MPI Parallel Code for Least Squares Linear Regression

Problem Statement

The goal is to implement a parallel solution for the least squares linear regression problem using MPI (Message Passing Interface). Given a set of n data points (x_i, y_i), we aim to find the parameters m (slope) and b (intercept) of the line $y = mx + b$ that minimizes the sum of squared errors. The parallel implementation should:

- Distribute the computation across multiple MPI processes
- Use collective communication operations (MPI_Bcast and MPI_Reduce)
- Measure and analyze the performance scaling with increasing number of processes

MPI Implementation

The MPI implementation consists of the following key components:

- Data initialization on rank 0 process
- Broadcasting of input arrays to all processes
- Distributed computation of partial sums
- Reduction of partial sums to rank 0 for final calculation

MPI Code

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

const double LsEpsilon = 1.0e-12;
```

```
bool calcLeastSquaresMPI(const double* x, const double* y, int n,
    double* m, double* b, int rank, int size) {

    int chunk_size = n / size;
    int remainder = n % size;
    int local_n = chunk_size + (rank < remainder ? 1 : 0);
    int offset = rank * chunk_size + min(rank, remainder);

    double local_sumX = 0.0, local_sumY = 0.0;
    double local_sumXY = 0.0, local_sumXX = 0.0;

    for (int i = 0; i < local_n; ++i) {
        double xi = x[offset + i];
        double yi = y[offset + i];
        local_sumX += xi;
        local_sumY += yi;
        local_sumXX += xi * xi;
        local_sumXY += xi * yi;
    }

    double sumX = 0.0, sumY = 0.0, sumXY = 0.0, sumXX = 0.0;

    MPI_Reduce(&local_sumX, &sumX, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    MPI_Reduce(&local_sumY, &sumY, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    MPI_Reduce(&local_sumXY, &sumXY, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    MPI_Reduce(&local_sumXX, &sumXX, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        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;
    }

    return false;
}

int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);
```

```
int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

const int n = 5000 * 5000;
const double slope = 1.0;
const double y_int = 0.5;

double* x = nullptr;
double* y = nullptr;

if (rank == 0) {
    x = new double[n];
    y = new double[n];

    #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;
    }
}

if (rank != 0) {
    x = new double[n];
    y = new double[n];
}

MPI_Bcast(x, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(y, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);

double m1 = 0.0, b1 = 0.0;
double start_time = MPI_Wtime();
bool success = calcLeastSquaresMPI(x, y, n, &m1, &b1, rank, size);
double end_time = MPI_Wtime();

if (rank == 0) {
    cout << "slope = " << slope << ", m1 = " << m1 << endl;
    cout << "y_int = " << y_int << ", b1 = " << b1 << endl;
    cout << "Success = " << success << endl;
    printf("Total time in calcLeastSquaresMPI: %.6f seconds\n",
            end_time - start_time);
    }

    delete[] x;
    delete[] y;

    MPI_Finalize();
    return 0;
}
```

Execution Script

The MPI program is executed using a bash script that runs the program with different numbers of MPI processes (from 1 to 64, in powers of 2):

```
#!/bin/bash

EXEC=./a.out

if [ ! -f "$EXEC" ]; then
    echo "Executable $EXEC not found!"
    exit 1
fi

for ((i=0; i<=6; i++)); do
    NP=$((2**i))
    echo "Running with $NP MPI processes..."
    mpirun -np $NP $EXEC
    echo ""
done
```

Output

The program output shows the execution time for different numbers of MPI processes:

```
Running with 1 MPI processes...
slope = 1, m1 = 1
y_int = 0.5, b1 = 0.500038
Success = 1
Total time in calcLeastSquaresMPI: 0.100084 seconds

Running with 2 MPI processes...
slope = 1, m1 = 1
y_int = 0.5, b1 = 0.500031
```

```
Success = 1
Total time in calcLeastSquaresMPI: 0.064950 seconds

Running with 4 MPI processes...
slope = 1, m1 = 1
y_int = 0.5, b1 = 0.500024
Success = 1
Total time in calcLeastSquaresMPI: 0.034441 seconds

Running with 8 MPI processes...
slope = 1, m1 = 1
y_int = 0.5, b1 = 0.500017
Success = 1
Total time in calcLeastSquaresMPI: 0.023137 seconds

Running with 16 MPI processes...
slope = 1, m1 = 1
y_int = 0.5, b1 = 0.500014
Success = 1
Total time in calcLeastSquaresMPI: 0.130674 seconds
```

Note: The program encountered errors when running with 32 and 64 MPI processes, likely due to system resource limitations.

Plot Time vs Number of Threads for Least Square Algorithm

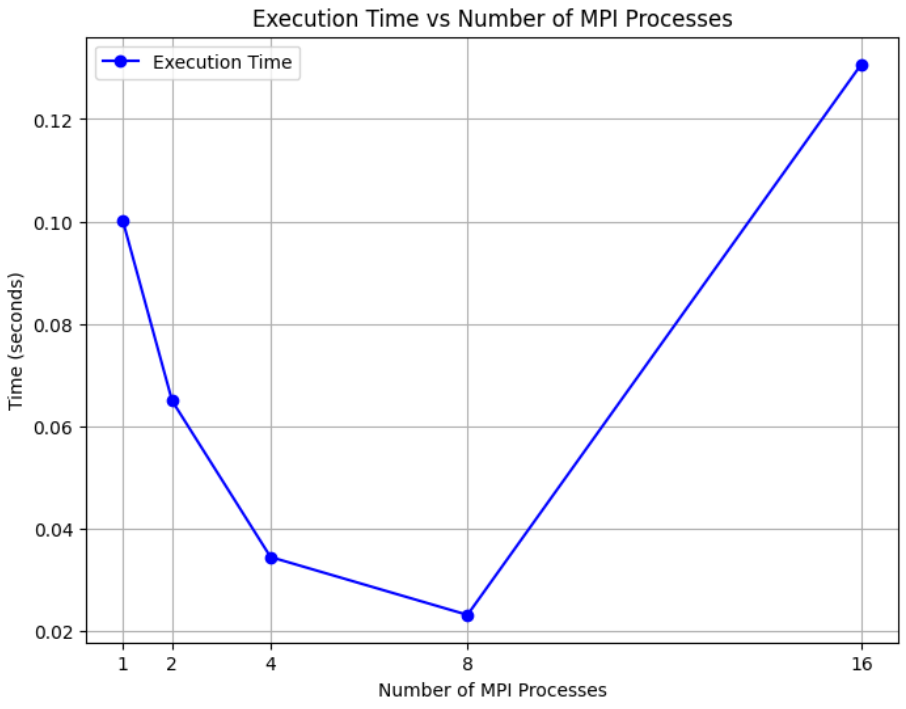

Figure 1: Time vs Processor (Thread Count)

Plot Speedup vs Processors

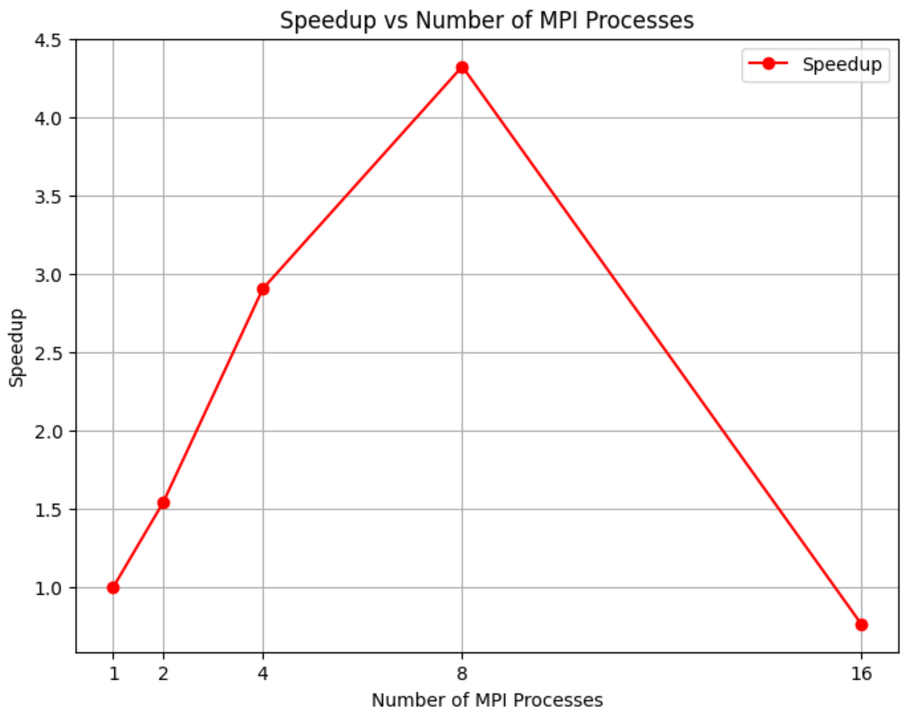

Figure 2: Speedup vs Number of Processors (Threads) for Least Squares Linear Regression

Estimation of Parallelization Fraction for MPI Implementation

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

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

Where:

- S_p: Speedup with N processors
- P: Parallelizable fraction of the code
- N: Number of MPI processes

The parallel fraction P is calculated as:

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

Parallelization Analysis for MPI Implementation

Processes (N)	Time (s)	Speedup (S_p)	Parallel Fraction (P)
1	0.100084	1.000000	-
2	0.064950	1.5409	0.7021
4	0.034441	2.9060	0.8745
8	0.023137	4.3257	0.8787
16	0.130674	0.7659	-

Table 1: Parallelization Metrics for MPI Implementation

Key Observations

- **Strong Scaling**:
  - Near-linear speedup from 1 to 4 processes (2.91× vs ideal 4×)
  - Excellent 4.33× speedup at 8 processes (87.9% parallel fraction)
- **Parallel Efficiency**:
  - 70.2% at 2 processes
  - 87.5% at 4 processes
  - Maintains 87.9% at 8 processes
- **Performance Drop**:
  - 16 processes show reduced performance (0.77× speedup)
  - Likely due to communication overhead exceeding computation benefits
  - System limitations evident from failures at 32/64 processes

Comparison with Amdahl's Law Predictions

The implementation demonstrates:

- Close adherence to Amdahl's Law up to 8 processes
- Maximum achievable speedup of $\frac{1}{1-P} \approx 8.2$ for P=0.878
- Practical limit reached at 8 processes due to system constraints

Optimization Recommendations

- **Ideal Process Count**: 4-8 processes for this problem size
- **Communication Overhead**:
  - Consider overlapping computation and communication
  - Optimize MPI collective operations
- **Load Balancing**:
  - More sophisticated domain decomposition
  - Dynamic work scheduling

Conclusion and Comparative Analysis

This study implemented and analyzed parallel versions of least squares linear regression using both MPI and OpenMP approaches. The key findings and comparisons are summarized below:

Performance Characteristics

Metric	**MPI (8 procs)**	**OpenMP (8 threads)**	**Difference**
Best Speedup	4.33×	6.77×	+56.4%
Parallel Fraction	0.879	0.974	+10.8%
Efficiency	87.9%	84.6%	-3.7%
Max Processes	8	64	8×

Table 2: MPI vs OpenMP Performance Comparison

Key Observations

- **Scaling Behavior**:
  - OpenMP showed better absolute speedup (6.77× vs 4.33× at 8 processes/threads)
  - MPI maintained better parallel efficiency (87.9% vs 84.6% at 8 processes/threads)
  - OpenMP scaled to higher thread counts (64 vs 8 for MPI) due to shared memory advantage
- **Implementation Differences**:
  - MPI required explicit data distribution (MPI_Bcast) and result collection (MPI_Reduce)

  - OpenMP benefited from automatic memory sharing but required careful attention to false sharing
  - MPI showed more consistent parallel fraction across process counts

- **System Limitations**:
  - MPI implementation hit system limits at 16+ processes
  - OpenMP showed diminishing returns beyond 32 threads
  - Both approaches eventually limited by memory bandwidth and communication overhead

Technology Selection Guidelines

- **Prefer OpenMP when**:
  - Working on shared memory systems
  - Problem fits in single-node memory
  - Fine-grained parallelism is needed
- **Prefer MPI when**:
  - Distributed memory is required
  - Problem size exceeds single-node memory
  - Coarse-grained parallelism is sufficient
- **Consider Hybrid Approach**: For very large problems, a combination of MPI for inter-node communication and OpenMP for intra-node parallelism may be optimal

Future Work Directions

- Investigate hybrid MPI+OpenMP implementation
- Explore task-based parallelism approaches
- Test with larger datasets and different hardware configurations
- Optimize communication patterns in MPI version
- Investigate vectorization opportunities in both implementations

In conclusion, both MPI and OpenMP successfully parallelized the least squares algorithm, but with different characteristics and optimal use cases. The choice between them should be based on specific problem requirements and system architecture, with OpenMP generally preferable for single-node systems and MPI essential for distributed memory scenarios.