CS3811 - High Performance Computing and Big Data Lab

Lab 4

> Name: M K Lokesh Kumar

> Registration No.: 2201113026

> Class: Cyber Security(Semester 5)

Experiment 1

Objective

To time the different loop scheduling modes available in OpenMP.

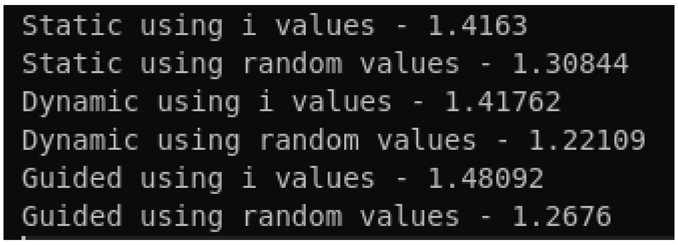
Code

Written in C++.

```cpp
#include <iostream>
#include <random>
#include <omp.h>
#include <time.h>
#include <chrono>
#include <unistd.h>
#include <fstream>

#define NUM_THREADS 4

using namespace std;

int get_rand_num() {
    static random_device rd;
    static mt19937 gen(rd());
    static uniform_real_distribution<> dis(0, 100);

    return dis(gen);
}

void temp(int time) {
    struct timespec req;
    req.tv_sec = time / 1000;
    req.tv_nsec = (time % 1000) * 1000000;
    nanosleep(&req, nullptr);
}
```

```
int main() {
    omp_set_num_threads(NUM_THREADS);

    ofstream outfile("data4_0.dat");

    // static - using i values
    double start = omp_get_wtime();
    #pragma omp parallel for schedule(static, 4)
    for (int i = 1; i <= 100; i++) {
        temp(i);
    }
    double duration = omp_get_wtime() - start;
    outfile << "Static using i values" << " - " << duration << endl;

    // static - using random values
    start = omp_get_wtime();
    #pragma omp parallel for schedule(static, 4)
    for (int i = 1; i <= 100; i++) {
        int rand = get_rand_num();
        temp(rand);
    }
    duration = omp_get_wtime() - start;
    outfile << "Static using random values" << " - " << duration << endl;

    // dynamic - using i values
    start = omp_get_wtime();
    #pragma omp parallel for schedule(dynamic, 4)
    for (int i = 1; i <= 100; i++) {
        temp(i);
    }
    duration = omp_get_wtime() - start;
    outfile << "Dynamic using i values" << " - " << duration << endl;

    // dynamic - using random values
    start = omp_get_wtime();
    #pragma omp parallel for schedule(dynamic, 4)
    for (int i = 1; i <= 100; i++) {
        int rand = get_rand_num();
        temp(rand);
    }
    duration = omp_get_wtime() - start;
    outfile << "Dynamic using random values" << " - " << duration << endl;

    // guided - using i values
    start = omp_get_wtime();
    #pragma omp parallel for schedule(guided, 4)
    for (int i = 1; i <= 100; i++) {
        temp(i);
    }
    duration = omp_get_wtime() - start;
    outfile << "Guided using i values" << " - " << duration << endl;

    // guided - using random values
    start = omp_get_wtime();
    #pragma omp parallel for schedule(guided, 4)
    for (int i = 1; i <= 100; i++) {
        int rand = get_rand_num();
        temp(rand);
    }
    duration = omp_get_wtime() - start;
    outfile << "Guided using random values" << " - " << duration << endl;

    outfile.close();

    return 0;
}
```

Output



```
Static using i values - 1.4163
Static using random values - 1.30844
Dynamic using i values - 1.41762
Dynamic using random values - 1.22109
Guided using i values - 1.48092
Guided using random values - 1.2676
```

Experiment 2

Objective

To time and compare the serial and parallel versions of a program that perform 1D convolution on a matrix.

Code

Writen in C++.

Serial Code

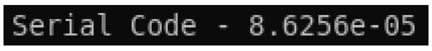
```
#include <iostream>
#include <vector>
#include <random>
#include <omp.h>
#include <fstream>

// Function to perform 1D convolution
std::vector<double> convolve1D(const std::vector<double>& input, const
std::vector<double>& kernel) {
    int input_len = input.size();
    int kernel_len = kernel.size();
    int output_len = input_len + kernel_len - 1;
    std::vector<double> output(output_len, 0.0);

    for (int i = 0; i < input_len; ++i) {
        for (int j = 0; j < kernel_len; ++j) {
            output[i + j] += input[i] * kernel[j];
        }
    }

    return output;
}

int main() {
    // Measure execution time
    double start = omp_get_wtime();

    std::random_device rd;
    std::mt19937 generator(rd()); // Mersenne Twister engine

    // Define the size of the vector
    int size = 100;

    // Create a vector of doubles
    std::vector<double> input(size);

    // Generate random numbers between 1 and 100 (inclusive)
    std::uniform_int_distribution<int> distribution(1, 100);

    // Fill the vector with random numbers
    for (int i = 0; i < size; ++i) {
        input[i] = static_cast<double>(distribution(generator));
    }

    // Define the kernel for convolution
    std::vector<double> kernel = {0.5, 1.0, 0.5};

    // Perform convolution
    std::vector<double> result = convolve1D(input, kernel);

    // Print the result
    for (double value : result) {
        std::cout << value << " ";
    }
    std::cout << std::endl;

    // Measure execution time
    double duration = omp_get_wtime() - start;

    // Output the execution time to a file
    std::ofstream outfile("data4_1.dat");
    outfile << "Serial Code - " << duration << std::endl;
    outfile.close();

    return 0;
```

```
}
```

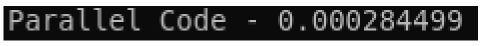
Parallelized code

```
#include <iostream>
#include <vector>
#include <random>
#include <omp.h>
#include <fstream>

// Function to perform 1D convolution
std::vector<double> convolve1D(const std::vector<double>& input, const
std::vector<double>& kernel) {
    int input_len = input.size();
    int kernel_len = kernel.size();
    int output_len = input_len + kernel_len - 1;
    std::vector<double> output(output_len, 0.0);

    #pragma omp parallel for
    for (int i = 0; i < output_len; ++i) {
        int start = std::max(0, i - kernel_len + 1);
        int end = std::min(i + 1, input_len);
        for (int j = start; j < end; ++j) {
            output[i] += input[j] * kernel[i - j];
        }
    }
    return output;
}

int main() {
    // Measure execution time
    double start = omp_get_wtime();

    // Define the size of the vector
    int size = 100;

    // Create a vector of doubles
    std::vector<double> input(size);

    // Use a single random number generator to ensure deterministic output
    std::random_device rd;
    std::mt19937 generator(rd());
    std::uniform_int_distribution<int> distribution(1, 100);

    // Fill the vector with random numbers
    #pragma omp parallel for
    for (int i = 0; i < size; ++i) {
        input[i] = static_cast<double>(distribution(generator));
    }
    // Define the kernel for convolution
    std::vector<double> kernel = {0.5, 1.0, 0.5};

    // Perform convolution
    std::vector<double> result = convolve1D(input, kernel);

    // Print the result
    for (double value : result) {
        std::cout << value << " ";
    }
    std::cout << std::endl;

    // Measure execution time
    double duration = omp_get_wtime() - start;

    // Output the execution time to a file
    std::ofstream outfile("data4_2.dat");
    outfile << "Parallel Code - " << duration << std::endl;
    outfile.close();

    return 0;
}
```

Output

- Serial code 

```
Serial Code - 8.6256e-05
```

- Parallelized code 

```
Parallel Code - 0.000284499
```