Programming Tools
OpenMP: Project 2

Aspasia Pallikaridou [1]

Aristotle University of Thessaloniki
School of Physics
MSc Computational Physics
Professor: N. Stergioulas

[1] apallika@physics.auth.gr

Contents

Chapter 1

Introduction

The aim of this assignment is to estimate the value of π using a Monte Carlo method and parallel programming techniques. The Monte Carlo method offers a numerical approach for integration by randomly sampling points and assessing how many of those points fall inside a defined geometric shape. In this case, we are approximating π by evaluating the area of a quarter circle inscribed in a unit square. To calculate π, the following relationship between the area of the quarter circle and the area of the square is used:

$$\pi = \frac{4E}{r^2} \tag{1.1}$$

where E represents the area estimated using Monte Carlo integration, and r is the radius of the circle. For simplicity, we set $r = 1$.

In addition to implementing this calculation, multi-threading programming was introduced to speed up the Monte Carlo method, as the algorithm inherently involves a large number of independent computations. The parallelized version of the program was developed using OpenMP. The speedup of the program was evaluated and fitted with a curve based on Amdahl's law, which predicts the theoretical maximum speedup achievable depending on the proportion of the code that can be parallelized. Furthermore, a convergence study was conducted by fitting a line to the log-log plot of the relative error in estimating π versus the number of points used, highlighting the relationship between the number of points and the accuracy of the Monte Carlo method.

The program was run on the Aristotelis cluster of the Aristotle University of Thessaloniki, which has 64 available threads, allowing significant performance improvements compering to the 8 threads that a common personal computer usually has.

Chapter 2

Methodology

2.1 Monte Carlo

The Monte Carlo technique that was used is basically Monte Carlo integration. It is a numerical technique for integration using arbitrary numbers. It randomly chooses points near the area in which the integral is evaluated.[1]

Concerning the particular integral 1.1 with r=1, it represents a 2D quarter circle 'A' that has for sides x=0, y=0 and $x^2 + y^2 \leq 1$. The hole shape 'A' is included in a 2D square with sides equal to 1 and area equal to 1. Therefore, the integral's value can be approached as the percentage of the arbitrary points that are inside the shape 'A' over those that belong to the square.

The steps that were followed during the computation are:

- Drawing N random numbers (x_i, y_i by using the function "numpy.random.random()". This function draws random float numbers in the interval [0, 1]. As a results, the points could be anywhere inside the square.
- Counting how many points are inside the shape 'A'. In order to achieve this, the following if condition was applied: if $(x^2 + y^2 \leq 1)$
- Calculating the integral as: I=1 A/N

2.2 Parallel Programming

The term parallel programming is used to describe the process of decomposing one problem into smaller tasks that can be executed simultaneously using multiple compute resources. The different tasks can be run at the same time by using multiple cores within a CPU. As a result, parallel programming is very important for large scale projects in which speed is critical. There are three models of parallel programming:

- Message - Passing programming
- Shared - Memory programming
- Accelerators (GPUs)

This report's implementation uses the shared - memory programming model which is the simplest of the three.

The most common and popular environment for shared memomry programming is the OpenMP (Open MultiProcessing). It consists many tools and functions and can be used through C/C++ and FORTRAN. OpenMP enables the use of more than one cores that the computer's CPU have, in the more overloaded points of the program. These points are called parallel regions.[3]

During the estimation of π, the parallel programming was used in the Monte Carlo computations. The N points were split to the threads, and finally, if they belonged to the quarter circle, had to be added to the A region. It is worth to notice that all the threads try to add to the A (*sum*) at the same time. In order to avoid confusions and errors, the sum had to be handled carefully, and be characterized as restricted.

The C++ program of this exercise includes implementations for up to sixty four threads, which were available from Aristotle University of Thessaloniki cluster Aristotelis. An easy way to change the number of threads in OpenMP is by the function: omp_set_num_threads(...);.

Additionally, speedup of the code due to the multiple threads was calculated, plotted, and fitted with the theoretical curve of the Amdahl's law. According to Amdahl's law, "the overall performance improvement gained by optimizing a single part of a system is limited by the fraction of time that the improved part is actually used" [2]:

$$S_{latency} = \frac{1}{(1-p) + \frac{p}{s}} \tag{2.1}$$

where $S_{latency}$ is the theoretical speedup of the execution of the whole task, s is the speedup of the part of the task that benefits from improved system resources, and is equal to the number of threads, and p is the proportion of execution time that can benefit from improved resources originally occupied. The speedup has been estimated in practice as:

$$S_{pr} = \frac{time\,with\,one\,thread}{time\,with\,multi\,threads} \tag{2.2}$$

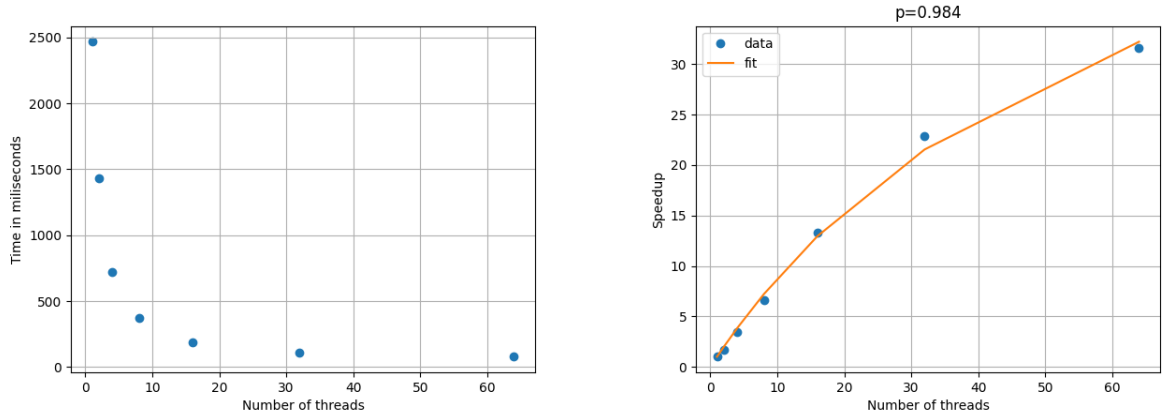
The only unknown value that has been left is the parameter p. By using scipy's curve_fit in Pyhton, p can be easily estimated correctly so that Amdahl's law properly fits the data.

On the last part of the particular exercise, the code was arranged properly so that to calculate pi using the maximum available number of threads (64), and a variety of number of points from 10^2 to 10^{11}. The target of this part is to calculate the convergence rate of the Monte Carlo implementation. A simple methodology to achieve it is to fit a line in a log-log- plot of the relative error in calculating pi vs. the number of points. The absolute value of the line's slope would be equal to the asked convergence rate. Monte Carlo's convergence rate is usually equal to 0.5 O(N$\sim$1/2).[4]

Chapter 3

Results

The number of cores that Aristotelis has is 32 (64 threads). Likely, it is big enough to observe significant speedups. As it can be seen in Figure 3.1a, the duration of the compilation without the multi-threading technique was about 2500 ms, for 10^7 points. Each time a thread was added, the duration became about the half of the previous, and finally, at 64 threads, became about 100 ms! The plot 3.1b depicts the speedup (data) of the program as calculated from the equation 2.2 and the fit of Amdahl's law. The proportion of execution time that can be parallelized (p) was computed as a parameter of the fit, and calculated equal to 0.984 (98,4%).



(a) (b)

Figure 3.1: Left: The duration of the compilation vs the number of threads. Each time a thread was added, the duration became about the half of the previous. Right: speedup (data) and the fit using Amdahl's law vs the number of threads. The proportion of execution time that can be parallelized (p) calculated equal to 0.984 (98,4%).

In the second part of the exercise, a line was fit in the log-log- plot of the relative error in calculating pi vs. the number of points, in order to study its slope. The line that fitted the data was the following:

$$LineFit = -0.45x + 0.75$$

as it can be seen in Figure 3.2. The slope was -0.45. Therefore, the asked convergence rate was:

$$ConvergenceRate = 0.45$$

which is truly about 0.5.

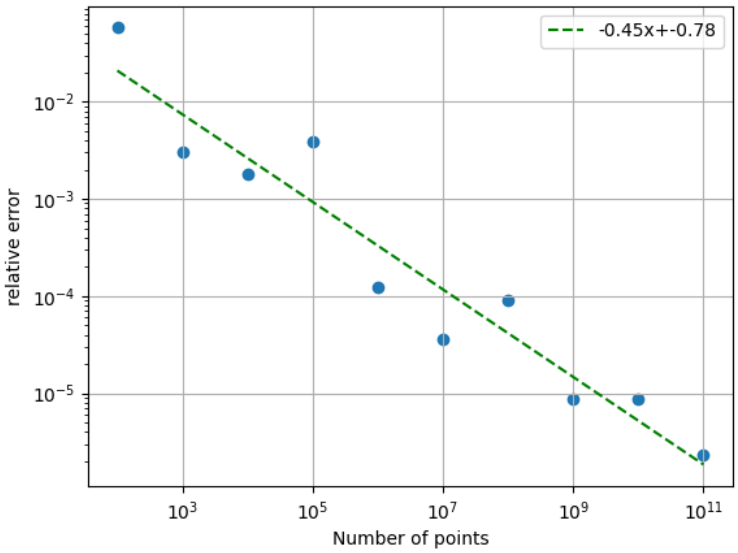


Figure 3.2: Fitting a line in the log-log- plot of the relative error in calculating π vs. the number of points. The convergence rate is the absolute value of the slope, 0.45

Chapter 4

Discussion

During the investigation of multi-threaded programming for estimating π using the Monte Carlo method, several key issues and observations arose:

- Random Number Generation Issue: Initially, the srand function with the current time as the seed was used to generate random points. However, the execution time did not decrease as expected when more threads were added. The issue was that multiple threads were attempting to use the same random number generator, leading to contention and inefficiency. To resolve this, it became necessary for each thread to have its own independent random number generator, allowing them to work autonomously and avoid conflicts.
- Comparison Between Aristotelis and Personal Computer[5] (64 vs 8 threads): Two different systems were used to compare performance. The Aristotelis cluster, which has 64 threads, was much faster and more efficient than the common computer, whose CPU supports only 8 threads. The results on the Aristotelis cluster were highly satisfactory, with a significant reduction in execution time, especially for large numbers of points (from 10^9 to 10^{11}). In contrast, the performance on the personal computer was much slower. Additionally, the proportion of execution time that could be parallelized on the personal machine was smaller—approximately 75%—compared to 98.4% on Aristotelis. This discrepancy underscores the importance of thread count in parallel computations.

Chapter 5

Conclusion

In this project, the Monte Carlo method was used to estimate π, and parallel programming with OpenMP was applied to speed up the calculations. Running the program on the Aristotelis cluster with 64 threads significantly reduced the computation time. The speedup followed Amdahl's law, showing that 98.4% of the code could be parallelized.

The convergence rate of the Monte Carlo method was found to be 0.45, close to the expected value of 0.5, confirming the method's accuracy. A key challenge was ensuring each thread had its own random number generator, which improved performance. Comparing results between a personal computer and the Aristotelis cluster showed that having more threads led to much better efficiency.

Bibliography

[1] Monte Carlo Integration

[2] Amdahl's law

[3] Introduction to Parallel programming, tutorial by N. Trifonidis

[4] Monte Carlo and quasi-Monte Carlo methods Russel E. Caflisch

[5] Github Repository