Parallel Monte Carlo Simulation for Pi Calculation

Our team:

Yáred Iessé Bustillo Medina - 57963128

Liana Mikhailova - 43583189

Josué David Pavón - 93116997

Introduction

Random sampling is a statistical technique used to answer complex questions in Monte Carlo simulations. This study estimates the value of Pi (π) using parallel Monte Carlo designs. The relationship is calculated by selecting points on the unit square and counting the number of points on the numbered circle. Parallel computing spreads the workload across multiple processors, increasing performance and efficiency. This work demonstrates the effectiveness and computational advantages of parallel Monte Carlo simulations for π values.

Project Objective:

Monte Carlo simulation to estimate the value of Pi number using OpenMP.

Description:

Use the Monte Carlo method to use random sampling.
The goal is to estimate the value of Pi, using the random method to get points in a unit square and count how many points fall within a quarter circle.

Steps:

1. Do Monte Carlo simulation to estimate the number of Pi in sequential mode.
2. use OpenMP. So that at the same time we have random point generation and counting running.
3. Update the common variables and validate them.
4. Testing - running points, their random number and threads. This is to analyse our performance.
5. Checking our performance and accuracy along with their parallels and consistency.

Our concept to the questions

- What is Monte Carlo method?

Monte Carlo method is a technique that has random sampling to solve problems. And these problems can be determinable.
The method is named after the casino at Monte Carlo. The same randomness in random sampling.

- What is random sampling?

Random sampling in parallel programming entails dividing the work of creating random samples among several processors or processing units. Random points, which are frequently used in simulations such as Monte Carlo methods, are independently generated by each processor and then combined to produce an extensive dataset. The accuracy and performance of operations like predicting the value of Pi or other probabilistic computations are improved by this parallel technique, which can handle larger sample volumes and faster computations.

- What are the advantages of parallel programming for Monte Carlo simulations?

In order to greatly increase computational efficiency, parallel programming divides the workload across several processors. This makes it possible for more random points to be created and analyzed

concurrently during Monte Carlo simulations, which results in quicker and more precise Pi calculations.

❖ Why and why do we use it to estimate Pi?

Estimating Pi = the path of randomly generating points within a unit square. So there are also checks - how many of them fall within the quarter circle inscribed in that square.
The reciprocal relationship between the dots inside the circle to the total number of dots is used to estimate the value of Pi.
This method helps us to generally understand the workings and the whole system of random sampling and statistical estimation.

❖ How does the Monte Carlo method of Pi estimation work with random point generation?

In order to generate random points, pairs of random x and y coordinates inside the range [0, 1] must be created. The next step involves evaluating these locations to see if they are part of the quarter circle that is given by $x^2 + y^2 \leq 1$. To estimate Pi, one uses the ratio of points inside the quarter circle to all points.

❖ What is OpenMP?

OpenMP is an API (application programming interface) for programming multithreaded applications in C, C++ and Fortran.
In general, it helps us to easily add parallelism to programs. This is needed to improve performance.
OpenMP uses directives that tell the compiler which parts of code should be executed in parallel.
Directives in OpenMP are special instructions that we can embed just for parallelism.

➔ Example:

Here we have a C programme to estimate Pi using the Monte Carlo method. In the sequential version we generate points and check if they are inside the circle in one thread. And with the help of OpenMP we can parallelise this process so that several threads generate and check points simultaneously. It is necessary to speed up the execution of the programme.

- ❖ What are the essential stages in utilizing OpenMP to implement Monte Carlo simulation for Pi estimation?

1. To comprehend the fundamental procedure, run the Monte Carlo simulation in sequential mode.
2. To parallelize the activities of counting and generating random points, use OpenMP.
3. Make sure that shared common variables between threads are updated and validated correctly.
4. To evaluate performance, run the simulation with varying point counts, random seeds, and thread counts to test the implementation.
5. Assess the accuracy and performance, making sure that the parallel execution is accurate and consistent.

- ❖ In OpenMP, how do you verify the common variables that are modified by several threads?

To validate, one must make sure that there are no race conditions and that the shared variables are updated accurately. This can be accomplished when modifying shared variables by ensuring atomicity or mutual exclusion using OpenMP features like #pragma omp atomic or #pragma omp critical.

- ❖ Which criteria would you apply to assess the parallel Monte Carlo simulation's accuracy and performance?

The execution time can be measured and compared to the sequential implementation in order to assess performance. By contrasting the estimated and known values of Pi, one can evaluate accuracy. It is also possible to employ metrics like efficiency (speedup divided by the number of threads) and speedup (ratio of sequential to parallel execution time).

- ❖ What potential obstacles might the Monte Carlo simulation for Pi estimate encounter when it is parallelized, and how might they be overcome?

The management of load balancing among threads, minimizing overhead from parallelization, and guaranteeing thread safety when

changing shared variables are the challenges. By employing the proper synchronization techniques, dynamically assigning work to threads, and streamlining the code to minimize parallelization overhead, they can be handled.

★ Why we chose this project:

Because it is very interesting and thanks to it we can use our knowledge and learn even more in practice - how to manage shared variables and tune performance for parallel programmes.

Description of the classes

Main class

```
#include <iostream> // for std::cout
#include <cstdlib>  // for rand() and srand()
#include <ctime>    // for time()

// Function to calculate Pi using the Monte Carlo method
double calculatePi(int numPoints) {
    int insideCircle = 0; // Counter for points inside the quarter circle

    // Generate points and count how many fall inside the quarter circle
    for (int i = 0; i < numPoints; ++i) {
        double x = static_cast<double>(rand()) / RAND_MAX; // Random x
coordinate
        double y = static_cast<double>(rand()) / RAND_MAX; // Random y
coordinate
        if (x * x + y * y <= 1.0) { // Check if the point is inside the quarter
circle
            ++insideCircle;
        }
    }

    // Estimate Pi based on the ratio of points inside the quarter circle
    return 4.0 * insideCircle / numPoints;
}

int main() {
```

```
    srand(static_cast<unsigned>(time(0))); // Seed the random number
generator
    int numPoints = 1000000; // Number of points to generate
    double pi = calculatePi(numPoints); // Calculate Pi
    std::cout << "Estimated Pi (Serial): " << pi << std::endl; // Print the
result
    return 0;
}
```

This C++ program estimates the value of Pi using the Monte Carlo method. It includes the necessary libraries for input/output operations, random number generation, and time functions. The `calculatePi` function generates a specified number of random points within a unit square and checks if each point lies inside a quarter circle inscribed within the square. The ratio of points inside the quarter circle to the total number of points is used to estimate Pi by multiplying this ratio by 4. In the `main` function, the random number generator is seeded with the current time to ensure different sequences of random numbers in each run, and the `calculatePi` function is called with one million points. The resulting estimate of Pi is then printed to the console.

Parallel Version

```
#include <iostream> // for std::cout
#include <cstdlib>  // for rand() and srand()
#include <ctime>    // for time()
#include <omp.h>    // for OpenMP functions

// Function to calculate Pi using the Monte Carlo method with OpenMP
double calculatePi(int numPoints) {
    int insideCircle = 0; // Counter for points inside the quarter circle

    #pragma omp parallel // Start parallel region
    {
            unsigned int seed = static_cast<unsigned int>(time(NULL)) ^
omp_get_thread_num(); // Unique seed for each thread
        #pragma omp for reduction(+:insideCircle) // Parallel for loop with
reduction
```

```
        for (int i = 0; i < numPoints; ++i) {
            double x = static_cast<double>(rand()) / RAND_MAX; // Random
x coordinate
            double y = static_cast<double>(rand()) / RAND_MAX; // Random
y coordinate
             if (x * x + y * y <= 1.0) { // Check if the point is inside the quarter
circle
                ++insideCircle;
            }
        }
    }

    // Estimate Pi based on the ratio of points inside the quarter circle
    return 4.0 * insideCircle / numPoints;
}

int main() {
     srand(static_cast<unsigned int>(time(0))); // Seed the random number
generator
    int numPoints = 1000000; // Number of points to generate
    double pi = calculatePi(numPoints); // Calculate Pi
      std::cout << "Estimated Pi (Parallel): " << pi << std::endl; // Print the
result
    return 0;
}
```

This enhanced C++ program estimates the value of Pi using the Monte Carlo method, but now includes parallel processing with OpenMP to improve performance. It incorporates the necessary libraries for input/output operations, random number generation, time functions, and OpenMP. The `calculatePi` function uses a parallel region (`#pragma omp parallel`) to allow multiple threads to execute simultaneously. Each thread uses a unique seed for the random number generator to ensure different sequences of random numbers. A parallel for loop (`#pragma omp for`) with a reduction clause is used to sum the points inside the quarter circle across all threads. The ratio of points inside the quarter circle to the total number of points is then multiplied by 4 to

estimate Pi. In the `main` function, the random number generator is seeded with the current time, and the `calculatePi` function is called with one million points. The resulting estimate of Pi is printed to the console.

Parallel version with Performance Measurement

```
#include <iostream> // for std::cout
#include <random> // for std::mt19937 and std::uniform_real_distribution
#include <ctime>  // for time()
#include <omp.h>  // for OpenMP functions

// Function to calculate Pi using the Monte Carlo method with OpenMP
double calculatePi(int numPoints) {
    int insideCircle = 0; // Counter for points inside the quarter circle

    #pragma omp parallel // Start parallel region
    {
        // Create a thread-local random number generator
        std::mt19937 generator(static_cast<unsigned int>(time(NULL)) ^ omp_get_thread_num());
        std::uniform_real_distribution<double> distribution(0.0, 1.0);

        #pragma omp for reduction(+:insideCircle) // Parallel for loop with reduction
        for (int i = 0; i < numPoints; ++i) {
            double x = distribution(generator); // Random x coordinate
            double y = distribution(generator); // Random y coordinate
            if (x * x + y * y <= 1.0) { // Check if the point is inside the quarter circle
                ++insideCircle;
            }
        }
    }

    // Estimate Pi based on the ratio of points inside the quarter circle
    return 4.0 * insideCircle / numPoints;
```

```
}

int main() {
    int numPoints = 1000000; // Number of points to generate

    double start = omp_get_wtime(); // Start timing
    double pi = calculatePi(numPoints); // Calculate Pi
    double end = omp_get_wtime(); // End timing

     std::cout << "Estimated Pi (Parallel): " << pi << std::endl; // Print the result
     std::cout << "Time taken: " << end - start << " seconds" << std::endl; // Print the elapsed time

    return 0;
}
```

This C++ program estimates the value of Pi using the Monte Carlo method, enhanced with parallel processing via OpenMP for improved performance. It utilizes the `std::mt19937` Mersenne Twister engine and `std::uniform_real_distribution` for generating high-quality random numbers. The `calculatePi` function is designed to run in parallel, with each thread generating its own random points. The parallel region (`#pragma omp parallel`) ensures that each thread operates independently with its own random number generator, seeded uniquely using the thread number. A parallel for loop (`#pragma omp for`) with a reduction clause is used to count the number of points that fall inside the quarter circle across all threads. The ratio of these points to the total number of points, multiplied by 4, provides the estimate of Pi. The `main` function sets the number of points to one million and measures the execution time using `omp_get_wtime()`. It then calls `calculatePi` and prints the estimated value of Pi along with the time taken to compute it.

Parallel Version with Threads Configuration

```
#include <iostream> // for std::cout
```

```
#include <random> // for std::mt19937 and std::uniform_real_distribution
#include <ctime>    // for time()
#include <omp.h>    // for OpenMP functions

// Function to calculate Pi using the Monte Carlo method with OpenMP
double calculatePi(int numPoints) {
    int insideCircle = 0; // Counter for points inside the quarter circle

    #pragma omp parallel // Start parallel region
    {
        // Create a thread-local random number generator
            std::mt19937 generator(static_cast<unsigned int>(time(NULL)) ^ omp_get_thread_num());
        std::uniform_real_distribution<double> distribution(0.0, 1.0);

          #pragma omp for reduction(+:insideCircle) // Parallel for loop with reduction
        for (int i = 0; i < numPoints; ++i) {
            double x = distribution(generator); // Random x coordinate
            double y = distribution(generator); // Random y coordinate
              if (x * x + y * y <= 1.0) { // Check if the point is inside the quarter circle
                ++insideCircle;
            }
        }
    }

    // Estimate Pi based on the ratio of points inside the quarter circle
    return 4.0 * insideCircle / numPoints;
}

int main() {
    int numPoints = 1000000; // Number of points to generate
    omp_set_num_threads(4); // Set the number of threads

    double start = omp_get_wtime(); // Start timing
```

```
    double pi = calculatePi(numPoints); // Calculate Pi
    double end = omp_get_wtime(); // End timing

       std::cout << "Estimated Pi (Parallel with 4 threads): " << pi << std::endl; // Print the result
     std::cout << "Time taken: " << end - start << " seconds" << std::endl; // Print the elapsed time

    return 0;
}
```

This C++ program estimates the value of Pi using the Monte Carlo method, enhanced with OpenMP for parallel processing. The `calculatePi` function generates random points and checks if they fall inside a quarter circle, with each thread in the parallel region creating its own random number generator seeded uniquely using the thread number and current time. The `#pragma omp for` directive with a reduction clause ensures that the counting of points inside the circle is done in parallel across multiple threads, efficiently summing up the results. In the `main` function, the number of threads is set to four using `omp_set_num_threads(4)`, and the number of points to generate is set to one million. The execution time is measured using `omp_get_wtime()`, and the estimated value of Pi along with the computation time is printed to the console. This approach leverages multiple threads to speed up the calculation, providing a parallel estimate of Pi and demonstrating the performance benefits of parallel computing.