Set 2 Programming Tools

Aristotle University of Thessaloniki

Martha Papadopoulou

August 2024

1 Introduction

This project aims to estimate the value of π using the Monte Carlo method by implementing a code in C/C++ with OpenMP. The assignment is divided into two sections. In the first section, the code will be executed with a fixed number of random points while varying the number of threads. The execution times will be recorded and plotted in a Jupyter notebook to analyze the speedup relative to the number of cores. Additionally, Amdahl's law will be applied to fit the resulting curve and to assess the proportion of the code that benefits from parallelization. In the second section, the convergence rate of the Monte Carlo method will be explored by running the code with all available threads while varying the number of random points. The results will be compared to the theoretical expectations of the Monte Carlo method.

2 Theoretical Overview

2.1 Monte Carlo method for π calculation

The implementation of the Monte Carlo method to calculate the value of π is relatively straightforward. The calculations occur within a unit square (1x1). Random coordinates for x and y are generated, and if a point satisfies the condition $x^2 + y^2 < 1$, it is considered to be inside the circle. The ratio of the number of points within the circle to the total number of points provides an estimate for $\pi/4$, given that the coordinates are positive (ref. [2]). Consequently, the value of π is calculated using the following formula:

$$\pi = 4 \cdot \frac{points\ inside\ the\ circle}{total\ number\ of\ points} \tag{1}$$

The convergence rate describes how quickly the estimate approaches the true value as the number of random samples increases. The estimate of π improves as the number of samples grows. It is known that the theoretical convergence rate of the Monte Carlo method is proportional to $1/\sqrt{N}$, meaning that the error decreases with the square root of the number of samples (ref. [2]). This factor is easily transformed into a logarithmic scale:

$$log\left(\frac{1}{\sqrt{N}}\right) = log(1) - \frac{1}{2}log(N) = -\frac{1}{2}log(N) \tag{2}$$

As evident the slope of the theoretical convergence of the Monte Carlo method is -0.5.

2.2 Speedup and Amdahl's law

In parallel programming, speedup quantifies how much faster a program runs when executed on multiple processors compared to a single processor. It is calculated as the ratio of the time the code took to run without parallelization to the time it took with parallelization.

$$speedup = \frac{time\ without\ parallelization}{time\ with\ parallelization} \tag{3}$$

Amdahl's law explains the potential speedup of a task when parts of it are parallelized. It shows that the overall performance improvement is limited by the portion of the task that can't be parallelized. This means that no matter how many processors are added, the speedup will always be restricted by the portion of the code that runs serially. The mathematical formula of Amdahl's law is the following.

$$S_{latency}(s) = \frac{1}{(1-p) + \frac{p}{s}} \tag{4}$$

Where, $S_{latency}$ is the theoretical speedup of the entire task, s is the speedup factor of the portion of the code that can be improved with system resources and p is a part of the execution time that can benefit from system resources (ref. [1]). Since the task is divided among N threads, we can say that the s factor is equal to N. As the number of processors increases, the speedup approaches a maximum value, which is constrained by the non-parallelizable portion of code.

3 Code Implementation

3.1 Monte Carlo without parallelization

Before parallelizing the code, we develop a basic implementation, which is the 'Monte_Carlo_pi.cpp' file. After initializing the necessary parameters and setting a seed for the random number generator using the time, we set the number of random points to 10^9. The code then generates random x and y coordinates for these points, ranging between 0 and 1, incrementing the circle count by 1 if the points lie within the circle. The value of π is then calculated using equation (1).

3.2 Parallelized Monte Carlo with varying number of threads

This code is implemented in the 'Monte_Carlo_pi_parallel.cpp' file. The logic remains the same as in the previous code, with a few additional features. To store the program's results, two text files are created, 'pi_data.txt' and 'time_data.txt'. The outer loop is introduced to set the number of threads to be used. Within this loop, the parallel portion of the program is executed, parallelizing the inner loop that processes the random points. This is accomplished using the '#pragma omp parallel for' line, where shared, private, and default variables are declared. Additionally, a reduction clause is applied to the circle points to prevent any incorrect accumulations. For each thread, the execution time is recorded, and the value of π is calculated.

These results are then processed in a Jupyter notebook, which reads the text files and loads the data into two lists 'pi_data' and 'time_data'. Subsequently, the execution time of each run is plotted against the number of threads used. We also plot the speedup achieved through parallelization against the number of threads used. To calculate the speedup, equation (3) is applied for each run.

Lastly, a function based on Amdahl's law is defined by equation (4) , and using the 'curve_fit' function from the 'scipy.optimize' Python library, the proportion of the code that benefits from parallelization p can be determined. After p is calculated, the speedup data are plotted alongside Amdahl's law to illustrate the correlation between them. Additionally, using equation (4), we can calculate the maximum possible speedup for 10,000 cores.

3.3 Parallelized Monte Carlo with varying number of points

The last C++ code is found in the 'Monte_Carlo_pi_points.cpp' file. In this version, the outer loop varies the maximum number of points, rather than the number of threads. The implementation is similar to the last one, with a few differences. The number of threads is set to the maximum available, which is 8 in this case. The program is executed with point counts ranging from 10^2 to 10^9, and for each run, the value of π is calculated. These values are accumulated in the 'pi_data_b.txt' file for further processing.

To study the convergence rate of our calculation, the 'pi_data_b' file is loaded into a new Jupyter notebook, where the convergence error for each value is calculated and plotted. To find the linear regression, we use the 'polyfit' function from 'numpy' and then plot the line on a log-log graph alongside the data points.

4 Code Results

The first code, which does not use parallelization, takes a few minutes to return a value of 3.141520432.

The following code, with parallelization and varying threads, produces the following results:

- For 1 thread: computation time is $104.18\,s$ and π value is 3.141520608.
- For 2 threads: computation time is $53.11\,s$ and π value is 3.141474576.
- For 3 threads: computation time is $38.61\,s$ and π value is 3.141572464.
- For 4 threads: computation time is $33.34\,s$ and π value is 3.141479228.
- For 5 threads: computation time is $30.14\,s$ and π value is 3.14150666.
- For 6 threads: computation time is $28.33\,s$ and π value is 3.14148966.
- For 7 threads: computation time is $23.51\,s$ and π value is 3.141507768.
- For 8 threads: computation time is $21.76\,s$ and π value is 3.141514768.

From the Jupyter notebook, we obtain the plot of execution time relative to the number of threads used, which is displayed in Figure 1.

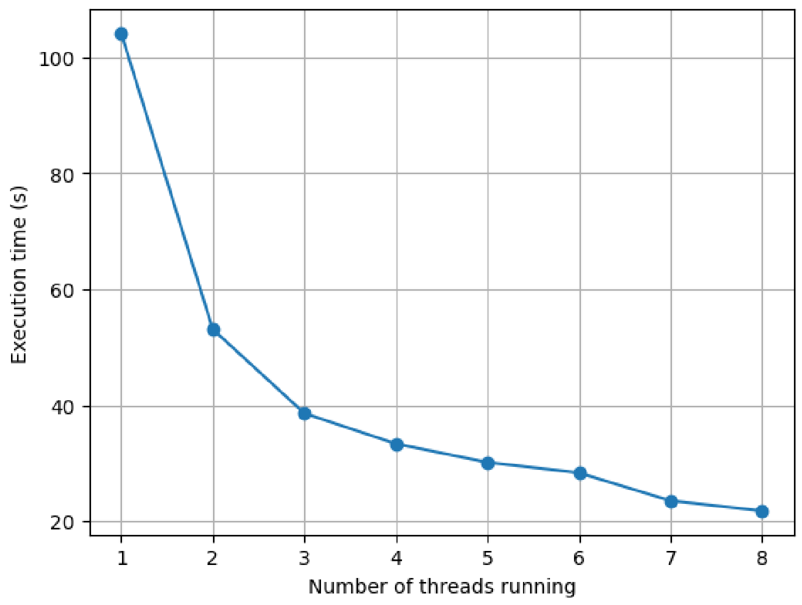

Figure 1: Plot of execution time of code versus the number of threads used for the computation. The execution time is in seconds.

After calculating the speedup and fitting Amdahl's law to the curve, we determine that $p = 0.8984$. With that value of p, the plot of Amdahl's law fit is drawn alongside the data points, as depicted in Figure 2.

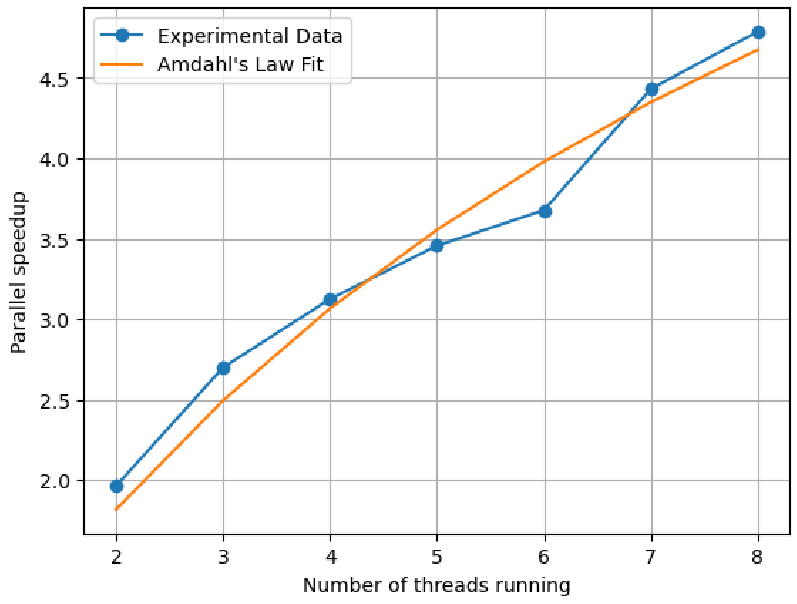

Figure 2: Plot of speedup versus the number of threads used for each run (blue line) and the Amdahl's law curve fit for the data (orange line).

It is also calculated that the maximum possible speedup with a limit of 10,000 cores is 9.8357.

For the last code, which involves parallelization and varying random points, the results of the π calculations are as follows:

- For 10^2 points: π value is 3.36
- For 10^3 points: π value is 3.104
- For 10^4 points: π value is 3.0736
- For 10^5 points: π value is 3.11936
- For 10^6 points: π value is 3.13576
- For 10^7 points: π value is 3.1408576
- For 10^8 points: π value is 3.14094576
- For 10^9 points: π value is 3.141462608

These results are then transferred into the Jupyter notebook, where the errors are computed. Using these errors to assess the convergence, the linear regression is calculated and plotted, as shown in Figure 3. The fitted linear regression line is given by $y = -0.45x + 0.31$.

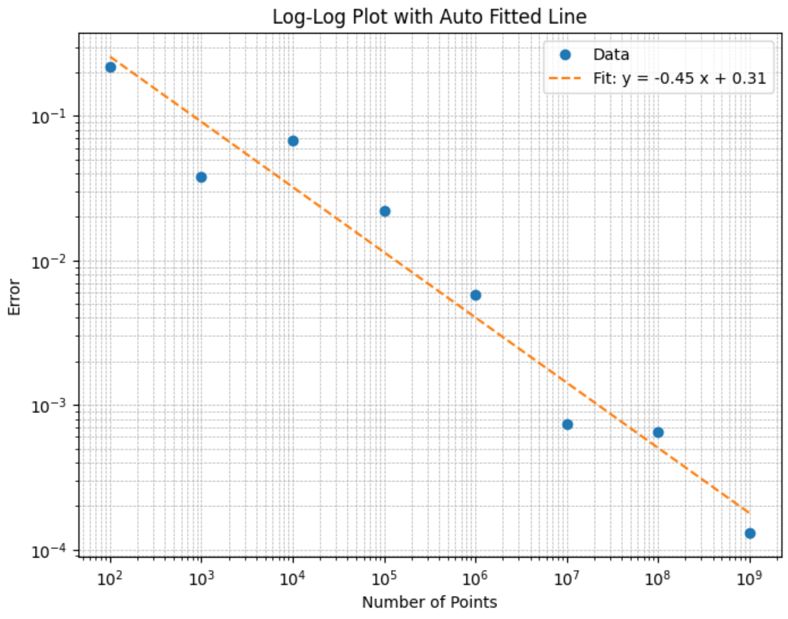

Figure 3: Plot of the convergence error of the calculated π values (blue line) and the fitted linear regression (orange line).

5 Discussion

The first code, which does not use parallelization, is straightforward and requires little explanation. Its purpose is mainly to demonstrate the difference in execution time between running the code with and without parallelization. The code successfully produces an estimated value within a few minutes of running.

For the Monte Carlo code with parallelization and varying threads, the results are mostly in line with theoretical expectations, but there are a few anomalies. As shown in Figure 1, the execution time decreases with each additional thread. The curve drops significantly for the first 4 threads, but the reduction becomes less pronounced with each additional thread. This behavior is expected because the program runs on a 4-core, 8-thread PC, meaning the improvement in performance is most effective up to the number of physical cores, after which the optimization should decline sharply. However, as seen in Figure 1, there's an unexpected drop in execution time at 7 threads, which is slight but causes issues later on in the speedup calculation.

In the speedup calculation, shown in Figure 2 with the blue line, the issue mentioned above becomes more prominent. While the speedup starts to increase more slowly for 5 and 6 threads, it unexpectedly rises sharply again for the 7 and 8-thread runs. According to Amdahl's Law, the fitting curve should ideally align up to 4 threads, as the speedup is expected to increase more gradually beyond that point. However, due to the significant increase observed up to 8 threads, all data points were included in the fitting process. The fitted curve is displayed in Figure 2, shown in orange.

From that fitting, we find that the portion of the code that benefits from parallelization p is 0.8984. This indicates that 89.84% of the code can be optimized through parallelization, while the remaining portion cannot be parallelized and is executed serially. This is a relatively high percentage, suggesting that most of the code is well-suited for parallel computing. Although this allows for a potentially high maximum speedup, in practice, the speedup is unlikely to reach the maximum theoretical value due to various other factors that may arise during computation.

With this high value of p, if we had 10,000 cores running, the maximum theoretical latency speedup would be 9.8357. This speedup is significantly higher than those shown in Figure 2, but it is only slightly lower than the maximum theoretical speedup of 'infinite' number of cores $1/(1-p)$, which is approximately 9.8425. This confirms that adding an excessive number of cores will yield very little additional benefit, as the speedup does not increase proportionally with the rising number of cores. Moreover, the scenario of using 10,000 cores is not cost-effective, as the resources required would not justify the minimal performance improvement.

For the final code, which implements Monte Carlo parallelization with a varying number of random points, we calculate the error for each run and fit a linear regression line to the data, as shown in Figure 3. The resulting linear regression line is $y = -0.45x + 0.31$. According to theory, we expect a slope of -0.5, and while the actual slope is close to the theoretical value, it deviates slightly. This discrepancy may occur because we included data points from runs with very few random points, such as 10^2 and 10^3. The accuracy might have improved had we added more runs with larger numbers of points, such as 10^{10} and 10^{11}, so that the range of N covers both small and large scales. Furthermore, since the Monte Carlo method relies on randomness, it is normal to have deviations from the expected values.

In the parallelized codes, there was an issue with computing π values using more than 10^9 points. The computed values were significantly inaccurate. This issue is likely due to limitations of the PC, such as numerical precision. It is suspected that the problem arises from precision errors associated with declaring variables as 'double', which may not be accurate enough for such high values of N.

6 Conclusion

In summary, the study effectively demonstrated the impact of parallelization on the Monte Carlo method for estimating π. The initial code without parallelization served as a baseline, successfully producing results within minutes. The parallelized version, which varied the number of threads, generally aligned with theoretical expectations. However, some anomalies were observed, such as an unexpected drop in execution time with 7 threads, which affected the speedup calculations. The fitting of Amdahl's Law revealed that 89.84% of the code benefits from parallelization, indicating a high potential for optimization. For 10,000 cores running, the theoretical latency speedup was 9.8357. It was demonstrated that adding more cores would not provide additional efficiency.

Furthermore, analysis of the final code, which varied the number of random points, showed that while the slope of the linear regression line was close to the theoretical value, minor deviations were present. These discrepancies were likely due to including runs with very few random points and the inherent randomness of the Monte Carlo method. Additionally, computing π values with more than 10^9 points led to significant inaccuracies, likely due to precision limits of the PC.

References

[1] Wikipedia contributors. Amdahl's law — Wikipedia, the free encyclopedia, 2024. [Online; accessed 24-August-2024].

[2] Wikipedia contributors. Monte carlo method — Wikipedia, the free encyclopedia, 2024. [Online; accessed 24-August-2024].