OpenMP Parallel Code for Least Squares Linear Regression

Problem Statement

Write a parallel code to solve the **least squares** problem for linear regression. Given a set of n data points (x_i, y_i), the goal is to find the parameters m (slope) and b (intercept) of the line $y = mx + b$ that minimizes the sum of squared errors. The formula for calculating m and b is:

$$m = \frac{n \cdot \sum XY - \sum X \cdot \sum Y}{\text{denom}}$$

$$b = \frac{\sum XX \cdot \sum Y - \sum X \cdot \sum XY}{\text{denom}}$$

where:

$$\text{denom} = n \cdot \sum XX - (\sum X)^2$$

The problem consists of the following tasks:

- **Serial Code**: Implement a serial (non-parallel) version of the least squares solution. The program should:
  - Allocate memory for the arrays of x and y data points.
  - Compute the necessary sums: $\sum X$, $\sum Y$, $\sum XY$, and $\sum XX$.
  - Calculate the slope m and intercept b using the formulas above.
- **Parallel Code**: Implement a parallel version of the least squares solution using OpenMP. The program should:
  - Use OpenMP to parallelize the summation operations for $\sum X$, $\sum Y$, $\sum XY$, and $\sum XX$.
- **Report - Thread vs Time**: Run the parallel code for various numbers of threads: **1, 2, 4, 6, 8, 10, 12, 16, 20, 32, 64**. For each run:
  - Measure the time taken for the least squares computation.
  - Report the execution time for each thread count.

- **Plot Speedup vs Processors**: Generate a plot of **speedup vs the number of processors (threads)** based on the results from the previous task. The speedup is calculated as:

$$\text{Speedup} = \frac{\text{Serial Time}}{\text{Parallel Time}}$$

- **Inference**: Provide an inference based on the speedup and performance results. Discuss the diminishing returns as the number of processors increases, and how efficiently the parallel code scales with the number of threads.

Serial Code

```
#include <iostream>
#include <omp.h>
#include <cmath>
using namespace std;

const double LsEpsilon = 1.0e-12;

//y = mx + b
bool calcLeastSquaresCPP(const double* x, const double* y,
    int n, double* m, double* b);

// const int n = 1024 * 1024;
const int n = 5000 * 5000;
// alignas(64) double x[n];
// alignas(64) double y[n];
double x[n];
double y[n];

int main()
{

    // Initialize the input vectors
 // All points lie on the line y = 1*x + 0.5
 const double slope = 1.0;
 const double y_int = 0.5;
    double start_time, end_time;

    start_time = omp_get_wtime();

 for (int i = 0; i < n; i += 4)
 {
  x[i] = i;
  y[i] = slope * x[i] + y_int;
  x[i+1] = (i+1);
  y[i+1] = slope * x[i+1] + y_int;
  x[i+2] = (i+2);
  y[i+2] = slope * x[i+2] + y_int;
  x[i+3] = (i+3);
  y[i+3] = slope * x[i+3] + y_int;
 }

    double m1 = 0, b1 = 0;
    bool rv1;

    rv1 = calcLeastSquaresCPP(x, y, n, &m1, &b1);

    end_time = omp_get_wtime();

    double time_taken = end_time - start_time;

    printf("slope = %6.4lf, m1 = %6.4lf\n", slope, m1);
    cout << "y_int = " << y_int << ", b1 = " << b1 << endl;
    cout << "rv1 = " << rv1 << endl;

    printf("Threads: %d, Time: %f seconds\n",
        omp_get_num_threads(), time_taken);
}

bool calcLeastSquaresCPP(const double* x, const double* y, int n,
    double* m, double* b) {
    if (n <= 0) {
        return false;
    }

    double sumX = 0, sumY = 0, sumXY = 0, sumXX = 0;
    for(int i=0; i<n; i++) {
        sumX += x[i];
        sumY += y[i];
        sumXX += x[i] * x[i];
        sumXY += x[i] * y[i];
    }

    double denom = n * sumXX - sumX * sumX;
    if (LsEpsilon >= fabs(denom)) return false;

    *m = (n * sumXY - sumX * sumY) / denom;
    *b = (sumXX*sumY - sumX*sumXY) / denom;
    return true;
}
```

Parallel Code Using OpenMP

```
#include <iostream>
#include <omp.h>
#include <cmath>
using namespace std;

const double LsEpsilon = 1.0e-12;

//y = mx + b
bool calcLeastSquaresCPP(const double* x, const double* y,
    int n, double* m, double* b);

const int n = 5000 * 5000;
// alignas(64) double x[n];
// alignas(64) double y[n];
double x[n];
double y[n];

int main()
{
    const double slope = 1.0;
    const double y_int = 0.5;
    double start_time, end_time;
    double time_with_1_thread;

    // Loop over different thread counts
    for (int num_threads = 1; num_threads <= 64; num_threads *= 2) {
        omp_set_num_threads(num_threads);
        // Initialize the input vectors
        // All points lie on the line y = 1*x + 0.5
        start_time = omp_get_wtime();

        #pragma omp parallel for
        for (int i = 0; i < n; i+=4)
        {
            x[i] = i;
            y[i] = slope * x[i] + y_int;
            x[i+1] = (i+1);
            y[i+1] = slope * x[i+1] + y_int;
            x[i+2] = (i+2);
            y[i+2] = slope * x[i+2] + y_int;
            x[i+3] = (i+3);
            y[i+3] = slope * x[i+3] + y_int;
        }

        double m1 = 0, b1 = 0;
        bool rv1;

        rv1 = calcLeastSquaresCPP(x, y, n, &m1, &b1);

        end_time = omp_get_wtime();

        // Capture the time with 1 thread for calculating speedup
        if (num_threads == 1) {
            time_with_1_thread = end_time - start_time;
        }

        double time_taken = end_time - start_time;
        double speedup = time_with_1_thread / time_taken;

        printf("slope = %6.4lf, m1 = %6.4lf\n", slope, m1);
        cout << "y_int = " << y_int << ", b1 = " << b1 << endl;
        cout << "rv1 = " << rv1 << endl;

        printf("Threads: %d, Time: %f seconds, Speedup: %f\n",
            num_threads, time_taken, speedup);
    }
}

bool calcLeastSquaresCPP(const double* x, const double* y, int n,
    double* m, double* b) {
    if (n <= 0) {
        return false;
    }

    double sumX = 0, sumY = 0, sumXY = 0, sumXX = 0;
    #pragma omp parallel for reduction(+:sumX, sumY, sumXY, sumXX)
    for(int i=0; i<n; i++) {
        sumX += x[i];
        sumY += y[i];
        sumXX += x[i] * x[i];
        sumXY += x[i] * y[i];
    }

    double denom = n * sumXX - sumX * sumX;
    if (LsEpsilon >= fabs(denom)) return false;
    *m = (n * sumXY - sumX * sumY) / denom;
    *b = (sumXX*sumY - sumX*sumXY) / denom;

    return true;
}
```

Output

The output of the code for the least squares problem shows the time taken for the serial and parallel implementations, using various thread counts.

Serial Least Squares Output:

```
gcc -o least_squares_serial least_squares_serial.c
./least_squares_serial
slope = 1.0000, m1 = 1.0000
y_int = 0.5, b1 = 0.500038
rv1 = 1
Threads: 1, Time: 0.426203 seconds
```

Parallel Least Squares Output:

```
gcc -fopenmp -o least_squares_parallel least_squares_parallel.c
./least_squares_parallel
slope = 1.0000, m1 = 1.0000
y_int = 0.5, b1 = 0.500038
rv1 = 1
Threads: 1, Time: 0.431912 seconds, Speedup: 1.000000
slope = 1.0000, m1 = 1.0000
y_int = 0.5, b1 = 0.500031
rv1 = 1
Threads: 2, Time: 0.113023 seconds, Speedup: 3.821445
slope = 1.0000, m1 = 1.0000
y_int = 0.5, b1 = 0.500024
rv1 = 1
Threads: 4, Time: 0.086315 seconds, Speedup: 5.003887
slope = 1.0000, m1 = 1.0000
y_int = 0.5, b1 = 0.500017
rv1 = 1
Threads: 8, Time: 0.063812 seconds, Speedup: 6.768531
slope = 1.0000, m1 = 1.0000
y_int = 0.5, b1 = 0.500014
rv1 = 1
Threads: 16, Time: 0.062280 seconds, Speedup: 6.934996
slope = 1.0000, m1 = 1.0000
```

```
y_int = 0.5, b1 = 0.500009
rv1 = 1
Threads: 32, Time: 0.061401 seconds, Speedup: 7.034291
slope = 1.0000, m1 = 1.0000
y_int = 0.5, b1 = 0.500007
rv1 = 1
Threads: 64, Time: 0.065984 seconds, Speedup: 6.545742
```

Plot Time vs Number of Threads for Least Square Algorithm

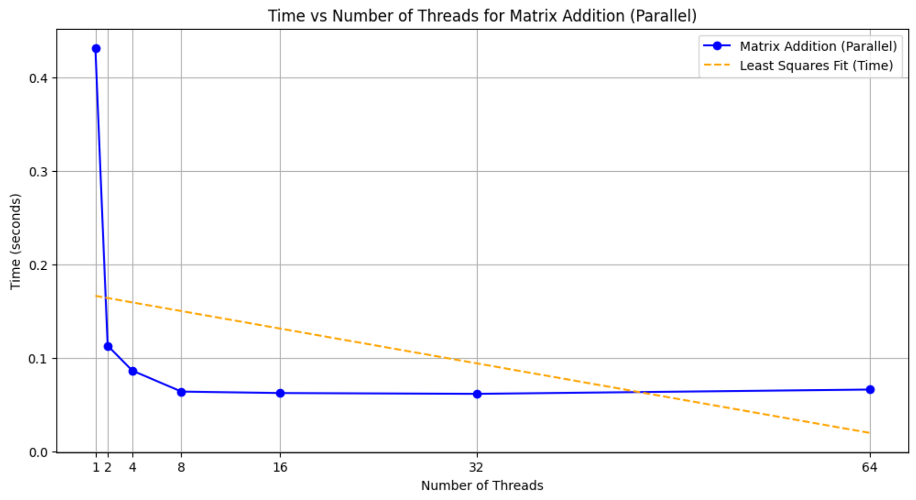

Figure 1: Time vs Processor (Thread Count)

Plot Speedup vs Processors

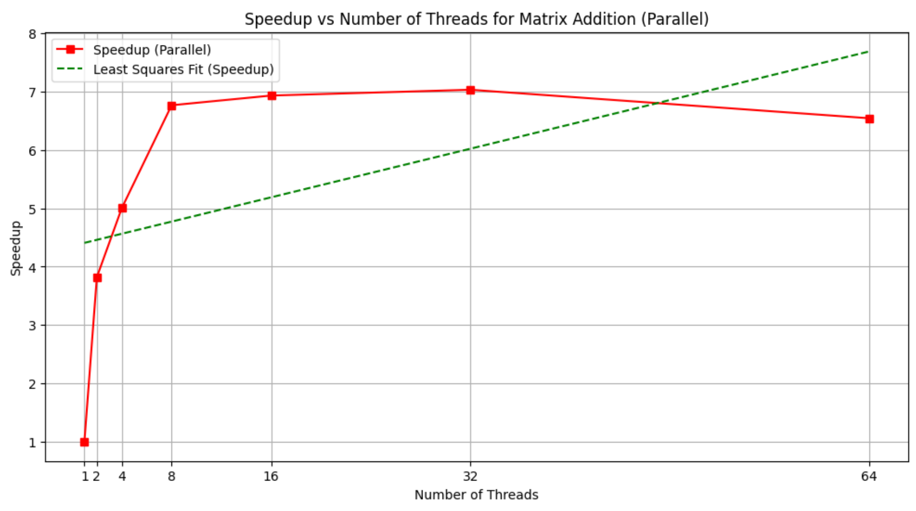

Figure 2: Speedup vs Number of Processors (Threads) for Least Squares Linear Regression

Estimation of Parallelization Fraction

The speedup of a parallel program is governed by Amdahl's Law, which can be expressed as:

$$S_p = \frac{1}{(1-P) + \frac{P}{N}}$$

Where:

- S_p is the speedup achieved with N processors.
- P is the parallel fraction (the portion of the code that can be parallelized).
- N is the number of processors (or threads).

From this law, we can derive the parallel fraction P as follows:

$$P = \frac{N\,(S_p - 1)}{S_p\,(N-1)}$$

Where:

- P is the parallelization fraction.
- S_p is the observed speedup with N processors.
- N is the number of processors (threads).

Parallelization Fraction for Least Square Algorithm

Using the observed speedups for Least Square Algorithm, we can estimate the parallel fraction for each number of processors. Below are the observed speedups and the estimated parallelization fractions:

Threads (Processors)	Speedup (S_p)	Estimated Parallel Fraction (P)
1	1.000000	-
2	3.821445	-
4	5.003887	-
8	6.768531	0.974009
16	6.934996	0.912857
32	7.034291	0.885511
64	6.545742	0.860677

Table 1: Parallelization Fraction for Least Squares Algorithm

Inference and Observations

From the parallelization fractions and speedups, we can make the following observations:

- As the number of threads increases, the parallelization fraction tends to decrease, indicating diminishing returns in performance.
- The highest parallelization fraction is observed at 8 threads, with a value of 0.974009.
- As the number of threads increases beyond 8, the parallel fraction decreases, with the lowest observed at 64 threads (0.860677).
- The diminishing returns become particularly noticeable as the number of threads exceeds 8, where the parallel fraction stabilizes and slightly decreases.
- For 32 and 64 threads, the observed speedup is relatively lower compared to the lower thread counts, highlighting the impact of overhead and non-parallelizable portions of the code.