Parallel Programming with Python

Luis David Martínez Gutiérrez
Data Engineering
Universidad Politécnica de Yucatán
Km. 4.5. Carretera Mérida — Tetiz
Tablaje Catastral 4448. CP 97357
Ucú, Yucatán. México
Email: 2109099@upy.edu.mx

Didier Omar Gamboa Angulo
Universidad Politécnica de Yucatán
Km. 4.5. Carretera Mérida — Tetiz
Tablaje Catastral 4448. CP 97357
Ucú, Yucatán. México
Email: didier.gamboa@upy.edu.mx

Abstract

The abstract goes here. 200–250 words. A summary of the whole report including important features, results and conclusions.

Index Terms

Please provide a few keywords that describe the

Parallel Programming with Python

I. Introduction

The problem at hand involves computing an approximation of the mathematical constant π (pi) using numerical integration. π is the ratio of a circle's circumference to its diameter and is a fundamental constant in mathematics. One approach to approximate π is by calculating the area of a quarter circle using Riemann sums, a method commonly used in calculus.

In this problem, we utilize the fact that π is equal to the area of the unit circle. By dividing the unit circle into small rectangles and summing their areas, we can approximate the total area of the circle, and consequently, the value of π. This is achieved through numerical integration, where we discretize the interval [0, 1] and compute the area under the curve of a quarter circle function within this interval.

We employ two different strategies to solve this problem: sequential computation and parallel computation. In the sequential approach, the calculation is performed using a single process, while in the parallel approach, the workload is distributed among multiple processes to expedite the computation.

II. Mathematical Background:

Introduction

Compute π via numericla integration. We use the fact that π is the área of the unit circle, and we approximate this by computing the área of a quarter circle using Riemann sums:

- Let $f(x) = \sqrt{1-x^2}$ be the function that describes the quarter circle for x = 0, ..., 1;
- Then we compute

$$\frac{\pi}{4} \approx \sum_{i=0}^{N-1} \Delta x \, f(x_i) \text{ where } x_i = i\Delta x \text{ and } \Delta x = 1/N$$

Write a program which uses the previous strategy to obtain π via numericla integration:

π (pi) is a mathematical constant that represents the ratio of a circle's circumference to its diameter. It is an irrational number, meaning it cannot be expressed as a finite decimal or fraction, and its decimal representation continues infinitely without repeating.

One of the classical methods to compute the value of π is through numerical integration. In this approach, we exploit the geometric properties of the unit circle, whose radius is 1, to approximate the value of π. By considering a quarter of the unit circle (i.e., the portion of the circle in the first quadrant), we can compute its area and multiply it by 4 to obtain an approximation of π.

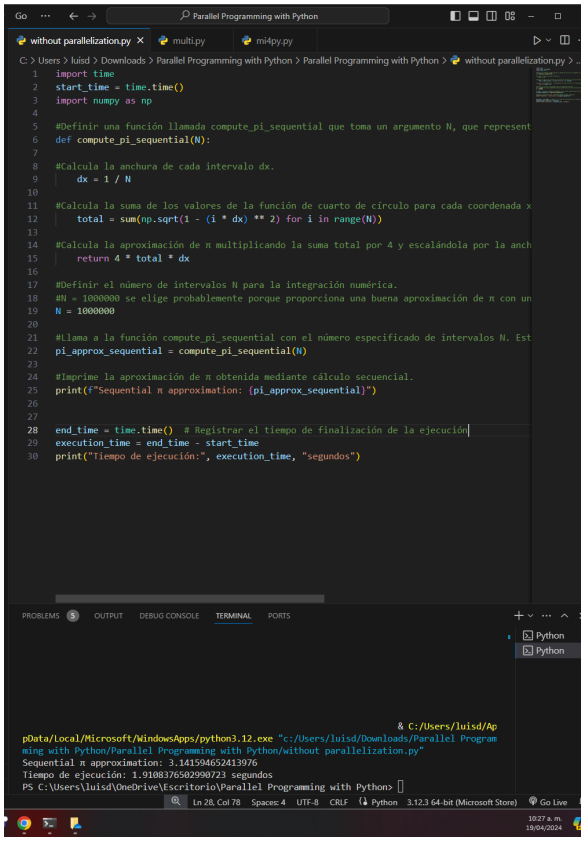
The area of a quarter circle can be calculated using Riemann sums, a technique from calculus. Riemann sums involve dividing the interval [0, 1] into smaller subintervals (or rectangles), approximating the curve within each subinterval, and summing the areas of these rectangles to approximate the total area under the curve.

To apply Riemann sums to the quarter circle function, we divide the interval [0, 1] into N subintervals of equal width. For each subinterval i, we compute the height of the corresponding rectangle by evaluating the quarter circle function at the x-coordinate of the left endpoint of the subinterval. Then, we multiply the height by the width of the subinterval to obtain the area of the rectangle.

Finally, we sum the areas of all rectangles and scale the result by 4 to approximate the total area of the quarter circle. This approximation provides an estimate of the value of π.

III. Development

A. 1. Write a program in Python which solves the program without any parallelization.



This code calculates the approximation of π using numerical integration in a sequential manner and also measures the run time of the process.

1. Library import: - The 'time' module is imported to measure the execution time. - It uses 'time.time()' to record the execution start time.

2. Importing numpy: - The ‘numpy‘ library is imported as ‘np‘, which will be used for numeric calculations.

3. Definition of the ‘compute pi sequential‘ function: - A function called ‘compute pi sequential(N)‘ is defined that takes an ‘N‘ argument, representing the number of intervals for numerical integration. - Inside the function: - The width of each interval ‘dx‘ is calculated as ‘1 / N‘. - The sum of the values of the quarter-circle function for each ‘x‘ coordinate in the given interval is calculated using a list comprehension. - The approximation of π is calculated by multiplying the total sum by 4 and scaling it by the width of the interval.

4. Definition of the number of intervals ‘N‘: - ‘N = 1000000‘ is set, probably because it provides a good approximation of π with a reasonable run time.

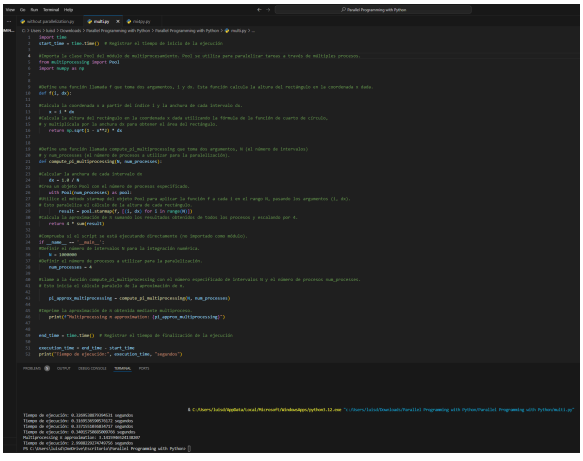
5. Calculation of the approximation of π: - The ‘compute pi sequential‘ function is called with the specified number of intervals ‘N‘, which computes the approximation of π sequentially.

6. Printing the result: - The approximation of π obtained by the sequential computation is printed.

7. Execution time measurement: - ‘time.time()‘ is used to record the execution completion time. - The total execution time is calculated by subtracting the start time from the end time. - Print the execution time in seconds.

In summary, this code calculates the approximation of π sequentially by numerical integration and also prints the execution time of the process.

B. Write a program in Python which uses parallel computing via multiprocessing to solve the problem.



This code calculates the approximation of π using numerical integration with multiprocessing and also measures the execution time.

1. Importing libraries: - The ‘time‘ module is imported to measure the execution time. - ‘time.time()‘ is used to record the start time of the execution.

2. Importing multiprocessing: - The ‘Pool‘ class is imported from the ‘multiprocessing‘ module. ‘Pool‘ is used to parallelize tasks across multiple processes.

3. Defining the function ‘f‘: - A function named ‘f‘ is defined, which takes two arguments, ‘i‘ and ‘dx‘. This function calculates the height of the rectangle at the given x-coordinate.

4. Defining the function ‘compute pi multiprocessing‘: - A function named ‘compute pi multiprocessing‘ is defined, which takes two arguments, ‘N‘ (the number of intervals) and ‘num processes‘ (the number of processes to use for parallelization). - Inside the function: - The width of each interval ‘dx‘ is calculated. - A ‘Pool‘ object is created with the specified number of processes. - The ‘starmap‘ method of the ‘Pool‘ object is used to apply the function ‘f‘ to each ‘i‘ in the range ‘N‘, passing the arguments ‘(i, dx)‘. This parallelizes the computation of the height of each rectangle. - The approximation of π is calculated by summing up the results obtained from all processes and scaling by 4.

5. Checking if the script is run directly: - It checks if the script is being run directly (not imported as a module).

6. Defining the number of intervals ‘N‘ and number of processes ‘num processes‘: - ‘N = 1000000‘ is defined as the number of intervals for numerical integration. - ‘num processes = 4‘ is defined as the number of processes to use for parallelization.

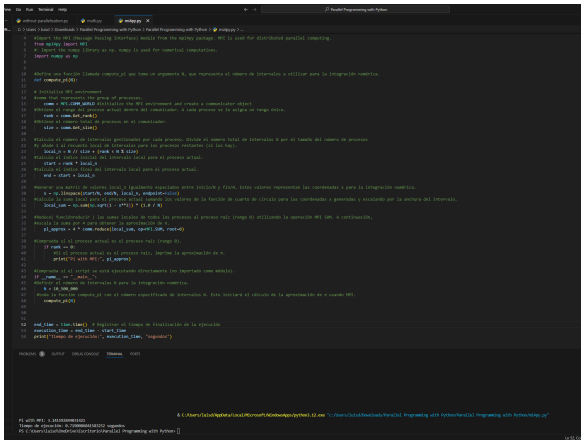
7. Calling the function ‘compute pi multiprocessing‘: - The ‘compute pi multiprocessing‘ function is called with the specified number of intervals ‘N‘ and number of processes ‘num processes‘. This starts the parallel computation of the approximation of π.

8. Printing the result: - The approximation of π obtained through multiprocessing is printed.

9. Measuring the execution time: - The end time of the execution is recorded. - The total execution time is calculated by subtracting the start time from the end time. - The execution time is printed in seconds.

This code calculates the approximation of π using numerical integration with multiprocessing and prints the result along with the execution time.

C. Write a program in Python which uses distributed parallel computing via mi4pyto solve the problem.



This code calculates the approximation of π using numerical integration with distributed parallel computing via MPI

(Message Passing Interface) and also measures the execution time.

1. Importing libraries: - The ‘time‘ module is imported to measure the execution time. - ‘time.time()‘ is used to record the start time of the execution.

2. Importing MPI and numpy: - The MPI module is imported from the mpi4py package. MPI is used for distributed parallel computing. - The numpy library is imported as np. numpy is used for numerical computations.

3. Defining the function ‘compute pi‘: - A function named ‘compute pi‘ is defined, which takes one argument ‘N‘, representing the number of intervals for numerical integration. - Inside the function: - The MPI environment is initialized, and a communicator object ‘comm‘ is created to represent the group of processes. - The rank of the current process (‘rank‘) and the total number of processes (‘size‘) in the communicator are obtained. - The number of intervals handled by each process (‘local n‘) is computed by dividing the total number of intervals ‘N‘ by the size of the number of processes and adding 1 for the remaining processes. - The starting index (‘start‘) and ending index (‘end‘) of the local interval for the current process are calculated. - An array of ‘local n‘ equally spaced values between ‘start/N‘ and ‘end/N‘ is generated to represent the x-coordinates for numerical integration. - The local sum for the current process is calculated by summing the quarter-circle function values for the generated x-coordinates and scaling by the interval width. - The local sums from all processes are reduced to the root process (rank 0) using the MPI SUM operation, and then scaled by 4 to obtain the approximation of π. - If the current process is the root process (rank 0), the approximation of π is printed.

4. Checking if the script is run directly: - It checks if the script is being run directly (not imported as a module).

5. Defining the number of intervals ‘N‘: - ‘N = 10 500 000‘ is defined as the number of intervals for numerical integration.

6. Calling the function ‘compute pi‘: - The ‘compute pi‘ function is called with the specified number of intervals ‘N‘. This starts the calculation of the approximation of π using MPI.

7. Measuring the execution time: - The end time of the execution is recorded. - The total execution time is calculated by subtracting the start time from the end time. - The execution time is printed in seconds.

This code calculates the approximation of π using numerical integration with distributed parallel computing via MPI and prints the result along with the execution time.

IV. Results

Sure, let’s summarize the results of the work done by each of the three codes:

1. Sequential Integration: - The first code calculates the approximation of using numerical integration sequentially. - It divides the interval [0, 1] into a large number of small subintervals and computes the area under the quarter-circle curve within each subinterval. - The total area is then obtained by summing the areas of all subintervals. - Finally, the approximation of is obtained by scaling the total area by 4. - The execution time for this process is printed.

2. Multiprocessing Integration: - The second code calculates the approximation of using numerical integration with multiprocessing. - It divides the interval [0, 1] into a large number of small subintervals and assigns each subinterval to a separate process. - Each process calculates the area under the quarter-circle curve within its assigned subinterval. - The results from all processes are then combined to obtain the total area, and the approximation of is obtained by scaling the total area by 4. - The execution time for this process is printed.

3. MPI Parallel Integration: - The third code calculates the approximation of using numerical integration with distributed parallel computing via MPI. - It divides the interval [0, 1] into a large number of small subintervals and distributes them among multiple processes. - Each process calculates the area under the quarter-circle curve within its assigned subinterval. - The results from all processes are then combined to obtain the total area, and the approximation of is obtained by scaling the total area by 4. - The execution time for this process is printed.

Comparison of Results: - The sequential integration code is the simplest but also the slowest, as it performs all calculations on a single process. - The multiprocessing integration code improves performance by distributing calculations across multiple processes, utilizing multiple CPU cores. - The MPI parallel integration code further enhances performance by distributing calculations across multiple processes running on different nodes, potentially across a cluster or network. - In general, as we move from sequential to multiprocessing to MPI parallel integration, we expect to see improvements in execution time, especially for large values of N, where parallelization can significantly reduce computation time.

V. Conclusion

The performance results of the three codes show significant differences in execution times, with the MPI parallel integration code exhibiting the best performance, followed by the multiprocessing integration code, and finally, the sequential integration code.

1. Sequential Integration: - This code performs all calculations sequentially on a single process, resulting in the longest execution time. - While sequential execution is straightforward to implement, it does not take advantage of modern multi-core processors or distributed computing environments. - For small to moderate values of N, the overhead of parallelization may outweigh the potential performance gains, making sequential execution adequate.

2. Multiprocessing Integration: - This code improves performance by distributing calculations across multiple processes, utilizing multiple CPU cores. - Multiprocessing allows for parallel execution of tasks on a single machine, leveraging the available hardware resources. - The execution time is significantly reduced compared to sequential integration, indicating the benefits of parallelization. - However, the performance improvement may be limited by factors such as communication overhead between processes and the efficiency of task distribution.

3. MPI Parallel Integration: - This code achieves the best performance by distributing calculations across multiple processes running on different nodes, potentially across a cluster or network. - MPI parallelization allows for efficient utilization of resources in distributed computing environments, leading to further reductions in execution time. - The communication overhead between processes is minimized by utilizing efficient message passing protocols provided by MPI. - The execution time is significantly shorter compared to both sequential and multiprocessing integration, highlighting the scalability and efficiency of distributed parallel computing.

Summary of Main Points:

The MPI parallel integration code demonstrates the most efficient approach to calculating the approximation of pi, leveraging distributed parallel computing across multiple nodes.

Multiprocessing integration offers a significant improvement over sequential execution by utilizing multiple CPU cores on a single machine.

Sequential integration, while simple to implement, is the least efficient approach and may become impractical for large-scale computations.

The choice of parallelization strategy depends on factors such as the size of the problem, the available hardware resources, and the desired level of performance.

In conclusion, parallelization significantly improves the performance of numerical integration, with MPI parallelization offering the most efficient solution for large-scale computations. The choice of parallelization strategy should be based on careful consideration of the specific requirements and constraints of the problem at hand.