CE 791 Assignment 7

Shridhar Thakar, Sanket Yannuwar

November 27, 2024

```
#include <mpi.h>
#include <random>
#include <functional>
#include <iostream>

class monteCarloIntegrator {
private:
    std::mt19937 gen;
    std::uniform_real_distribution<double> dist;
    double a, b;
    size_t nPoints;
    int rank, size;
    //- communicator used by class
    MPI_Comm comm;
    // Add performance tracking members
    double computeTime;
    size_t flopCount;

public:
    //- constructor
    monteCarloIntegrator(
        double lower_limit,
        double upper_limit,
        size_t nPoints,
        std::random_device& rd,
        MPI_Comm global_comm
    ) :
        a(lower_limit),
        b(upper_limit),
```

```
        nPoints(nPoints),
        gen(rd()),
        computeTime(0.0),
        flopCount(0),
        comm(global_comm)
    {
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        gen.seed(rd() + rank);
        dist = std::uniform_real_distribution<double>(a,b);
    }

    // Add getter for performance metrics
    std::pair<double, size_t> getPerformanceMetrics() const {
        return {computeTime, flopCount};
    }

    double integrate(const std::function<double(double)>& func) {
        double local_sum = 0.0;
        double global_sum = 0.0;
        double start_time = MPI_Wtime();  // Start timing

        if (rank == 0) {  // Master process
            size_t points_per_slave = nPoints / (size - 1);
            size_t remaining_points = nPoints % (size - 1);

            for (int i = 1; i < size; ++i) {
                size_t slave_points = points_per_slave + (i <= remaining_p
                MPI_Send(&slave_points, 1, MPI_UNSIGNED_LONG, i, 0, MPI_COM
            }

            for (int i = 1; i < size; ++i) {
                double slave_result;
                MPI_Recv(&slave_result, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORL
                global_sum += slave_result;  // 1 FLOP (addition)
            }

            // Final calculation
            global_sum = (b - a) * (global_sum / nPoints);
// 3 FLOPs (subtraction, multiplication, division)
```

```
            flopCount = (size - 1) + 3;  // Count master's FLOPs
        }
        else
        {  // Slave processes
          size_t local_points;
          MPI_Recv(&local_points, 1, MPI_UNSIGNED_LONG, 0, 0, MPI_COMM_WOR

            // Process points and count FLOPs
          for (size_t i = 0; i < local_points; ++i)
          {
            double x = dist(gen);
            double fx = func(x);        // 2 FLOPs for x * x
            local_sum += fx;            // 1 FLOP for addition
          }

          // Each slave's FLOP count (for x^2 function)
          flopCount = local_points * 3;  // 3 FLOPs per iteration

          MPI_Send(&local_sum, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
        }

          // Record computation time
        computeTime = MPI_Wtime() - start_time;

        // Gather total FLOP counts from all processes
        size_t global_flops;
        MPI_Reduce(&flopCount, &global_flops, 1, MPI_UNSIGNED_LONG, MPI_SUM,
        flopCount = global_flops;
        return (rank == 0) ? global_sum : 0.0;
    }
};

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);

    std::random_device rd;
```

```
    double lower_bound = 0.0;
    double upper_bound = 1.0;
    size_t num_points = 1000000;

    monteCarloIntegrator integrator(lower_bound, upper_bound, num_points, r
    auto func = [](double x) { return x * x; };

    double result = integrator.integrate(func);

    if (rank == 0) {
        double analytical = 1.0/3.0;
        auto [compute_time, flop_count] = integrator.getPerformanceMetrics
        double mflops = (flop_count / compute_time) / 1e6;

        std::cout << "Performance Metrics:" << std::endl;
        std::cout << "Number of processes: " << size << std::endl;
        std::cout << "Computation time: " << compute_time << " seconds" <<
        std::cout << "Total FLOPs: " << flop_count << std::endl;
        std::cout << "MFLOPS: " << mflops << std::endl;
        std::cout << "\nResults:" << std::endl;
        std::cout << "Monte Carlo estimate: " << result << std::endl;
        std::cout << "Analytical solution: " << analytical << std::endl;
        std::cout << "Absolute error: " << std::abs(result - analytical) <<
    }

    MPI_Finalize();
    return 0;
}
```