TP3 - OpenMP (Parallel Sections, Single, Master, Synchronization)

Imad Kissami

June, 2025

Exercise 1 : Work Distribution with Parallel Sections

- Write a program that initializes an array of size `N` with random values.
- Use `#pragma omp sections` to divide the work :
  - Section 1 : Compute the sum of all elements.
  - Section 2 : Compute the maximum value.
  - Section 3 : Compute the standard deviation.
- Ensure that all computations run in parallel.

Exercise 2 : Ordered Execution with Single

- Implement an OpenMP program where multiple threads generate numbers in parallel.
- Only one thread at a time should print a number using `single`.
- The numbers must be printed in ascending order.

Example Output (using 4 threads) :

```
Thread 3 generated value: 12
Thread 1 generated value: 27
Thread 2 generated value: 34
Thread 0 generated value: 89
```

Exercise 3 : Exclusive Execution - Master vs Single

- Write a program where :
  - A master thread initializes a matrix.
  - A single thread prints the matrix.
  - All threads compute the sum of all elements in parallel.
- Compare execution time with and without OpenMP.

Exercise 4 : Barrier Synchronization

- Implement a program where multiple threads execute different stages of computation :
  - Stage 1 : Read input data (only one thread should do this).
  - Stage 2 : All threads process the data in parallel.
  - Stage 3 : A single thread writes the final result.
- Use `barrier` to enforce correct execution order.

Exercise 5 : Load Balancing with Parallel Sections

- Implement a task scheduling mechanism using parallel sections.
- Simulate three different workloads :
  - Task A (light computation)
  - Task B (moderate computation)
  - Task C (heavy computation)
- Measure the execution time and optimize the workload distribution.

Exercise 6 : Critical vs Atomic for Shared Counters

- Implement a counter that multiple threads increment simultaneously.
- Test the program using both :
  - `critical` section
  - `atomic` directive
- Compare performance and explain the differences.

Exercise 7 : Producer-Consumer Problem

- A producer generates values and places them in a shared buffer.
- A consumer retrieves these values and processes them.
- The two entities must be synchronized to avoid race conditions.

Sample code :

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

#define N 1000000

void fill_rand(int n, double *A) {
  for (int i = 0; i < n; i++)
    A[i] = rand() % 100;
}

double Sum_array(int n, double *A) {
  double sum = 0.0;
  for (int i = 0; i < n; i++)
  sum += A[i];
  return sum;
}

int main() {
  double *A, sum, runtime;
  int flag = 0; // Synchronization flag

```

```c
    A = (double *)malloc(N * sizeof(double));

    runtime = omp_get_wtime();

    fill_rand(N, A); // Producer fills the array

    sum = Sum_array(N, A); // Consumer computes the sum

    runtime = omp_get_wtime() - runtime;

    printf("In %lf seconds, the sum is %lf\n", runtime, sum);

    free(A);
    return 0;
}
```