Practice 3
Thread-level parallelism: Parallelization through OpenMP and asynchronous scheduling

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 **share the same memory**, i.e. a pointer on one core is the same pointer for the rest of the microprocessor cores.

Task 1.1 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:

1.1.1 What is the purpose of the `chunk` variable?
1.1.2 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?

1.1.3 What is the use of the `schedule` and what other possibilities are there?
1.1.4 What times and other performance metrics can we measure in parallelized code sections with OpenMP?
NOTE: to compile with openMP add `-fopenmp` to the compile options.

Task 1.2: Pre-training std::async

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

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

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;
}
```

Review the std::async documentation and answer the following questions:
1.2.1 What is the `std::launch::async` parameter used for?
1.2.2 Calculate the time the program takes with `std::launch::async` and `std::launch::deferred`. What is the reason for the time difference?
1.2.3 What is the difference between the `wait` and `get` methods of `std::future`?
1.2.4 What advantages does `std::async` offer over `std::thread`?

Task 2: 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 parallelism offered by OpenMP in function of different **software structures**.

Task 3: Use of reduction and sections
Given the following C++ code:

```
#include <iostream>
#include <vector>
#include <random>
#include <omp.h>
#include <chrono>
#include <future>

std::random_device os_seed;
const int seed = 1;

std::mt19937 generator(seed);
std::uniform_int_distribution<> distribute(0, 1000);

double average(const std::vector<double> &v)
{
  double sum = 0.0f;
  for(int i=0;i<v.size();i++)
    sum += v[i];
  return sum/v.size();
}

double maximum(const std::vector<double> &v)
{
  double max = 0.0f;
  for(int i=0;i<v.size();i++)
    if (v[i]>max) max = v[i];
  return max;
}

double minimum(const std::vector<double> &v)
{
  double min = 1.0f;
  for(int i=0;i<v.size();i++)
    if (v[i]<min) min = v[i];
  return min;
}

int main(){

  int size = 100000000;
  std::vector<double> v(size);
  for(int i=0;i<v.size();i++)
    v[i] = distribute(generator)/1000.0;

  auto start=std::chrono::steady_clock::now();
  double min, max, avg;

  min = minimum(v);
  max = maximum(v);
  avg = average(v);

  auto end=std::chrono::steady_clock::now();
  std::chrono::duration<double> elapsed_seconds=end - start;

  std::cout<<"Elapsed:"<<elapsed_seconds.count()<<std::endl;
  std::cout<<"Min:"<<min<<" Max:"<<max<<" AVG:"<<avg<<std::endl;
}
```

Ignoring the vector initialization time and always starting from the initial code, answer the following questions:

3.1: How long does the sequential program take to run?
3.2: Use the `reduction` clause of openMP to parallelize the calculations of the `minimum`, `maximum` and `average` functions. How long does the program take to run?
3.3: Use the sections clause to make each thread execute a function. How long does it take to execute the program?
3.4: Now use `std::async` to launch each function in a task instead of sections. How long does it take to execute the program? NOTE: be sure to use `std::ref` to pass the vector by reference instead of by value to avoid a copy on each call.
3.5: Combine `reduction` with `sections`. How long does it take to execute the program, is improvement observed, why? NOTE: look at this function omp_set_max_active_levels
3.6: Combine openMP `reduction` with `std::async`. How long does it now take to execute, and is there any improvement? Why?
3.7: Calculate the gain of 3.2, 3.3, 3.4, 3.5 and 3.6 with respect to 3.1. Which version obtains the best gain and why?

Task 4: Running conditions

A race condition occurs when several threads read and/or write to the same variable without taking into account the value modified by another thread. For example:

```
#include <iostream>
#include <omp.h>
int main() {
    int max = 0;
    int min = 1000;
    #pragma omp parallel for
    for (int i=1000;i>=0;i--) {
        if (i > max) max = i;
        if (i < min) min = i;
    }
    std::cout<<"Max:"<<max<<" Min:"<<min<<std::endl;
}
```

We can see that there is a dependency between the reading of the max variable and its writing. Each thread must evaluate the current value of `max` and update it if necessary (the same happens with `min`). Several threads can read the current value of `max` at the same time and update it at the same time so that the final value will be that of the last thread instead of the largest.

4.1: Test the compiled code with openMP, what is the result obtained, is it correct? NOTE: use `watch -n1 ./program` to run the program every second and check that the result is always the same.
4.2: Add the `#pragma omp critical` clause on each if. What is the result obtained? Is it correct now?

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

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

int main() {
  int size = 10000000;
  // 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 < size; 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 of "<<v1.size()<<"
elements:"<<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(size);
  for (int i = 0; i < size; i++)
    v2[i] = i;

  end = std::chrono::high_resolution_clock::now();
  elapsed = std::chrono::duration_cast<std::chrono::milliseconds>(end-start);
  std::cout<<"Initialization with size of"<<v2.size()<<"
elements:"<<elapsed.count()<<"ms"<<std::endl;
}
```

Answer the following questions:

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

4.4: Could a problem occur when parallelizing any of the for loops, and why?

4.5: Parallelize with OpenMP both loops without modifying the code (only adding pragma omp parallel for and/or pragma omp critical if necessary). What is the gain obtained? Explain the results obtained.

<u>Task 5:</u> Use the sequential implementations from Task 2.1.3 of Practice 2 and parallelize with OpenMP on a shared-memory parallel machine:

- Calculation of PI as the definite integral between 0 and 1 of the derivative of the arctangent. See the link: https://lsi2.ugr.es/jmantas/ppr/tutoriales/tutorial_mpi.php?tuto=03_pi
- Calculation of PI using the Monte Carlo Method. See the link https://www.geogebra.org/m/cF7RwK3H

Calculate the following times for each program:

- Sequential time of the program without parallelization.
- Parallel time of the parallelized program executed with different number of threads T=1,2,3,4,5,6 ...Plot these times and comment on the results. Is speed gain observed? Why?
- Compare the times obtained with the MPI version, which is faster and why?

General notes to the practice:

- The implementation will have to be able to run under the Linux operating system of the laboratory, and will take 4 sessions (8h)
- It is mandatory to deliver a Makefile with the appropriate rules to compile and clean the different programs implemented (make clean) in a simple way.
- The students will deliver, in addition to the application developed, a report, structured according to the professor's indications, with the information obtained.
- Delivery: to be delivered after 4 practice sessions.

- 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 Department and EPS management will be informed about this incidence. Repeated misconduct in this or any other subject will lead to the notification of the corresponding vice-rectorate of the faults committed so that they can study the case and sanction according to the legislation.