Aristotle University of Thessaloniki

Programming Tools: OpenMp

"Estimating π with Monte Carlo Simulation using Parallel Programming"

Zachariadou Anastasia

October 2024

Abstract

The purpose of this project was to estimate the value of π through Monte Carlo Simulations, using openMP for parallelization. To achieve this, three codes were developed. The main program, written in C++, contained the Monte Carlo simulation process, parallelized with the help of OpenMp. The other two codes, written in Python and executed in Jupyter Notebook, were designed to automate the execution of the main program and analyze the results. More specifically, execution time and speedup were evaluated for increasing numbers of threads. The convergence rate of the Monte Carlo simulation was also calculated.

Contents

Chapter 1

Introduction

1.1 Problem Statement

The aim of the task was to **estimate the value of π using the Monte Carlo method**. This method involves repeatedly generating random samples to obtain numerical results. The specific objectives of the task:

- Implement the **Monte Carlo** method to estimate π in **C++**.
- **Parallelize** the code using **OpenMP**.
- Analyze performance in a Jupyter notebook, plotting **execution time** and **speedup**.
- Find the **percentage of code that benefits from parallelization** and the **maximum possible speedup for 10000 cores**.
- Calculate the **convergence rate** of the Monte Carlo method.

1.2 Theoretical Background

In this section, some basic theoretical background regarding key concepts of our study are presented.

1.2.1 Monte Carlo Simulation

The Monte Carlo method is a statistical technique that utilizes **random sampling** to produce estimated for numerical values. In the context of this study, the value of pi was estimated. Random points within a unit square were generated from a uniform distribution and the proportion of points falling inside

a quarter circle inscribed within the square was determined. The value of pi was calculated as:

$$\pi \approx 4 \cdot \frac{N_{\text{inside}}}{N_{\text{total}}} \tag{1.1}$$

1.2.2 Parallel Programming

Parallel programming is a computing paradigm that allows **multiple processes** or threads to execute **simultaneously**. Typically, this approach results in higher efficiency and reduces execution times by dividing the code to smaller, concurrent tasks.

In this project, parallel programming was employed to enhance the performance of our simulation and compare the results for different number of threads and points. This was achieved by utilizing OpenMP. The generation of random points was distributed among the available threads.

1.2.3 Speedup

Speedup is a **measure of the performance improvement** of a parallelized program compared to its sequential counterpart. The formula used:

$$\text{Speedup} = \frac{T_{\text{sequential}}}{T_{\text{parallel}}} \tag{1.2}$$

1.2.4 Amdahl's Law

Amdahl's Law is a formula used to compute the **theoretical speedup** when using multiple processors. In other words, it states that the overall performance improvements resulting from optimizing a part of a system (code) is limited by the fraction of time that the improved part is actually used.

$$S_{latency} = \frac{1}{(1-p) + \frac{p}{s}} \tag{1.3}$$

where: $S_{latency}$ is the theoretical speedup, s is the speedup of the part of the code that benefits from parallelization and p is the proportion of execution time that the part benefiting from parallelization originally occupied.

1.2.5 Convergence Rate

The convergence rate in the context of Monte Carlo simulations refers to **how quickly the estimated value approaches the true value as the number of random samples (points) increases**. For our case the error is expected to

converge at a rate proportional to $O(N^{-1/2})$, where N is the number of random points sampled.

$$\text{Error} \propto \frac{1}{\sqrt{N}} \tag{1.4}$$

In our study, we fit a line to the log-log plot of errors vs number of points. Thus, the slope of this line corresponded to the convergence rate.

$$\log(\text{Error}) = \log(c) - 0.5\log(N) \tag{1.5}$$

$$Y = b + ax \tag{1.6}$$

From 1.5 and 1.6, the theoretical convergence rate is predicted to be:

$$\text{rate}_{theor} = -0.5 \tag{1.7}$$

Chapter 2

Implementation

2.1 C++ Code: Parallelized Monte Carlo Simulation

In this section, the Monte Carlo method implementation created in C++ is presented and explained step by step.

1. **Libraries**:

   Apart from the usual libraries for input and output operations, the libraries required for OpenMp, random number generation and mathematical operations are included.

```cpp
#include <iostream>
#include <omp.h>
#include <cstdlib>
#include <cmath>
#include <random>
#include <chrono> // time-based seeding
```

2. **Main function**:

   The main function handles command-line arguments for the **number of threads** and **points** to be used in the calculation. After checking if the number of user inputs is correct (2), the input arguments are parsed to determine the specific numbers that will be used in the simulation and parallel computation. A **global counter** to keep track of the **points inside the unit circle** is initialized as well.

```cpp
int main(int argc, char *argv[]) {

  // Checking for problematic number of arguements
    if (argc != 3) {
```

```
5
6         std::cerr << "Usage: " << argv[0] << " <num_threads> <num_points
    >\n";
7         return 1;
8     }
9
10
11  // Conversions
12     int num_threads = std::atoi(argv[1]); // number of threads
13     long long num_points = std::atoll(argv[2]); // number of points to
    generate
14
15     omp_set_num_threads(num_threads); // setting number of threads to
    use in openmp
16
17     long long inside_circle = 0; // counter to keep track of points
    inside the circle
18                   // initialized to 0
```

3. **Parallel Region**:

   The computation is set to be performed in **parallel**. A uniform distribution (for numbers 0 to 1) is created, from which **random points** will be generated according to the thread's **unique seed**.

```
1  // Parallel Region
2  #pragma omp parallel
3      {
4          // Creating a unique seed using current time plus thread number
5          unsigned int seed = static_cast<unsigned int>(std::chrono::
    system_clock::now().time_since_epoch().count()) + omp_get_thread_num
    ();
6          long long local_inside_circle = 0; // local counter to keep
    track of points inside the circle
7          std::mt19937 generator(seed); // Initializin generator with the
    unique seed
8          std::uniform_real_distribution<double> distribution(0.0, 1.0);
    // creating uniform distribution from 0 to 1
```

4. **Monte Carlo Simulation**:

   Each thread generates random points from the distribution (coordinates x and y). If the sum of the squared coordinates of the current point is **less than or equal to 1** (equivalent of radius=1), the point is counted as being inside the circle and the local counter of points inside the circle is incremented. This process is performed until the desired number of points is reached. The important part is that the **iterations are distributed among the available threads**. After the loop ends, the results for points inside the circle are **added to the global counter**. For this part, an **atomic operation is used to avoid race conditions** (unpredictable results due to simultaneous updating of the same variable by

multiple threads).

```
#pragma omp for // distributing iterations among the available threads
        for (long long i = 0; i < num_points; i++) {

            double x = distribution(generator); // random x
            double y = distribution(generator); // random y

            if (x*x + y*y <= 1.0) { // if the radius is less than or equal to 1 (unit circle)

          local_inside_circle++; // incrementing local point counter
            }
        }
#pragma omp atomic // This operation is atomic to prevent race conditions
        inside_circle += local_inside_circle; // adding to global point counter
    }
```

5. **Pi Estimation**:

   Lastly, the **value of** π is estimated as the **ratio of points inside the circle to the total number of points multiplied by 4** (to account for four quadrants).

```
    double pi_estimate = (4.0 * inside_circle) / num_points; // pi estimation

    //  Printing the estimated value of Pi
    std::cout << pi_estimate << std::endl;

    return 0;
}
}
```

2.2 1st Notebook: Automation, Execution Time, Speedup

The purpose of the first Python code is to automate the execution of the C++ code and analyze the performance of the simulation. To achieve this, the C++ code is ran for different number of threads, as is detailed in this section.

1. **Libraries**:

   The necessary libraries are imported:

   - subprocess: used for calling and running the C++ executable file

- numpy: used when working with arrays
- matplotlib: used for visualization
- curve_fit from scipy_optimize: used for fitting itemtime: used to time the different executions of the program

```
# Importing libraries
import subprocess
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
import time
```

2. **Function to execute C++ Code**:

   First, a function to **automate** the execution of the C++ code is created, taking as argument the desired **number of threads** and **points**. Using "time", the execution time of the run is returned.

```
# Function to run the executable
def run_cpp_code(num_threads, num_points): # arguements: number of threads, number of points

    # command to run the C++ executable
    command = f'"C:\\Users\\             \\OpenMp\\monte_carlo_pi_2.exe" {num_threads} {num_points}'


    # Starting the timer
    start_time = time.time()

    # Checking for errors in execution
    try:
        # Running the command
        output = subprocess.check_output(command, shell=True, stderr=subprocess.STDOUT, universal_newlines=True)
    except subprocess.CalledProcessError as e:
        # Printing the error for debugging
        print(f"Command failed with exit status {e.returncode}")
        print(f"Error output:\n{e.output}")
        return None

    # Calculating the execution time
    execution_time = time.time() - start_time

    # Printing current execution information: threads and points used
    print(f"Running with {num_threads} threads and {num_points} points.")

    # Printing the output: pi estimation of current execution
    print(f"Pi estimate from C++ code:\n{output.strip()}")
```

```
29
30      return execution_time
```

3. **Running the code for different numbers of threads**:

   Our system has a 6-core processor and **12 logical processors** (threads). Thus, list of thread counts is defined, from 1 to 12. The function for executing the C++ code is called and **the code is executed for each count** and a **set number of points** ($10^8 here$). The estimations for the pi value are printed and the execution time is appended to a list.

```
1  # Defining parameters for the executions
2  num_points = 100000000 # number of points to be used in all executions
3  num_threads_list = [1, 2, 4, 6, 8, 10, 12]  # list of numbers of threads
       the C++ code will be executed with
4  execution_times = [] # list to store execution times
5
6  # Running the C++ code with different numbers of threads
7  for num_threads in num_threads_list:
8      exec_time = run_cpp_code(num_threads, num_points) # calling function
       that runs the executable
9      execution_times.append(exec_time) # appending time of current
      execution to list
```

4. **Plotting Execution time vs Number of Threads**:

   The **execution times** are plotted **against the number of threads**. Log scaled is used on the y-axis for better visibility

```
1  execution_times = np.array(execution_times)
2
3  # Plotting Execution time vs Number of Threads
4  plt.figure(figsize=(10, 5))
5  plt.plot(num_threads_list, execution_times, marker='o', color="purple")
6  plt.title('Execution Time vs Number of Threads')
7  plt.xlabel('Number of Threads')
8  plt.ylabel('Execution Time (seconds): log')
9  plt.xscale('linear')
10 plt.yscale('log')  # log scale for y-axis
11 plt.grid()
12 plt.show()
```

5. **Plotting Speedup vs Number of Threads**:

   Fist, **speedup** values are calculated using 1.2. The results are plotted against the Number of Threads.

```
1      # Calculating speedup
2  speedup = execution_times[0] / execution_times
3
4  # Plotting Speedup vs Number of Threads
```

```
plt.figure(figsize=(10, 5))
plt.plot(num_threads_list, speedup, marker='o', color="purple")
plt.title('Speedup vs Number of Threads')
plt.xlabel('Number of Threads')
plt.ylabel('Speedup')
plt.xscale('linear')
plt.yscale('linear')
plt.grid()
plt.show()
}
```

6. **Displaying plots in the same graph**:

   For neater visualization, both plots are displayed in the same graph. Log scale is used on the y-axis.

```
# Displaying both in the same graph
fig, ax1 = plt.subplots(figsize=(10, 5))

# Execution Time vs Number of Threads
line1 = ax1.plot(num_threads_list, execution_times, marker='o', color="purple", label='Execution Time')
ax1.set_xlabel('Number of Threads')
ax1.set_ylabel('Execution Time (seconds) [log scale]', color="purple")
ax1.set_xscale('linear')
ax1.set_yscale('log')
ax1.grid()

# Speedup vs Number of Threads
ax2 = ax1.twinx()
line2 = ax2.plot(num_threads_list, speedup, marker='o', color="green", label='Speedup')
ax2.set_ylabel('Speedup [log scale]', color="green")
ax2.set_xscale('linear')
ax2.set_yscale('log')


plt.title('Execution Time and Speedup vs Number of Threads')

# Legend
lines = line1 + line2
labels = [l.get_label() for l in lines]
ax1.legend(lines, labels, loc='center right', bbox_to_anchor=(0.95, 0.5))


plt.tight_layout()


plt.show()
```

7. **Amdahl's Law**:

   A function is defined to **fit** the speedup data to **Amdahl's Law**, according to 1.3. The **proportion of parallelizable code** is estimated using

curve_fit with an initial guess of 0.9. Then, using the definition of Amdahl's law function, we determine the **maximum speedup for 10000 threads**. A plot of the speedup values resulting from our code vs the theoretical speedup values from Amdahl's law is also created fro visualization purposes.

```
# Amdahl's Law function
def amdahls_law(x, p):
    return 1 / (1 - p + p / x)

# Initial guess for the proportion of parallelizable code
p_guess = 0.9

# Curve fitting to find the optimal value of p
p_opt, _ = curve_fit(amdahls_law, num_threads_list, speedup, p0=p_guess)

# Printing  results
print(f'Estimated proportion of parallelizable code (p): {p_opt[0]}')
print(f'Maximum speedup in the limit of 10000 threads: {amdahls_law
    (10000, p_opt[0])}')

# Generating theoretical speedup based on the fitted p value
speedup_theoretical = amdahls_law(num_threads_list, p_opt[0])

# Plotting the observed and theoretical speedup
plt.figure(figsize=(10, 6))
plt.plot(num_threads_list, speedup, 'o-', label='Observed Speedup',
    color='purple')
plt.plot(num_threads_list, speedup_theoretical, '--', label=f'Amdahl\'s
    Law (p = {p_opt[0]:.2f})', color='orange')

plt.yscale('log')
plt.title('Observed vs Theoretical Speedup (Amdahl\'s Law)')
plt.xlabel('Number of Threads')
plt.ylabel('Speedup (log scale)')
plt.grid(True)
plt.legend(loc='best')

plt.show()
```

8. **Extra: Fitting for up to 6 threads**:

   As an extra step, the previous process is repeated but this time for **only up to 6 threads** (which is equal to the cores of our processor). This was done to see how the values would change **without the benefit of hyperthreading**.

```
# Amdahl's Law fitting for up to 6 threads

p_guess2 = 0.9  # Initial guess for p
p_opt2, _ = curve_fit(amdahls_law, num_threads_list[0:4], speedup[0:4],
    p0=p_guess2)

```

```
print(f'Estimated proportion of parallelizable code (p): {p_opt2[0]}')
print(f'Maximum speedup in the limit of 10000 threads: {amdahls_law
    (10000, p_opt2[0])}')
```

2.3 2nd Notebook: Convergence Rate Analysis

In the second notebook, the convergence rate of the Monte Carlo method is analyzed. This is achieved by running the C++ code via the notebook for different numbers of randomly generated points.

1. **Libraries**:

   Same libraries as 1st notebook are imported

```
# Importing libraries
import subprocess
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
```

2. **Function to execute C++ Code for different numbers of points**:

   As in the previous notebook, a function to **automate** the execution of the C++ code is created. This time, the number of threads used is fixed (12) and the program is executed for **a range of point counts**, from $10^2 to 10^{11}$. The estimated value of pi is recorded and the **error** is calculated as the **absolute difference between the true value of pi and the estimated one**. It is then appended to a list.

```
# Function for running the C++ code for different number of points
def run_cpp_code(num_threads, num_points):
    command = f'"C:\\Users\\              \\OpenMp\\monte_carlo_pi_2.exe"
    {num_threads} {num_points}'

    # Running the command and capturing output
    output = subprocess.check_output(command, shell=True,
   universal_newlines=True)

    pi_estimate = float(output.strip())
    return pi_estimate

# True value of Pi
true_pi = np.pi

# Number of points to use for each run (from 10^2 to 10^11)
points_list = [10**i for i in range(2, 12)]
errors = []

```

```
# Running the C++ code for each number of points and calculating the error
for num_points in points_list:
    pi_estimate = run_cpp_code(num_threads=12, num_points=num_points)
    error = abs(pi_estimate - true_pi)
    errors.append(error)
    print(f"Points: {num_points}, Pi Estimate: {pi_estimate}, Error: {error}")
```

3. **Linear fit for convergence Rate**:

   A line is fitted to the log of the data values for the number of points and errors. This way, following the logic presented in 1.5, 1.6, the **convergence rate** can be determined through the **slope** of the line. The errors vs number of threads plot and the fitted line are showcased in the same graph.

```
# Fitting a line to the log-log plot to find the convergence rate
def linear_fit(x, a, b):
    return a * x + b

# Log-log plot of the error vs. number of points
plt.figure(figsize=(10, 6))
plt.loglog(points_array, errors_array, marker='o', linestyle='--', label='Simulation Errors')

# Taking the log of both axes
log_points = np.log10(points_array)
log_errors = np.log10(errors_array)

# Fitting a line to the log-log data
params, _ = curve_fit(linear_fit, log_points, log_errors)

# Getting the slope (convergence rate) and intercept
slope, intercept = params
print(f"Convergence rate: {slope}")

# Generating fitted line
fitted_log_errors = linear_fit(log_points, slope, intercept)
fitted_errors = 10**fitted_log_errors  # Exponentiate to get out of log scale

# Plotting the fitted line
plt.loglog(points_array, fitted_errors, label=f'Fitted Line (slope={slope:.2f})')


plt.title('Error in Pi Estimate vs. Number of Points with Fitted Line')
plt.xlabel('Number of Points (log scale)')
plt.ylabel('Error (log scale)')
plt.grid(True)
plt.legend()

plt.show()
```

4. **Theoretical Value Comparison**:

   Finally, the result from the fitting (value of the slope) is compared to the **theoretical** value for the convergence rate from 1.7.

```
    # Comparison with theoretical value
theoretical_slope = -0.5

print(f"Calculated Convergence Rate: {slope}")
print(f"Theoretical Convergence Rate: {theoretical_slope}")

# Compare the results
if np.isclose(slope, theoretical_slope, atol=0.1):
    print("The calculated slope is close to the theoretical expectation.
    ")
else:
    print("The calculated slope deviates from the theoretical
    expectation.")
```

Chapter 3

Results and Discussion

In this section, we showcase the results of our computational implementation.

3.1 Increasing number of threads

The results for executing the C++ code for a range of different thread counts are presented below. The number of random points generated was fixed at 10^8.

3.1.1 Pi value estimations

```
Running with 1 threads and 100000000 points.
Pi estimate from C++ code:
3.1416
Running with 2 threads and 100000000 points.
Pi estimate from C++ code:
3.1416
Running with 4 threads and 100000000 points.
Pi estimate from C++ code:
3.1417
Running with 6 threads and 100000000 points.
Pi estimate from C++ code:
3.14134
Running with 8 threads and 100000000 points.
Pi estimate from C++ code:
3.14159
Running with 10 threads and 100000000 points.
Pi estimate from C++ code:
3.1415
Running with 12 threads and 100000000 points.
Pi estimate from C++ code:
3.14176
```

Figure 3.1: Estimated values for Pi produced by Monte Carlo simulation for a range of different thread counts (1, 2, 4, 6, 8, 10, 12). Number of random points generates was set at 10^8

It can be concluded that the Monte Carlo method was employed successfully. For the number of points selected, the estimates for the pi value are very close to the exact value. This holds true for all executions, i.e. for all different thread counts.

3.1.2 Execution Time & Speedup vs Number of Threads

The following graph contains the plots for the Execution Time & Speedup vs Number of Threads.

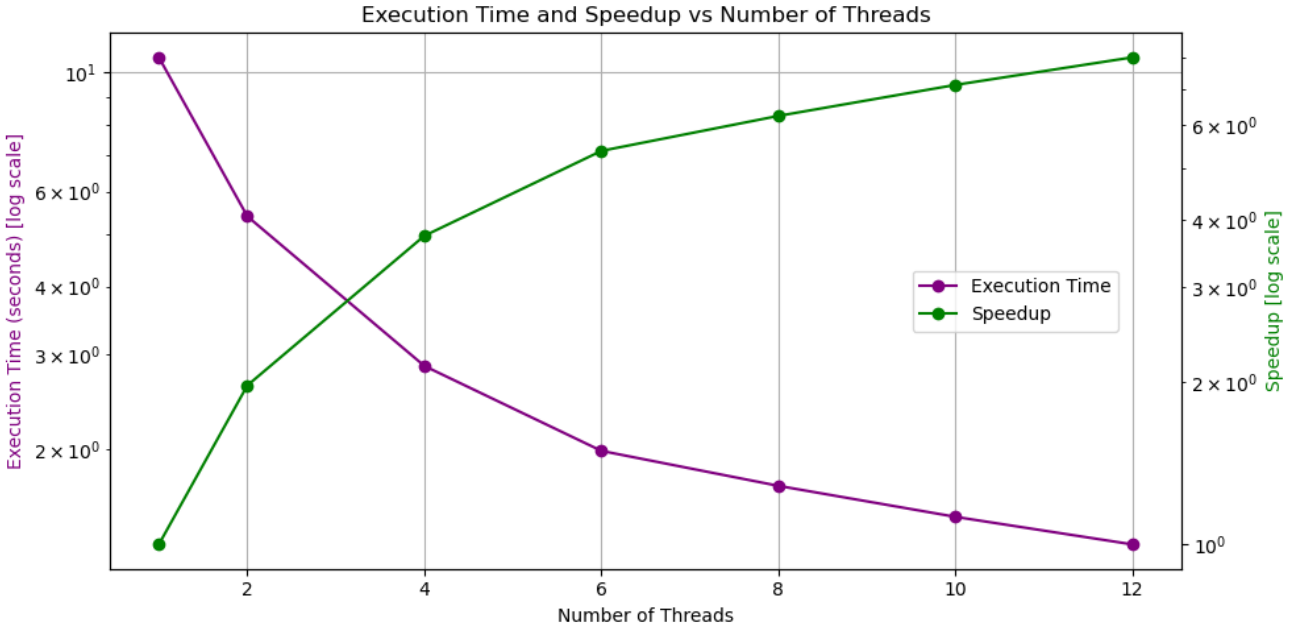

Figure 3.2: Exection time (purple) & Speedup (green) vs Number of Threads. Y-axis is in log scale

Observing the plots, two things are clear: **Execution time keeps decreasing with the increase of threads**, while speedup increases. Both signify **performance improvement** caused by the parallelization of our program. However, it should be noted that after some point (at 8 or 10 threads here) the improvement is so minimal that it potentially is not worth it for the specific problem. If more threads than needed are employed for the execution of a simple task, the result may be increase in **overhead** and slower execution.

3.1.3 Amdahl's Law

The graph below contains the **speedup** results derived from the simulation together with the **theoretical curve from Amdahl's Law** (1.3). Following that, the results for the **proportion p** of the code that **benefits from parallelization** and the **maximum possible speedup in the limit of 10000 cores**

are given for two cases: using up to **12 threads** (number of logical processors in our system) and using up to **6 threads** (number of cores of our processor).

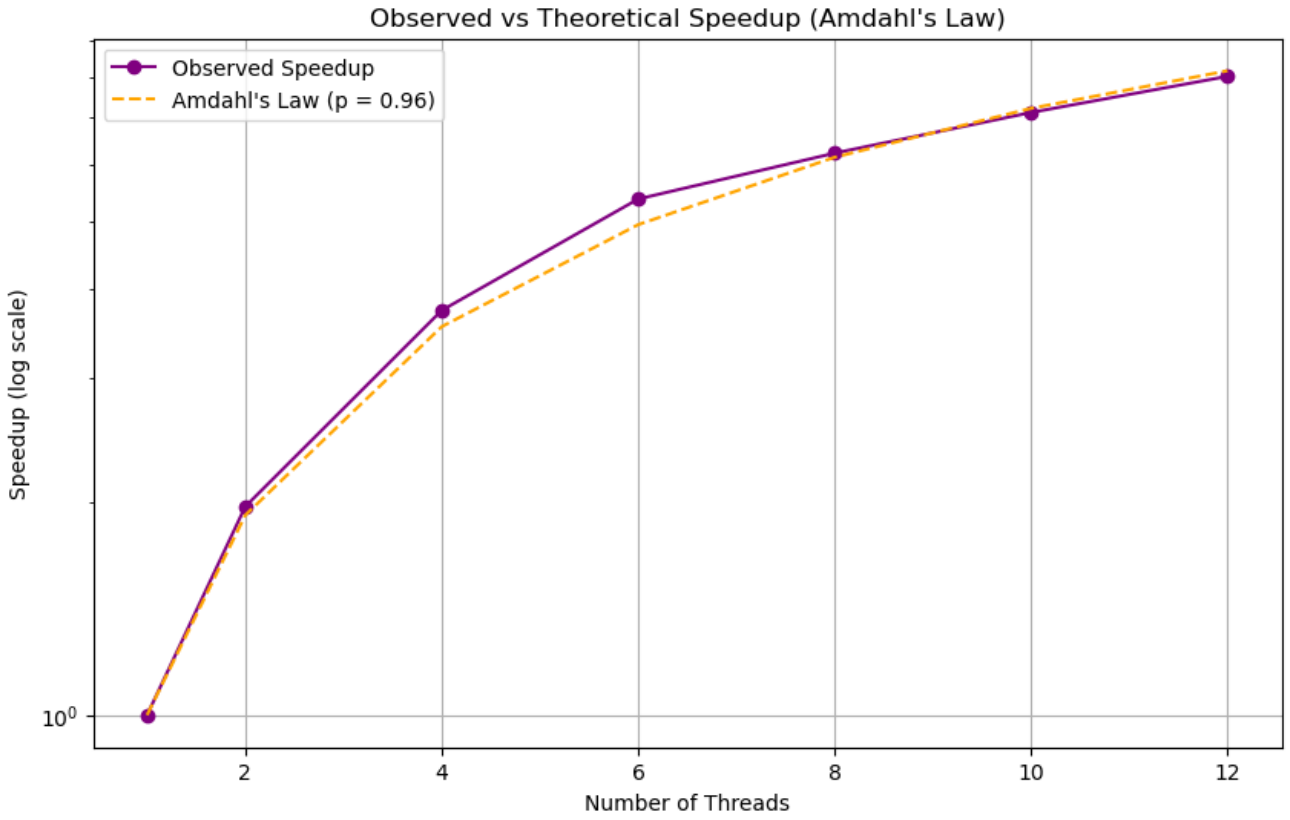

Figure 3.3: Simulation derived Speedup and fitted curve Speedup using Amdahl's law vs Number of Threads. Log scale on y-axis.

Results of fitting with up to 12 threads data:

```
Estimated proportion of parallelizable code (p): 0.9571884717082322
Maximum speedup in the limit of 10000 threads: 23.306086190099403
```

The estimated proportion of parallelizable code, **p=0.957**, means that 95.7% of the code can benefit from parallelization. The **maximum theoretical speedup** in the limit of 10,000 threads is estimated to be **23.3**. This indicates that even with a very large number of threads, the speedup ihas a limit due to the 4.3% non-parallelizable portion of the code.

Results of fitting with up to 6 threads data:

```
Estimated proportion of parallelizable code (p): 0.9766871795941889
Maximum speedup in the limit of 10000 threads: 42.7158952003537
```

Here, a sightly higher parallelizable portion is found, at **p=0.977**. Howeverm this leads to a significantly higher **theoretical speedup** of approximately **42.7** in the limit of 10,000 threads.

Comparison

The difference in p and maximum speedup between the two cases can be attributed to the fact that hyperthreading (when more threads than cores are used) introduces diminishing returns due to **overhead**. With up to 6 threads (equal to the physical cores in the system), the parallel execution is found to be more efficient. Once we go beyond the core count (12 threads), the performance gains slow down, and the overhead from managing additional threads reduces the proportion of effectively parallelized code, lowering the potential speedup. This is why the **6-thread fit predicts a higher theoretical speedup compared to the 12-thread fit: the overhead from using more threads than cores ends up reducing the efficiency of parallelization**!

3.2 Increasing number of points

The results produced by executing the C++ code for different numbers of randomly generated points are demonstrated here. The number of threads was fixed at 12 (equal to the number of logical processors available).

3.2.1 Convergence Rate

The convergence rate of the Monte Carlo method was calculated by fitting a line to the log of data for the error and the number of points. The plot of the data with the fitted line in log scale is shown below, followed by the result for the convergence rate, which corresponds to the slope of the line.

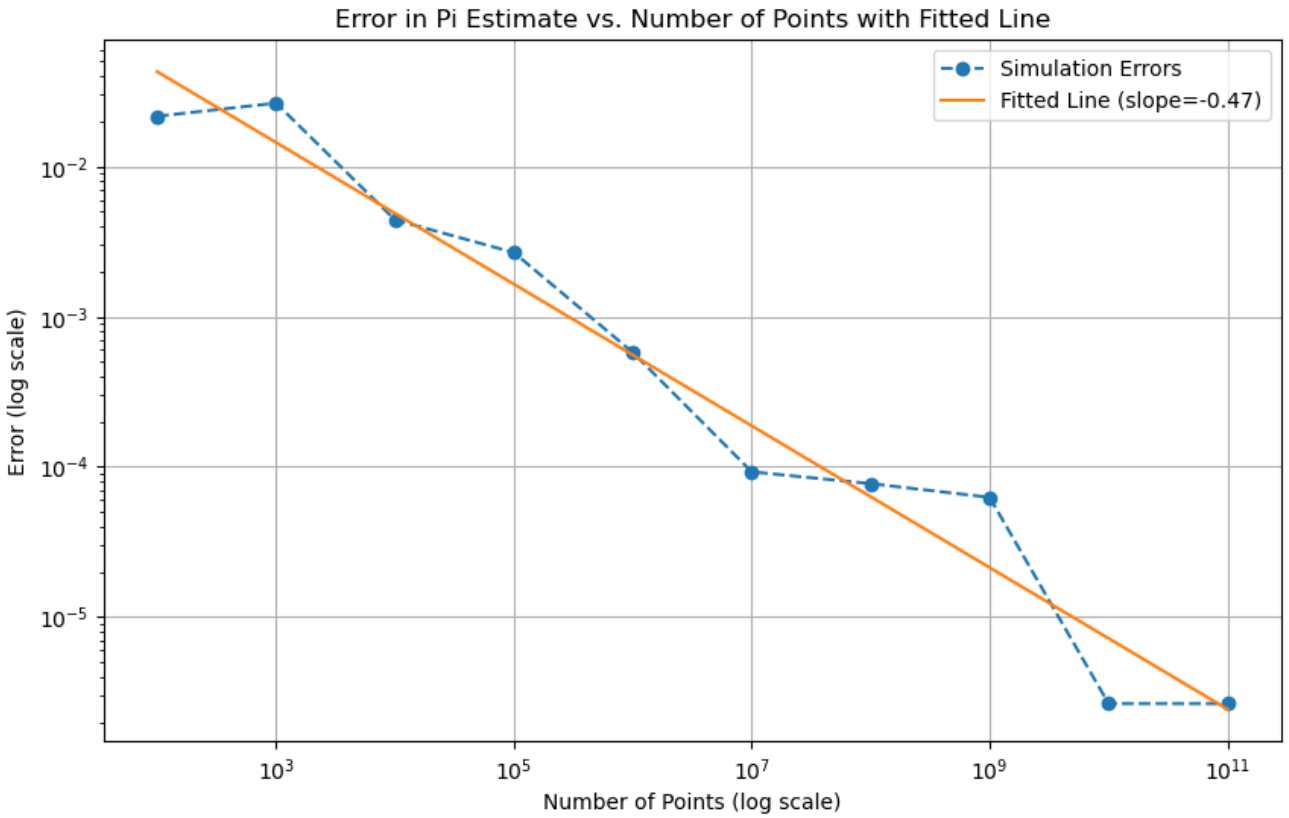

Figure 3.4: log-log plot of the Error in estimations from the Monte Carlo Simulations vs the Number of Points (blue) together with fitted line (orange)

```
Calculated Convergence Rate: -0.47167105923229524
Theoretical Convergence Rate: -0.5
The calculated slope is close to the theoretical expectation.
```

The slope of the fitted line, which corresponds to the **convergence rate** of the method is about **-0.4717**, which is close to the theoretical value of **-0.5** proposed in 1.7. This serves as a confirmation that the implementation of the Monte Carlo method is **working as expected** and it provides a reliable approximation of pi with the **accuracy improving slowly but steadily with the increase in points** (large increase in number of points is needed to see significant gains in accuracy)

Chapter 4

Appendix

C++ code for parallelized Monte Carlo simulation

Jupyter Notebook for running C++ code for increasing thread counts

Jupyter Notebook for running C++ code for increasing number of points