can be delivered as illustrative.

- **Answer with justification:** Which is the more efficient implementation of the 2?

Task 3: Write a report analyzing the design using OpenMP and std::async and the results obtained.

A report will be written in which different aspects of both the sequential version and the proposed parallel implementation will be discussed and explained. The results obtained by testing them will also be analyzed. The report should give clear answers to the questions raised in each task. Each teacher will indicate how the individual parts of each member of the groups will be handed in.

General notes to the practice:

- The answers and documentation generated in both Part I and Part II tasks will be integrated in the joint structured report, including the individual sections as an annex, which will be handed in at the end of practicum 3.
- The GitHub development platform will be used for the work in part II and in the memory. This will allow the management of a **common work repository to be** created by each group including its members and the teacher. The teacher will make his or her user known on this platform in order to be added.
- The implementation must be able to run under the Linux operating system of the practice laboratory.
- The attached code uses the OpenCV library to run the image processing. This library is already installed in the PCs of the labs, but to make it work in your PCs, it is necessary to compile and install OpenCV. To do so, please follow the instructions in this link. Another option is to use the EPS virtual machine.
- To compile, the enclosed code includes a CMakeLists.txt file that contains the external library dependencies information and how to build the sources. This file is used to build a Makefile using CMake. To do this, create a folder named build in the same directory and run the command `cmake -S . -B ./build`. Once the command is executed, inside the build folder there will be a Makefile file with which you can run the make command to build the executable. Inside the build folder, run the command make -j4 to compile the executable.
- To run the development process use the command `./unraw <raw_file> <output_file>`, e.g. `./unraw ../_DSC2078.NEF output.jpg`.
- Information about OpenMP can be found at www.openmp.org. To compile use the make generated by cmake which already includes the -fopenmp flag.
- The practicum will be due by the method of your practicum professor's choice prior to the practicum session the week of December 11, 2023.

***Note:**

The theoretical/practical work must be original. The detection of copying or plagiarism will result in a grade of "0" in the corresponding test. The direction of the Department and the EPS will be informed about this incidence. Repeated misconduct in this or any other subject will lead to the notification of the corresponding vicerectorate of the faults committed so that they can study the case and sanction according to the legislation.