OpenMP Assignment

Name: Akshay Gujjar
Student ID: 027960788

Matrix Multiplication:

- In this assignment, the functionality being enhanced is the multiplication of two matrices.
- The variable NUM_THREADS stores the number of threads, ROW_A stores the number of rows in matrix A, COLUMN_A stores the number of columns in matrix B, ROW_B stores the number of rows in matrix B, COLUMN_B stores the number of columns in matrix B, ROW_A, ROW_RESULT stores the number of rows in the result matrix, COLUMN_B stores the number of columns in the result matrix.
- matrix_multiply_naive is the naïve implementation for calculating the product of two matrices, whereas the matrix_multiply_openMp is the OpenMP implementation.
- time_spent1 returns the time spent for naïve implementation, and time_spent2 returns the time spent for the optimized implementation using OpenMP.
- time_spent1 / time_spent2 gives us the speedup time.
- I was able to achieve 2.6x faster speed using OpenMP.

Compilation and Execution Steps:

- Install docker on your machine. Alternatively, you can also use Online Ubuntu Playground.
- Create a file with the code at the end of this document.
- Compile the code using the following command: gcc -fopenmp openMp_v3.c -o openMp_v3 -Wall
- Print the output using the following command: ./openMp_v3
- Alternatively, you can compile and run using a single command: gcc -fopenmp openMp_v3.c -o openMp_v3 -Wall && ./openMp_v3
- Number of threads used is 2.

Output:

```
root@a0dbd82a1839:/home# gcc -fopenmp openMp_v3.c -o openMp_v3 -Wall && ./openMp_v3
Done random value initialization
Naive Matrix Multiplication of m & n is done
Time elpased is 0.000630 seconds for naive cluster finding.

openMP matrix multliplication done with 2 threads
Time elpased is 0.000234 seconds for OpenMP Matrix multiplication
2.692308 X time faster.
```

Code:

```
#include <stdio.h>
#include <time.h>
#include <unistd.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>
#define NUM_THREADS 2
#define ROW_A 48
#define COLUMN_A 32
#define ROW_B COLUMN_A
#define COLUMN_B 64
#define ROW_RESULT ROW_A
#define COLUMN_RESULT COLUMN_B

void print_matrixA(double matrix[][COLUMN_A]){
#ifdef DEBUG
   printf("matrixA\n");
   for(int i = 0; i < ROW_A; i++){
      for(int j = 0; j < COLUMN_A; j++){
         printf("%4.2f ", matrix[i][j]);
      }
      printf("\n");
   }
#endif
}

void print_matrixB(double matrix[][COLUMN_B]){
#ifdef DEBUG
   printf("matrixB\n");
   for(int i = 0; i < ROW_B; i++){
      for(int j = 0; j < COLUMN_B; j++){
         printf("%4.2f ", matrix[i][j]);
      }
      printf("\n");
   }
#endif
}

void print_result(double matrix[][COLUMN_RESULT]){
#ifdef DEBUG
```

```
   printf("result:\n");
   for(int i = 0; i < ROW_RESULT; i++){
      for(int j = 0; j < COLUMN_RESULT; j++){
         printf("%8.2f ", matrix[i][j]);
      }
      printf("\n");
   }
#endif
}

void matrix_multiply_naive(double matrixA[][COLUMN_A],
                           double matrixB[][COLUMN_B],
                           double result[][COLUMN_RESULT]){
   double sum;
   for(int row = 0; row < ROW_A; row++){
      for(int col=0; col < COLUMN_B; col++){
         sum = 0.0;
         for(int i = 0; i < COLUMN_A; i++) {
            sum += (*(*(matrixA + row) + i)) * (*(*(matrixB + i) + col));
         }
         *(*(result + row) + col) = sum;
      }
   }
}

void matrix_multiply_openMp(double matrixA[][COLUMN_A],
                           double matrixB[][COLUMN_B],
                           double result[][COLUMN_RESULT]){
   double sum;
   int row;
   int col;
   int i;
   #pragma omp parallel for private(sum, row, col, i) num_threads(NUM_THREADS)
   for(row = 0; row < ROW_A; row++){
      for(col=0; col < COLUMN_B; col++){
         sum = 0.0;
         for(i = 0; i < COLUMN_A; i++) {
            sum += (*(*(matrixA + row) + i)) * (*(*(matrixB + i) + col));
         }
         *(*(result + row) + col) = sum;
      }
   }
}
```

```
int main()
{
   double m[ROW_A][COLUMN_A];
   double n[ROW_B][COLUMN_B];
   double result_naive[ROW_RESULT][COLUMN_RESULT];
   double result_openMp[ROW_RESULT][COLUMN_RESULT];

   // Fill some random data
   for (int i = 0; i < ROW_A ; i++){
      for(int j=0; j < COLUMN_A; j++){
         m[i][j] = rand() % (1<<3);
      }
   }

   for (int i = 0; i < ROW_B ; i++){
      for(int j=0; j < COLUMN_B; j++){
         n[i][j] = rand() % (1<<3);
      }
   }
   printf("Done random value initialization\n");
   print_matrixA(m);

   print_matrixB(n);

   double time_spent1 = 0.0;
   double time_spent2 = 0.0;
   ////////////
   clock_t begin = clock();
   matrix_multiply_naive(m, n, result_naive);
   clock_t end = clock();
   printf("Naive Matrix Multiplication of m & n is done\n");
   print_result(result_naive);
   time_spent1 += (double)(end - begin) / CLOCKS_PER_SEC;
   printf("Time elpased is %f seconds for naive matrix multiplication.\n\n", time_spent1);
   ///////////////
   begin = clock();
   matrix_multiply_openMp(m, n, result_openMp);
   end = clock();
   printf("openMP matrix multliplication done with %0d threads\n", NUM_THREADS);
   time_spent2 += (double)(end - begin) / CLOCKS_PER_SEC;
   print_result(result_openMp);
   printf("Time elpased is %f seconds for OpenMP Matrix multiplication \n", time_spent2);
   printf("%f X time faster.\n", time_spent1/time_spent2);
   // Free the memory
```

```
    return 0;
}
```