Parallel Programming with Python

Priscila Guadalupe Tzuc Alonzo
Data Engineering
Universidad Politécnica de Yucatán
Ucú, Yucatán, México
2109148@upy.edu.mx

References

***Abstract*—The approximation of π, an essential mathematical constant critical in various fields such as geometry and physics, remains a significant subject for computational algorithms. This study explores the approximation of π by calculating the area of a unit circle through the Riemann sum method, a foundational concept in calculus. The method approximates π by summing the areas of numerous small rectangles under the curve described by $f(x) = \sqrt{1-x^2}$. By incrementally refining the number of rectangles, the approximation of π becomes more precise. This approach not only reinforces understanding of numerical integration but also serves as a platform for examining computational efficiency through sequential and parallel processing techniques.**

Introduction

The pursuit of computing π with high precision has historically tested the limits of numerical methods and computational capabilities. This project leverages the Riemann sum approach, where the area under the curve of a quarter circle, represented by the function $f(x) = \sqrt{1-x^2}$, is integrated from 0 to 1 to approximate $\pi/4$. This approach simplifies the complex calculation of π to a manageable task of summing rectangles' areas under a curve. To enhance computational efficiency, the study implements this method in three computational strategies: a non-parallelized solution, parallel computing using Python's multiprocessing, and distributed computing using MPI with the mpi4py library. Each method scales differently with increasing N, impacting both the accuracy and the computational speed, providing a unique perspective on the trade-offs between computational accuracy and performance.

I. Numerical Method

For the specific problem, the function $f(x) = \sqrt{1-x^2}$ is used to describe the arc of a quarter circle, since the function is derived from the Pythagorean theorem where the radius of the circle is 1. Integrating this function over the interval from 0 to 1 yields the area under the curve, which is a quarter of the total area of the circle. The integral $\int_0^1 \sqrt{1-x^2}\,dx$ is equivalent to $\pi/4$; therefore, Riemann sums are employed, which is a basic numerical method for estimating definite integrals by summing areas of multiple small rectangles under the curve.

The area of each rectangle is determined as $f(x_i) \times \Delta x$, where Δx is the width of each rectangle and $f(x_i)$ is the height of our function. To refine the approximation:

- We increase N, the number of rectangles, thus reducing the width $\Delta x = \frac{1}{N}$.
- The total sum $\sum_{i=0}^{N-1} f(x_i)\Delta x \approx \frac{\pi}{4}$, which can be scaled by 4 to estimate π.

II. Computational Aspect

Three computational strategies are used to solve this problem:

Non-parallelized Python Solution

Most Relevant Part of the Code

The core of the computation lies within the loop where each rectangle's area is computed. This section is responsible for the accuracy and efficiency of the π approximation:

```
for i in range(N):
    x_i = i * delta_x
    f_x_i = math.sqrt(1 - x_i**2)  # f(x) = sqrt(1 - x^2)
    sum_area += f_x_i * delta_x
```

This loop is crucial as it sequentially calculates the area of each rectangle, essential for the numerical integration process used to approximate π.

Profiling: Execution Time Evaluation

To evaluate how the execution time varies with different values of N, we can measure the time taken for each execution by running the function `calculate_pi(N)` with increasing values of N. Below is the complete script and its result at runtime.

```
import math
import time

def calculate_pi(N):
    # Width of each small rectangle
    delta_x = 1.0 / N
    sum_area = 0.0

    # Calculate area using the sum of rectangles
    for i in range(N):
        x_i = i * delta_x
        f_x_i = math.sqrt(1 - x_i**2)  # f(x) = sqrt
(1 - x^2)
        sum_area += f_x_i * delta_x

    # The total area under the curve is an
    approximation for pi/4, so multiply by 4
    pi_approx = sum_area * 4
    return pi_approx
```

```
# Number of rectangles
N = 1000000

# Start timing
start_time = time.time()

pi_value = calculate_pi(N)

# Stop timing
end_time = time.time()

# Calculate execution time
execution_time = end_time - start_time

print(f"Approximation of    with N={N} {pi_value}")
print(f"Execution time: {execution_time} seconds")
```

Visualization of Results

```
Approximation of π with N=100000000 3.1415926735892157
Execution time: 34.828962564468384 seconds
```

Fig. 1.

This code is focused to implement a simple loop in Python to sum the Riemann sums sequentially, but it has its limitations, as N increases, the computation becomes more demanding, causing significant slowdowns due to the sequential nature of the processing.

A. Parallel Computing Using Multiprocessing

The code is designed to approximate the value of π using the numerical method of integrating the area of a quarter circle. It does so by using multiple processes to speed up the calculation, a method well-suited for multi-core systems.

Key Functions and Variables:

- **calculate_area(x)**:
  - **Purpose**: This function computes the area of a rectangle at a specific position x within the quarter circle. The rectangle's height is determined by the circle's equation $\sqrt{1-x^2}$, and its width is δ_x, which is the inverse of the total number of rectangles N.
  - **Relevant Code**:

```
        return math.sqrt(1 - x**2) * delta_x
```

    The multiplication of the square root expression by δ_x gives the area of one rectangle. This function is critical because it defines how the area under the curve is calculated, which is the core of the integration process.
- **parallel_pi_calculation(N, num_processes)**:
  - **Purpose**: This function orchestrates the parallel computation of π. It divides the task of calculating the area of each rectangle among multiple processes.
  - **Key Components**:
    - **Global delta_x**: Defines the width of each rectangle, crucial for scaling the rectangle's area calculation correctly.
    - **x_values List Creation**:

```
        x_values = [i * delta_x for i
    in range(N)]
```

    This list comprehension generates x-coordinates at which the rectangle heights are evaluated, ranging from 0 to just less than 1, spaced by δ_x.
    - **Process Pool Creation**:

```
        with Pool(processes=
    num_processes) as pool:
            areas = pool.map(
    calculate_area, x_values)
```

    Here, a pool of worker processes is created. The 'pool.map' function applies 'calculate_area' to each item in 'x_values', distributing these tasks across multiple processes. This parallel execution is the most critical aspect of this solution as it leverages multiprocessing to reduce total computation time.
    - **Area Summation and Multiplication**:

```
        return 4 * sum(areas)
```

    After collecting the areas calculated by the different processes, they are summed and multiplied by 4 to get the approximate value of π, reflecting the quarter circle's full area.

Execution and Output

The script sets the number of subdivisions (N) and the number of processes. It then calculates π using the defined function and prints the result. The choice of N and $num\_processes$ can significantly affect both accuracy and performance.

Profiling Execution Time

To evaluate how the execution time varies with N, the time was measured with the time module and in the same way the 0s of N can be increased.

```
import math
from multiprocessing import Pool
import time  # Import the time module

def calculate_area(x):
    """ Function to calculate the area of a single
    rectangle """
    return math.sqrt(1 - x**2) * delta_x

def parallel_pi_calculation(N, num_processes):
    """ Function to calculate pi using multiple
    processes """
    global delta_x
    delta_x = 1.0 / N  # Width of each rectangle

    # Create a list of x values where the height of
    the rectangle will be calculated
    x_values = [i * delta_x for i in range(N)]

    # Start timing before the calculation starts
    start_time = time.time()

    # Create a pool of processes and calculate areas
```

```
    with Pool(processes=num_processes) as pool:
        areas = pool.map(calculate_area, x_values)

    # Stop timing after the calculation is complete
    end_time = time.time()

    # Calculate total execution time
    execution_time = end_time - start_time

    # Sum the areas calculated by different
    processes and multiply by 4 to get pi
    total_area = sum(areas)
    pi_approx = 4 * total_area

    return pi_approx, execution_time

# Number of rectangles and number of processes
N = 1000000
num_processes = 8  # Adjust this according to your
    system's CPU cores

# Calculate pi and get the execution time
pi_value, time_taken = parallel_pi_calculation(N,
    num_processes)
print(f"Approximated value of pi: {pi_value}")
print(f"Execution time: {time_taken} seconds")
```

This code will help understand the scalability of the solution as N increases, which is crucial for evaluating the efficiency of the parallelization approach.

Visualization of Results

```
Approximated value of pi: 3.1415928535523587
Execution time: 6.970126628875732 seconds
```

Fig. 2.

B. Distributed Computing Using mpi4py

The provided script is designed to compute the value of π (pi) using the numerical integration technique across distributed systems with the help of the MPI (Message Passing Interface) through the `mpi4py` library. This approach divides the task across multiple processors, which allows for leveraging parallel computing to increase efficiency and reduce execution time.

Key Components of the Code:

1) **Import Statements:**
   - `from mpi4py import MPI`: This imports the MPI module from `mpi4py`, which provides tools for parallel processing using MPI.
   - `import numpy as np`: Numpy is imported for efficient numerical operations.
   - `import time`: Used to measure execution time.
2) **Function `f(x)`:**
   - Calculates the height of the rectangle at position `x` using the formula $f(x) = \sqrt{1 - x^2}$. This represents the function defining the curve of a quarter circle.
3) **Function `main()`:**
   - **MPI Setup**: Initializes MPI environment, where `comm` represents the global communicator that includes all processes. `rank` is the identifier for the current process, and `size` is the total number of processes involved.
   - **Variable Setup**:
     - `N`: Total number of rectangles used in the Riemann sum.
     - `delta_x`: Width of each rectangle.
     - `local_n`: Number of rectangles that each process will handle.
     - `local_a` and `local_b`: Start and end x-values for the rectangles handled by each process.
   - **Local Sum Calculation**:
     - Each process computes the area of its subset of rectangles, using a for loop that iterates over its range (`local_n`) and calculates the area at each point.
   - **Reduction Operation**:
     - `comm.reduce()`: This operation gathers and sums all `local_sum` values from all processes at the root process (rank 0).
   - **Final Computation and Output on Root**:
     - If the process is the root, it finalizes the computation by multiplying the total sum by 4 (to account for the four quadrants of the circle) and prints the approximated value of π along with the execution time.

Profiling

To evaluate the execution time as a function of N, we would adjust N and potentially the number of processes (`size`) to observe how the execution time changes. Notably, this requires running the script in a distributed environment where MPI is effectively utilized.

Profiling Considerations::

- **Increasing** N: As N increases, the granularity of the computation increases, leading to potentially more accurate results but longer computation times.
- **Scaling with Processors (`size`)**: Adding more processors should ideally decrease the execution time, assuming the overhead of communication and synchronization among processes does not outweigh the performance gains from parallel computation.

To measure the execution time effectively, it's crucial to run this script in an MPI-enabled environment, often on a multi-node cluster or a system that supports parallel execution frameworks. The timing measurements are local to each process, but only the root process reports the final execution time, which reflects the time taken for the segment of computation assigned to it, not including initial setup and reduction times.

This is the code:

```
from mpi4py import MPI
import numpy as np
import time
```

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

def main():
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()

    # Total number of rectangles
    N = 1000000
    delta_x = 1.0 / N

    # Each process will handle a subset of the total
     range
    local_n = N // size
    local_a = rank * local_n * delta_x
    local_b = local_a + local_n * delta_x

    local_sum = 0.0

    # Start timing for each process
    start_time = time.time()

    for i in range(local_n):
        x = local_a + (i + 0.5) * delta_x
        local_sum += f(x) * delta_x

    # Gather all local sums into the root process
    total_sum = comm.reduce(local_sum, op=MPI.SUM,
    root=0)

    # End timing for each process
    end_time = time.time()

    # Calculate and print the final result and
    execution time in the root process
    if rank == 0:
        pi_approx = total_sum * 4
        execution_time = end_time - start_time
        print(f"Approximated value of pi: {pi_approx
    }")
        print(f"Execution time: {execution_time}
    seconds")

if __name__ == '__main__':
    main()
```

Visualization of Results

```
Approximated value of pi: 3.1415926539343633
Execution time: 1.480806589126587 seconds
```

Fig. 3.

COMPARATION

NON-PARALLELIZED PYTHON SOLUTION

Focus: This method employs a simple single-threaded for loop to compute the area under the curve $f(x) = \sqrt{1-x^2}$ by summing the areas of rectangles with width Δx and height $f(x_i)$. It runs sequentially on a CPU core without parallelization.

Advantages

- **Simplicity**: The code is simple and easy to understand, so it is good for educational purposes and simple applications.
- **Ease of implementation**: No additional configuration required for parallel processing, avoiding the complexities of managing multiple threads or processes.

Challenges

- **Scalability**: As N increases, the computation becomes more demanding, and the performance of the method drops significantly due to its sequential nature.

PARALLEL COMPUTING USING MULTIPROCESSING

Focus: Uses Python's `multiprocessing` library to parallelize the task across multiple CPU cores. Each process computes a portion of the total area, and the results are aggregated to approximate π.

Advantages

- **Improved efficiency**: By distributing the workload across multiple processors, faster computation can be achieved compared to the single-threaded approach.
- **Resource utilization**: Makes better use of multicore processors by executing tasks in parallel, thereby increasing computational speed.

Challenges

- **Overhead**: Managing multiple processes can introduce overhead, especially when collecting and summing the results of each process, so that when we get the value of pi, it gives us a much larger result than the first code.

DISTRIBUTED COMPUTING USING MPI

Approach: Uses the MPI framework through `mpi4py` for distributed computing across multiple nodes, dividing the computing task so that each node processes a portion of the total area.

Advantages

- **Scalability**: Can handle computations that are too large for a single machine using resources from multiple computers on a network.
- **High performance**: Ideal for large-scale numerical problems where execution time is crucial and can be significantly reduced compared to single-node computations.

Challenges

- **Communication overhead**: The need for communication between different nodes can introduce latency, potentially affecting performance.
- **Configuration complexity**: Requires more complex configuration of the MPI environment, which may not be justified for small tasks.

Comparative summary

- **Single-threading**: Best for simple, small-scale tasks; limited by CPU core efficiency.
- **Multiprocessing**: Balances complexity and efficiency; leverages multicore processors on the same machine, but does not ensure pi value is lower.
- **MPI**: Scales across multiple machines; excels in high-performance computing environments, but involves complex configuration and communication management.

CONCLUSION

The numerical approximation of π via the Riemann sum method demonstrates the practical application of calculus in solving real-world computational problems. The sequential approach, while straightforward, suffers from scalability issues as the number of rectangles increases, leading to longer computation times. On the other hand, parallel and distributed computing methods significantly enhance computational efficiency, particularly in systems with multi-core or multiple nodes. However, these methods introduce complexity in terms of setup and management, requiring careful consideration of the overhead associated with managing multiple processes and communication between them. This study illustrates the importance of choosing the appropriate computational strategy based on the problem size and available resources, balancing between computational accuracy, speed, and resource utilization. The results advocate for a thoughtful application of parallel and distributed computing, especially in large-scale computational tasks where performance is crucial.

REFERENCES

[1] "multiprocessing — Process-based parallelism," Python documentation, 2024. https://docs.python.org/3/library/multiprocessing.html (accessed Apr. 19, 2024).

[2] "Chapter 13. Parallel Your Python — Python Numerical Methods," Berkeley.edu, 2020. https://pythonnumericalmethods.studentorg.berkeley.edu/notebooks/chapter13.00-Parallel-Your-Python.html (accessed Apr. 19, 2024).

[3] "Multiprocessing — Python Numerical Methods," Berkeley.edu, 2020. https://pythonnumericalmethods.studentorg.berkeley.edu/notebooks/chapter13.02-Multiprocessing.html (accessed Apr. 19, 2024).