Sequential and Parallel Monte Carlo Integration

A Performance Comparison

Mustassum Tanvir
6-22-2024

Abstract

This study compares the performance of sequential and parallel Monte Carlo Integration methods, focusing on execution times, speedup, and accuracy of the estimated integral. The analysis shows that while parallel integration offers better performance in terms of execution times, using fewer processes can yield significantly better speedup. Despite some variability in the estimated integral, parallel integration does not show a statistically significant difference in accuracy compared to sequential integration. This suggests that the performance benefits of parallel integration may outweigh concerns about stability, highlighting the importance of considering the trade-offs between accuracy and performance when choosing between sequential and parallel processing methods.

Contents

Introduction

Monte Carlo integration is a powerful technique for estimating definite integrals using random sampling. It is particularly useful for complex integrals in high-dimensional spaces where other methods may struggle. In this report, we compare the performance of Monte Carlo integration implemented in a sequential manner with its parallel implementation using the Message Passing Interface (MPI).

Parallel computing has become essential for tackling computationally intensive tasks. MPI is a widely used standard for writing parallel programs, especially in distributed memory systems. By parallelizing Monte Carlo integration with MPI, we aim to accelerate the integration process by distributing the computational workload across multiple processors.

Implementation

Sequential Implementation

In the sequential implementation of Monte Carlo integration, we first define the limits of integration (a = 0, b = 1) and the number of random points (N = 1000) to be generated. We then define the function $f(x) = x^2$ to be integrated. Next, we iterate N times, generating random points between a and b using NumPy's random.uniform() function. For each iteration, we calculate the integral using the formula

$$\frac{(b-a)}{\mathrm{N}} \sum_{\mathrm{i}=1}^{\mathrm{N}} f(xi)$$

where xi is a random point. We store the calculated integral for each iteration in a list plt_vals. After all iterations, we calculate the mean of plt_vals to estimate the integral value. At the end, we plot the distribution of the calculated areas using Matplotlib's hist() function and display the estimated integral value while execution time for the sequential implementation is measured using Python's time module and printed to the console.

Parallel Implementation

In the parallel implementation of Monte Carlo integration using MPI, we utilize the mpi4py library to parallelize the computation across multiple processes. We again define the limits of integration (a = 0, b = 1) and the total number of random points (N = 1000).

Each process in the MPI communicator generates a subset of random points, calculated as local_N = N // size, where size is the total number of processes. The local_ar array stores these random points. Each process then calculates the local integral of the subset of points using the function $f(x) = x^2$. The local integral is then gathered to the root process (rank 0) using the comm.gather() function, which collects

all local integrals into the integrals array on the root process. The root process (rank 0) calculates the total integral by summing all the gathered local integrals.

Analysis

Execution Times

The execution times were consistently in favour of Parallel integration, which showed better performance throughout the board. The execution times line chart can be seen in the figure below.

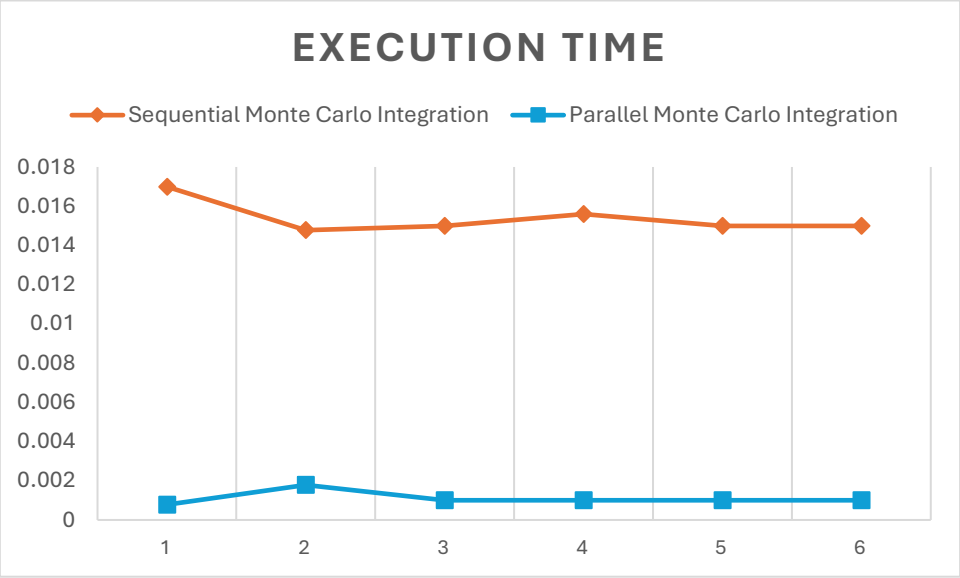

Figure 1. Line Chart of Execution Times.

Speedup

When the speedup of the Monte Carlo Integration was calculated, with the help of the formula

$$\text{Speedup} = \text{Sequential Execution Time}/\text{Parallel Execution Time}$$

It can be seen that 2 processes have significantly better performance when compared to 4 parallel processes. This indicates that the higher overhead causes degradation in the performance, which leads to a lower average speedup.

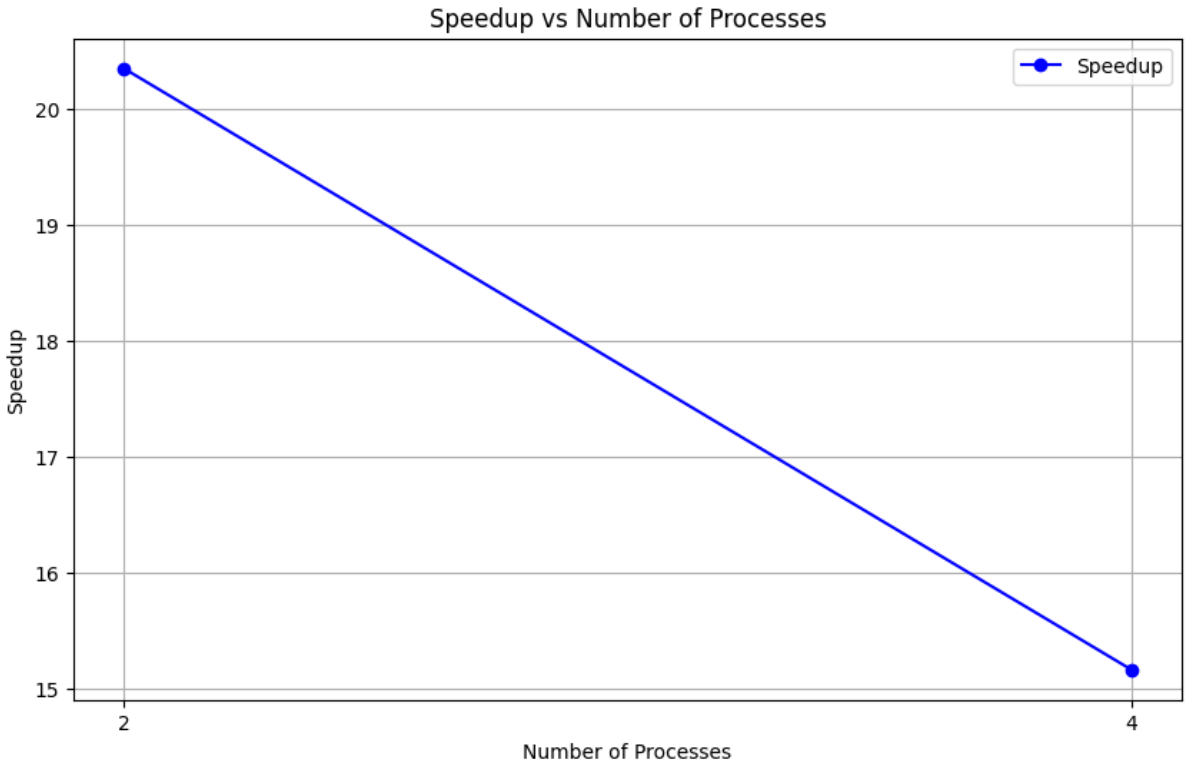

Figure 2. Line Chart of Average Speedup Between Processes.

Estimated Integral

The integral estimation is the bread and butter of this experiment, and no performance increase or speedup is worth anything if it results in inaccurate estimation. From the tests conducted, and their subsequent estimations in the line chart below, it can be seen that while sequential integration was almost always the same, in the range of 0.33 to 0.335, parallel integration bounced around quite a lot.

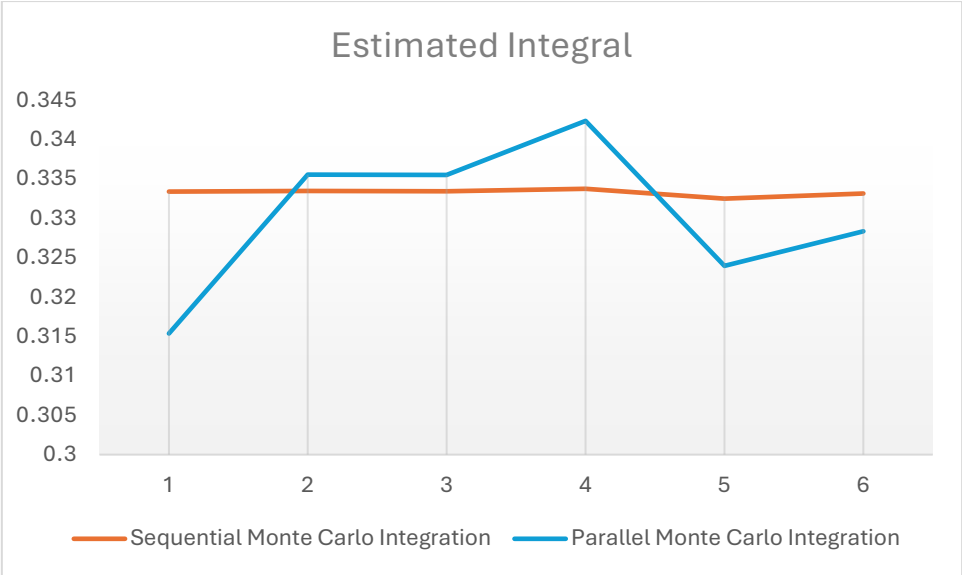

Figure 3. Line Chart of Estimated Integral.

This accuracy of the sequential integration can also be seen from the histogram below, which shows that most of the probable answers fell close to 0.33, which is the accurate answer for this integral as well.

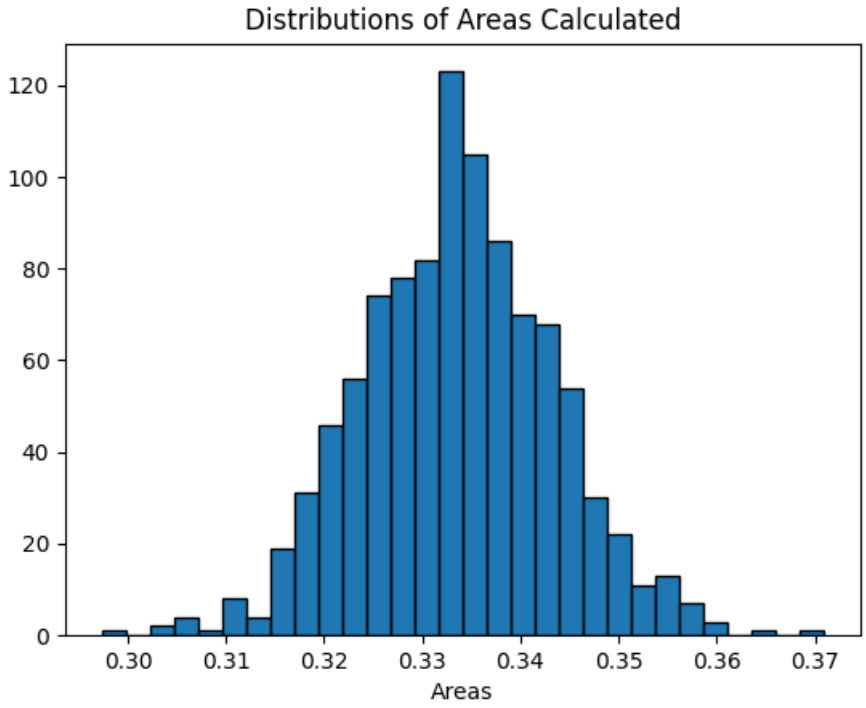

Figure 4. Histogram of probable answers.

However, to quantify this finding, a t-test was conducted assuming unequal variances, which shows that there is not statistically significant different between the two variables. Further showing that the reliability and stability of sequential implementation may not be that important.

	Sequential Monte Carlo Integration	*Parallel Monte Carlo Integration*
Mean	0.333268	0.330168
Variance	1.84E-07	9.32E-05
Observations	6	6
Hypothesized Mean Difference	0	
df	5	
t Stat	0.78563	
P(T<=t) one-tail	0.233831	
t Critical one-tail	2.015048	
P(T<=t) two-tail	0.467662	
t Critical two-tail	2.570582	

Table 1. t-test Results of Estimated Integrals.

Conclusion

The analysis of the Monte Carlo Integration experiment reveals interesting insights into the trade-offs between sequential and parallel processing. Parallel integration consistently outperformed sequential integration in terms of execution times, showcasing better overall performance. However, the analysis also highlighted that using 2 processes yielded significantly better speedup compared to using 4 parallel processes, indicating that higher overhead in managing more processes can lead to performance degradation.

While sequential integration provided more stable and reliable estimates of the integral, parallel integration showed more variability in its estimates. Despite this, a t-test assuming unequal variances did not show a statistically significant difference between the two methods, suggesting that the performance benefits of parallel integration may outweigh concerns about stability. Ultimately, the choice between sequential and parallel integration depends on the specific requirements of the application, balancing the need for accurate estimation with the desire for improved performance.

Appendix

Table for Execution Times

Execution Time	
Sequential Monte Carlo Integration	**Parallel Monte Carlo Integration**
0.016993	0.000781
0.014776	0.001783
0.015001	0.000999
0.015594	0.000999
0.014987	0.000999
0.014998	0.000999

Table for Estimated Integrals

Estimated Integral	
Sequential Monte Carlo Integration	**Parallel Monte Carlo Integration**
0.333373	0.315372
0.333455	0.335515
0.33343	0.335481
0.333735	0.34235
0.332486	0.323946
0.333125	0.328341

Code for Sequential Implementation

```
# Importing the modules

from numpy import random

import numpy as np

import matplotlib.pyplot as plt

import time


# Limits of integration

a = 0

b = 1

N = 1000


# Function to calculate x^2

def f(x):
```

```
    return x**2

# List to store all the values for plotting
plt_vals = []

start_time = time.time()

# Iterate through all the values to generate multiple results
for _ in range(N):
    # Generate random points between a and b
    ar = random.uniform(a, b, N)
    integral = np.sum(f(ar))
    ans = (b - a) / N * integral
    plt_vals.append(ans)

end_time = time.time()
execution_time = end_time - start_time
print(f"Sequential Execution Time: {execution_time} seconds")

# Print the estimated integral
estimated_integral = np.mean(plt_vals)
print(f"Estimated integral: {estimated_integral}")

# Plot the distribution of calculated areas
plt.title("Distributions of Areas Calculated")
plt.hist(plt_vals, bins=30, ec="black")
plt.xlabel("Areas")
plt.show()
```

Code for Parallel Implementation

```
from mpi4py import MPI
import numpy as np
```

```
from numpy import random
import matplotlib.pyplot as plt
import time

# Limits of integration
a = 0
b = 1
N = 1000

# Function to calculate x^2
def f(x):
    return x**2

# Initialize MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

start_time = None
if rank == 0:
    start_time = time.time()

# Each process generates a subset of random points
local_N = N // size
local_ar = random.uniform(a, b, local_N)
local_integral = np.sum(f(local_ar)) * (b - a) / local_N
integrals = comm.gather(local_integral, root=0)

if rank == 0:
    end_time = time.time()
    execution_time = end_time - start_time
```

```
    integral = np.sum(integrals)
    print(f"Parallel Execution Time: {execution_time}
seconds")
    print(f"Estimated integral: {integral}")
    plt_vals = integrals
    plt.title("Distributions of Areas Calculated")
    plt.hist(plt_vals, bins=30, ec="black")
    plt.xlabel("Areas")
    plt.show()
```