TP4 - OpenMP (Parallel Sections, Single, Master, Synchronization)

Imad Kissami

February, 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.
- Do not recompute the sum inside Section 3. The standard deviation must use the result obtained in Section 1.

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

#define N 1000000

int main() {

   double *A = malloc(N * sizeof(double));
   if (A == NULL) {
      printf("Memory allocation failed\n");
      return 1;
   }

   double sum = 0.0;
   double mean = 0.0;
   double stddev = 0.0;
   double max;

   // Initialization
   srand(0);
   for (int i = 0; i < N; i++)
      A[i] = (double)rand() / RAND_MAX;

   // Compute sum and max
   sum = 0.0;
   max = A[0];
   for (int i = 0; i < N; i++) {
      sum += A[i];
      if (A[i] > max)
         max = A[i];
```

```c
    }

    // Compute mean
    mean = sum / N;

    // Compute standard deviation
    stddev = 0.0;
    for (int i = 0; i < N; i++)
        stddev += (A[i] - mean) * (A[i] - mean);

    stddev = sqrt(stddev / N);

    // Print results
    printf("Sum     = %f\n", sum);
    printf("Max     = %f\n", max);
    printf("Std Dev = %f\n", stddev);

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

Exercise 2 : 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.

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

#define N 1000

void init_matrix(int n, double *A) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            A[i*n + j] = (double)(i + j);
        }
    }
}

void print_matrix(int n, double *A) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            printf("%6.1f ", A[i*n + j]);
        }
        printf("\n");
    }
}

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

int main() {

    double *A;
    double sum;
    double start, end;

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

```c
    if (A == NULL) {
        printf("Memory␣allocation␣failed\n");
        return 1;
    }

    start = (double) clock() / CLOCKS_PER_SEC;

    /* Initialization */
    init_matrix(N, A);

    /* Printing (can be commented if N is large) */
    /* print_matrix(N, A); */

    /* Sum computation */
    sum = sum_matrix(N, A);

    end = (double) clock() / CLOCKS_PER_SEC;

    printf("Sum␣=␣%lf\n", sum);
    printf("Execution␣time␣=␣%lf␣seconds\n", end - start);

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

Exercise 3 : 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.

```c
#include <math.h>

void task_light(int N) {
    double x = 0.0;
    for (int i = 0; i < N; i++) {
        x += sin(i * 0.001);
    }
}

void task_moderate(int N) {
    double x = 0.0;
    for (int i = 0; i < 5*N; i++) {
        x += sqrt(i * 0.5) * cos(i * 0.001);
    }
}

void task_heavy(int N) {
    double x = 0.0;
    for (int i = 0; i < 20*N; i++) {
        x += sqrt(i * 0.5) * cos(i * 0.001) * sin(i * 0.0001);
    }
}
```

Exercise 4 : Synchronization and Barrier Cost

- Implement a dense matrix-vector multiplication.
- Version 1 : Implicit barrier.
- Version 2 : Use `schedule(dynamic)` with `nowait`.

- Version 3 : Use `schedule(static)` with `nowait`.
- Measure :
  - CPU time
  - Speedup
  - Efficiency
- Explain :
  - Why barriers limit scalability.
  - When `nowait` becomes dangerous.
- Run the code with 1, 2, 4, 8, 16 threads and compare the execution time of version 1 and 3. Plot for each version the :
  - CPU time
  - Speedup & Efficiency
  - `MFLOP/s` = $\frac{FLOPs}{t\times 10^6}$

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

void dmvm(int n, int m,
double *lhs, double *rhs, double *mat)
{
    for (int c = 0; c < n; ++c) {
        int offset = m * c;
        for (int r = 0; r < m; ++r)
            lhs[r] += mat[r + offset] * rhs[c];
    }
}

int main(void)
{
    const int n = 40000; // columns
    const int m = 600;   // rows

    double *mat = malloc(n * m * sizeof(double));
    double *rhs = malloc(n * sizeof(double));
    double *lhs = malloc(m * sizeof(double));

    // initialization
    for (int c = 0; c < n; ++c) {
        rhs[c] = 1.0;
        for (int r = 0; r < m; ++r)
        mat[r + c*m] = 1.0;
    }

    for (int r = 0; r < m; ++r)
        lhs[r] = 0.0;

    dmvm(n, m, lhs, rhs, mat);

    // print result
    printf("lhs:\n");
    for (int r = 0; r < m; ++r)
        printf("lhs[%d] = %.1f\n", r, lhs[r]);


    free(mat);
    free(rhs);
    free(lhs);
    return 0;
}
```