Approximating π through Numerical Integration using Riemann Sums

Héctor Emiliano González Martínez
Universidad Politénica de Yucatán
Data Engineering
2109077@upy.edu.mx

***Abstract*—This report explores the computation of π by approximating the area of a unit circle using numerical integration. Three Python-based implementations are discussed: a single-threaded approach, a multiprocessing approach, and a distributed computing approach using MPI.**

I. Introduction

The mathematical constant π (pi) is a fundamental element in various fields of science and engineering, particularly in calculations involving circles and spherical geometries. Traditionally, π can be computed using various techniques ranging from geometrical constructions to infinite series. However, these methods, while precise, often require advanced mathematical tools or are not easily scalable with basic computational approaches.

A practical approach to approximate π is through numerical integration, leveraging the relationship between π and the area of a circle. By focusing on a quarter-circle, we can simplify the problem to computing the integral of a function that represents this geometric figure. Specifically, we define $f(x) = \sqrt{1 - x^2}$, which describes the top half of a quarter-circle embedded in the unit square where x ranges from 0 to 1. The integral of $f(x)$ over this interval directly relates to $\pi/4$.

To approximate this integral, we employ the Riemann sums method, a fundamental concept in numerical analysis. This method involves dividing the area under $f(x)$ into a finite number of small intervals and summing up the areas of rectangles under the curve at these intervals. Mathematically, this is expressed as:

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

where $x_i = i\Delta x$ and $\Delta x = 1/N$ represents the width of each interval.

This project aims to implement and compare three computational strategies to approximate π using the described method: 1. **Single-threaded approach**: A straightforward implementation using a Python script that computes the sum sequentially. 2. **Multi-processing approach**: An enhancement of the single-threaded approach that utilizes multiple CPU cores to divide and conquer the problem, potentially decreasing the computation time significantly. 3. **Distributed computing approach**: Using the ‘mpi4py‘ library, this approach distributes the computation across multiple nodes in a computing cluster, aiming for even greater efficiency and scalability.

Each method will be critically analyzed to assess its performance, accuracy, and scalability. The implementations will be tested on different hardware setups, including a laptop and a desktop PC, to evaluate how different computing environments affect their performance. These results will inform conclusions about the suitability of each approach for different scales of computational resources, leading to insights on how numerical integration techniques like Riemann sums can be optimized in practical applications.

II. Solutions

A. Non-Parallel Approach

The script *pure.py* employs NumPy to calculate π without parallelization, focusing on a straightforward numerical integration method. This approach is based on the Riemann sum, which is essential for approximating the area under curves in numerical analysis.

1) Code Overview: The function *approximate_pi* in *pure.py* performs the numerical integration using the following key steps, which directly correspond to the mathematical formulation for computing π as described in the introduction.

2) Defining the Function: First, we define the function that represents the upper half of the quarter-circle. This is crucial as the accuracy of π depends on the precise calculation of this curve's area.

```
def f(x):
    return np.sqrt(1 - x**2)
```

This function, $f(x) = \sqrt{1 - x^2}$, describes the curve of the quarter-circle within the unit square, which we integrate to approximate $\pi/4$.

3) Computing the Riemann Sum: The core of the script is the computation of the Riemann sum, where we discretize the interval [0, 1] and sum up the areas of rectangles under the curve.

```
# Calculate the step size for the integration
dx = 1.0 / N
# Generate an array of x values from 0 to 1 - dx,
    spaced evenly into N intervals
x = np.linspace(0, 1 - dx, N)
# Compute the function values at each x
f_x = np.sqrt(1 - x**2)
# Compute the Riemann sum
area_quarter_circle = np.sum(f_x * dx)
```

In these lines, dx represents the width of each rectangle under the curve, x is a NumPy array containing the x-coordinates at which $f(x)$ is evaluated, and f_x stores the heights of the rectangles. The Riemann sum np.sum$(f_x \cdot dx)$ multiplies the heights of these rectangles by their width to approximate the integral of $f(x)$ over the interval.

4) Scaling to Approximate π: Finally, since the integral approximates $\pi/4$, we scale the result by 4 to estimate π.

```
pi_approx = 4 * area_quarter_circle
```

This multiplication adjusts the approximation from the quarter-circle's area to the full circle, effectively estimating the value of π.

5) Execution: The script includes a condition to ensure that it runs the approximation only when executed as the main program, setting $N = 10,000,000$ for a detailed approximation:

```
if __name__ == "__main__":
    N = 10000000
    pi_approximated = approximate_pi(N)
    print("Approximated Pi:", pi_approximated)
```

This block sets up the number of intervals (N), calls the approximation function, and prints the result. The large value of N ensures a finer discretization, which enhances the accuracy of the approximation.

This non-parallel method, while simple and straightforward, sets a baseline for performance and accuracy, against which we will compare more complex parallel computing techniques in the following sections.

B. Multiprocessing Approach

The *multi.py* script enhances the computation of π by employing Python's *multiprocessing* module, which allows parallel execution across multiple CPU cores. This method aims to reduce the computational time significantly by dividing the task into smaller chunks that can be processed concurrently.

1) Function Definition: The approach begins by defining the function that models the upper half of a circle's equation, crucial for the numerical integration.

```
def f(x):
    return np.sqrt(1 - x^2)
```

This function, $f(x) = \sqrt{1 - x^2}$, computes the y-values of the circle at given x-values. These y-values represent the heights of rectangles used in the Riemann sum computation.

2) Integral Calculation: The script defines a function to calculate the integral over a specified range using the rectangle method. This function computes the sum of rectangle areas under the curve from 'start' to 'end' with a width of 'dx'.

```
def integrate(start, end, dx):
    x = np.arange(start, end, dx)
    y = f(x)
    return np.sum(y * dx)
```

The 'integrate' function generates an array of x-values, calculates the corresponding y-values using $f(x)$, and then sums up the products of y-values and the differential dx, thus estimating the area under $f(x)$ over the given interval.

3) Parallel Computation: The core of this approach is the parallel computation setup, where the integration task is split across multiple processes. Each process handles a portion of the overall interval.

```
def parallel_pi(N, num_processes):
    dx = 1.0 / N
    chunk_size = N // num_processes
    ranges = [(i * chunk_size * dx, (i + 1) *
    chunk_size * dx, dx) for i in range(
    num_processes)]
    with Pool(num_processes) as pool:
        results = pool.starmap(integrate, ranges)
    total_area = sum(results)
    return 4 * total_area
```

This function calculates the step size 'dx', determines the size of each chunk of the interval, and creates a list of tuples specifying the start and end points for each chunk. The 'Pool' object from the 'multiprocessing' module is used to distribute these tasks to different processes, which execute the 'integrate' function concurrently. The results from all processes are summed to obtain the total area, which is then scaled by 4 to approximate π.

4) Execution: The script ensures it executes the parallel computation when run as the main program, setting a high number of intervals for increased accuracy.

```
if __name__ == "__main__":
    N = 100000000
    num_processes = 4
    pi_approximated = parallel_pi(N, num_processes)
    print("Approximated Pi using multiprocessing:",
    pi_approximated)
```

Here, N is set to 100 million to refine the approximation granularity, and the number of processes is specified as 4. This setup allows the script to leverage multicore processors effectively, demonstrating the significant potential for performance enhancement in numerical computations through parallel processing.

This multiprocessing method not only speeds up the computation compared to the single-threaded approach but also serves as a practical example of how concurrent computing can be applied in numerical analysis to handle computationally intensive tasks more efficiently.

C. MPI-based Distributed Approach

The *mpi_script.py* leverages the *mpi4py* library to implement a distributed computing solution for approximating π. This method uses multiple nodes in a computing cluster, allowing for a significant speed-up by parallelizing the computation across different machines connected via a network.

1) Function Definition: The core mathematical function, representing the curve of a quarter-circle, is defined with an emphasis on computational efficiency by using the float32 data type. This data type strikes a balance between precision and memory usage, suitable for distributed computing environments.

```
def f(x):
    return np.sqrt(1 - x^2, dtype=np.float32)
```

This function $f(x) = \sqrt{1 - x^2}$ calculates the height of the curve at given x-values, crucial for determining the area under the curve using Riemann sums.

2) Distributed Numerical Integration: The script defines a function to perform numerical integration over a specific interval assigned to each process. This distributed approach optimizes the usage of computational resources by splitting the interval into smaller segments.

```
def integrate(start, end, dx):
    num_points = int((end - start) / dx)
    x = np.linspace(start, end, num_points, endpoint
    =False, dtype=np.float32)
    y = f(x)
    return np.sum(y) * dx
```

Here, each process calculates its segment of the integral by generating x-values within its assigned range and computing the corresponding y-values. The sum of the products of y-values and the differential dx approximates the area under the curve for that segment.

3) Parallel Execution Setup: The MPI setup involves initializing communication, determining each process's role, and distributing the integration tasks among the available processes.

```
def mpi_pi(N):
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()

    dx = 1.0 / N
    local_n = N // size + (1 if rank < N % size else
     0)
    local_start = rank * (N // size) * dx + min(rank
    , N % size) * dx
    local_end = local_start + local_n * dx

    local_sum = integrate(local_start, local_end, dx
    )
    total_sum = comm.reduce(local_sum, op=MPI.SUM,
    root=0)

    if rank == 0:
        pi_approx = 4 * total_sum
        print("Approximated Pi using MPI:",
    pi_approx)
```

This segment of the code outlines how each node calculates a portion of the total integral and then uses MPI's reduce function to sum all partial results to the root process. The root process then multiplies the total by 4 to approximate π.

4) Execution and Scalability: The script is structured to run the main function only when executed directly, which is essential in a distributed computing environment to avoid unintended multiple initializations.

```
if __name__ == '__main__':
    N = 10000000
    mpi_pi(N)
```

Setting $N = 10,000,000$ ensures a fine granularity for the approximation, demonstrating the script's capability to handle large-scale computations efficiently. This MPI-based approach is particularly effective for large datasets and high-resolution computations, offering scalability and performance that are crucial for advanced scientific computing tasks.

This section demonstrates how MPI can be utilized to distribute and parallelize heavy computational tasks across multiple computing nodes, thus significantly enhancing performance and scalability for large-scale numerical integration tasks aimed at approximating π.

III. Profiling

A. Testing Environment

Two different computing environments were utilized to profile the scripts for computing π. Their specifications are as follows:

1) Laptop Specifications:

- **Processor:** 11th Gen Intel®Core™i5-1135G7 @ 2.40GHz (8 threads)
- **Graphics:** Mesa Intel®Xe Graphics (TGL GT2)
- **Memory:** 12.0 GiB
- **Operating System:** Ubuntu 22.04.4 LTS

2) Desktop PC Specifications:

- **Processor:** Intel®Core™i7-14700KF (28 threads)
- **Graphics:** NVIDIA RTX 4060 8GB
- **Memory:** 32.0 GiB
- **Operating System:** Ubuntu 22.04.4 LTS

B. Execution Time Comparison

Laptop:

Method	Total CPU Time	Wall Time	Result
pure.py	1.79 s	1.8 s	3.141593
multi.py	22.1 ms	1.73 s	3.141593
mpi_script.py	N/A	662 ms	3.141593

Desktop:

Method	Total CPU Time	Wall Time	Result
pure.py	718 ms	718 ms	3.141593
multi.py	14.5 ms	258 ms	3.141593
mpi_script.py	N/A	472 ms	3.141593

IV. Conclusions

This study has methodically examined the computational efficiency of various methods for approximating π, using both single-threaded and parallel computing techniques. Our analysis, performed on two different hardware setups, has uncovered several important findings:

- **Impact of Parallel Computing:** The implementation of multiprocessing and distributed computing through MPI generally improves performance over the single-threaded approach. However, the extent of this enhancement varies. While MPI often achieves notable reductions in execution times by utilizing multiple computing nodes to process larger segments of computation in parallel, its effectiveness can depend significantly on the hardware used and the specific computational context.
- **Hardware Utilization:** The advanced capabilities of the desktop PC with a high-performance processor and more memory led to markedly improved execution times across

all methods when compared to the laptop. This underscores the importance of robust hardware to maximize the benefits of parallel computational techniques.

- **Efficiency of Multiprocessing:** While the multiprocessing approach does not scale as aggressively as MPI, it still presents a considerable advantage over the non-parallel approach, especially noticeable in environments with multiple cores but limited by single-machine constraints.
- **Scalability of Distributed Computing:** The MPI approach not only improved performance on a single machine but also demonstrated significant scalability potential. This suggests that for computationally intensive tasks, expanding to more nodes in a cluster could yield even more substantial performance gains.

In conclusion, the adoption of parallel computing strategies has been validated as a highly effective means of improving the performance of complex numerical tasks, with clear implications for fields requiring high precision and computational intensity.