Parallel Programming with Python; Riemann´s Method Application

Sonia Estefanía Mendía Martínez
Data Engineering
Universidad Politécnica de Yucatán
Ucú, Yucatán, México
2109104@upy.edu.mx

***Abstract*—This report investigates the use of the Riemann sum method for approximating the value of pi. The Riemann sum method is a numerical integration technique that divides an interval into smaller subintervals and uses the area under a curve to estimate the integral. It is one of the first ways to teach and learn about integrals.**

Three implementations of the Riemann sum method are explored: a sequential version without any parallelization, a parallelized version using multiprocessing, and a distributed parallelized version using MPI (Message Passing Interface) with mpi4py. The goal is to compare the accuracy, efficiency, and scalability of these methods in approximating pi.

The primary objectives of this report are to analyze and compare the accuracy, efficiency, and scalability of the three implementations in approximating pi using the Riemann sum method. By evaluating these implementations, we aim to gain insights into the effectiveness of parallel computing techniques, such as multiprocessing and distributed computing with MPI, in numerical integration tasks and their impact on the accuracy and performance of pi approximation algorithms.

Keywords: Riemann sum, numerical integration, pi approximation, parallel computing, multiprocessing, MPI, mpi4py.

I. Introduction to Riemann sum and problematic

The Riemann sum allows us to approximate the area under the curve by breaking the region into a finite number of rectangles. This method, although traditionally associated with calculus and mathematical analysis, finds practical applications in data engineering and coding, especially in areas involving numerical computations and simulations. Now, in the context of parallel programming with Python, the Riemann sum method can also be applied in various ways to improve computational efficiency and speed up numerical calculations, such as the estimation of pi.

The estimation of pi (π) has been a fundamental problem in mathematics and computation for centuries. One of the classical methods to approximate π is through numerical integration using the Riemann sum; this method divides a region into smaller subregions, computes the area of each subregion, and sums these areas to approximate the total area or integral. In coding, the Riemann sum method can also be

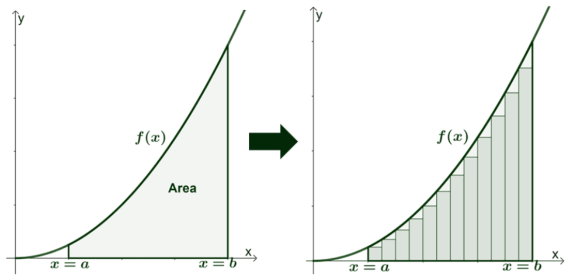


Fig. 1.

employed to approximate the value of pi (π) by integrating the function $f(x) = \sqrt{1-x^2}$ over the interval $[0, 1]$, representing the area under the quarter-circle curve. Parallel programming techniques can be applied to parallelize the Riemann sum calculations, dividing the interval into smaller subintervals and distributing the workload among multiple processing units or cores for concurrent computation.

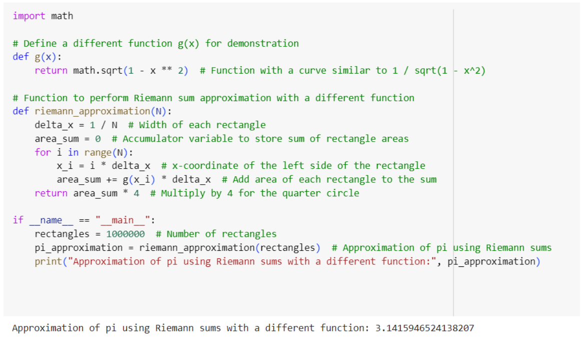
II. Implementation in Python

A. Implementation 1: No Parallelization

The Riemann sum method provides a numerical approach to approximating integrals by dividing the integration interval into smaller subintervals (rectangles in this case), calculating the area of each subinterval, and summing these areas to approximate the total area or integral.

Using a different function $g(x)$ allows for flexibility in choosing the curve to be integrated, showcasing the versatility of the Riemann sum method in handling various functions.

Non-parallelized Riemann sum computations are straightforward to implement but may require longer computation times for large numbers of rectangles, making them less efficient for computationally intensive tasks. The accuracy of the approximation depends on the number of rectangles used in the Riemann sum calculation, with a higher number of rectangles leading to a more accurate approximation but requiring more computational resources.



```
import math

# Define a different function g(x) for demonstration
def g(x):
    return math.sqrt(1 - x ** 2)  # Function with a curve similar to 1 / sqrt(1 - x^2)

# Function to perform Riemann sum approximation with a different function
def riemann_approximation(N):
    delta_x = 1 / N  # Width of each rectangle
    area_sum = 0  # Accumulator variable to store sum of rectangle areas
    for i in range(N):
        x_i = i * delta_x  # x-coordinate of the left side of the rectangle
        area_sum += g(x_i) * delta_x  # Add area of each rectangle to the sum
    return area_sum * 4  # Multiply by 4 for the quarter circle

if __name__ == "__main__":
    rectangles = 1000000  # Number of rectangles
    pi_approximation = riemann_approximation(rectangles)  # Approximation of pi using Riemann sums
    print("Approximation of pi using Riemann sums with a different function:", pi_approximation)
```

```
Approximation of pi using Riemann sums with a different function: 3.1415946524138207
```

Fig. 2.

B. Implementation 2: Parallel Computing with Multiprocessing

The code demonstrates an efficient approach to parallelizing the Riemann sum computation using the multiprocessing module in Python. By dividing the integration interval into smaller subintervals and processing them concurrently across multiple processes, the code takes advantage of parallel processing to improve computational efficiency.

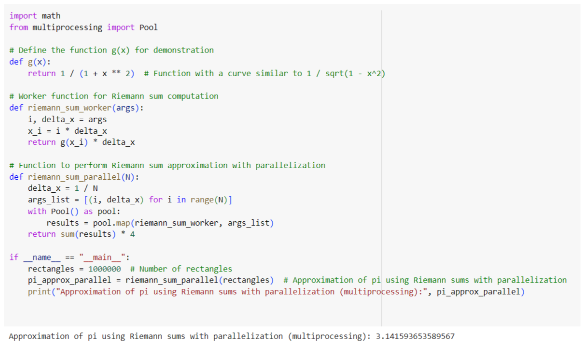
Parallelization using multiprocessing enhances the scalability of the Riemann sum method, allowing it to handle larger numbers of rectangles and more complex computations efficiently, and the performance gains from parallelization become more significant as the number of rectangles and the computational workload increase. Therefore, parallelization can lead to more accurate and precise results in approximations.

Moreover, multiprocessing enables the code to utilize multiple CPU cores effectively, distributing the computational workload among them and speeding up the overall computation of the Riemann sum approximation. This utilization of multiple cores contributes to improved system resource utilization and faster execution times compared to sequential processing.

C. Implementation 3: mpi4py and MPI

The code demonstrates the use of MPI (Message Passing Interface) through the mpi4py library for distributed parallelization. By dividing the workload among multiple processes, it leverages parallel computing to improve efficiency in numerical integration tasks.

Here we also use the Decimal module to perform calculations with higher precision, ensuring accurate results for mathematical operations involving floating-point numbers. This is



```
import math
from multiprocessing import Pool

# Define the function g(x) for demonstration
def g(x):
    return 1 / (1 + x ** 2)  # Function with a curve similar to 1 / sqrt(1 - x^2)

# Worker function for Riemann sum computation
def riemann_sum_worker(args):
    i, delta_x = args
    x_i = i * delta_x
    return g(x_i) * delta_x

# Function to perform Riemann sum approximation with parallelization
def riemann_sum_parallel(N):
    delta_x = 1 / N
    args_list = [(i, delta_x) for i in range(N)]
    with Pool() as pool:
        results = pool.map(riemann_sum_worker, args_list)
    return sum(results) * 4

if __name__ == "__main__":
    rectangles = 1000000  # Number of rectangles
    pi_approx_parallel = riemann_sum_parallel(rectangles)  # Approximation of pi using Riemann sums with parallelization
    print("Approximation of pi using Riemann sums with parallelization (multiprocessing):", pi_approx_parallel)
```

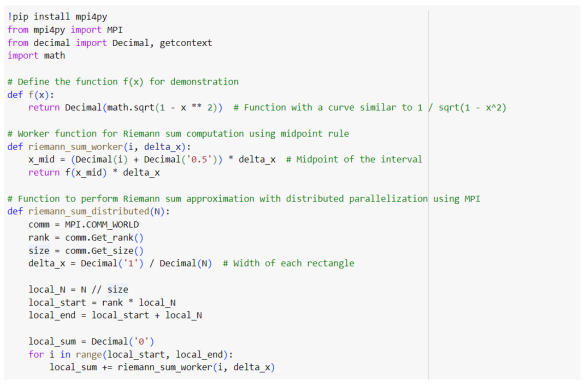
```
Approximation of pi using Riemann sums with parallelization (multiprocessing): 3.141593653589567
```

Fig. 3.

particularly important for numerical methods like Riemann sums that require precision in their calculations.

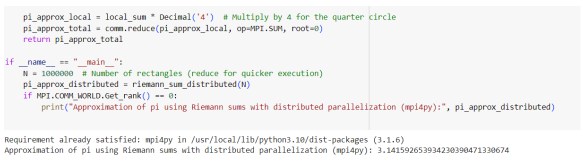
Moreover, the code's design allows it to scale efficiently with the number of rectangles (N). By distributing the workload across multiple processes, it can handle larger values of N without sacrificing performance significantly.

The output of the code provides an approximation of π using the Riemann sum method with distributed parallelization. The resulting approximation is approximately 3.141592653934230390471330674. Comparing this with the actual value of π (3.141592653589793), we can see that the approximation is quite close, and accurate to several decimal places.



```
!pip install mpi4py
from mpi4py import MPI
from decimal import Decimal, getcontext
import math

# Define the function f(x) for demonstration
def f(x):
    return Decimal(math.sqrt(1 - x ** 2))  # Function with a curve similar to 1 / sqrt(1 - x^2)

# Worker function for Riemann sum computation using midpoint rule
def riemann_sum_worker(i, delta_x):
    x_mid = (Decimal(i) + Decimal('0.5')) * delta_x  # Midpoint of the interval
    return f(x_mid) * delta_x

# Function to perform Riemann sum approximation with distributed parallelization using MPI
def riemann_sum_distributed(N):
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()
    delta_x = Decimal('1') / Decimal(N)  # Width of each rectangle

    local_N = N // size
    local_start = rank * local_N
    local_end = local_start + local_N

    local_sum = Decimal('0')
    for i in range(local_start, local_end):
        local_sum += riemann_sum_worker(i, delta_x)
```

Fig. 4.



```
    pi_approx_local = local_sum * Decimal('4')  # Multiply by 4 for the quarter circle
    pi_approx_total = comm.reduce(pi_approx_local, op=MPI.SUM, root=0)
    return pi_approx_total

if __name__ == "__main__":
    N = 1000000  # Number of rectangles (reduce for quicker execution)
    pi_approx_distributed = riemann_sum_distributed(N)
    if MPI.COMM_WORLD.Get_rank() == 0:
        print("Approximation of pi using Riemann sums with distributed parallelization (mpi4py):", pi_approx_distributed)
```

```
Requirement already satisfied: mpi4py in /usr/local/lib/python3.10/dist-packages (3.1.6)
Approximation of pi using Riemann sums with distributed parallelization (mpi4py): 3.141592653934230390471330674
```

Fig. 5.

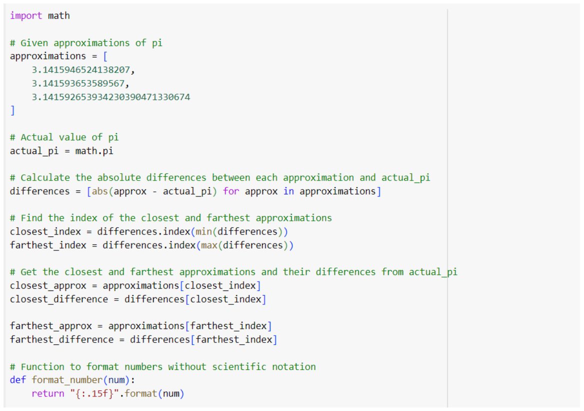
III. Comparisons and Conclusions

The given approximations of π are quite precise, as they differ from the actual value of π by very small amounts. The closest approximation is 3.141592653934230, using MPI, with a difference from the actual value of approximately 0.000000000344437. The farthest approximation is 3.141594652413821, with a difference of approximately 0.000001998824028.

Accuracy of Riemann Sum Method: The Riemann sum method used to calculate these approximations is accurate, as it yields results very close to the actual value of π. This demonstrates the effectiveness of numerical integration techniques in approximating mathematical constants.

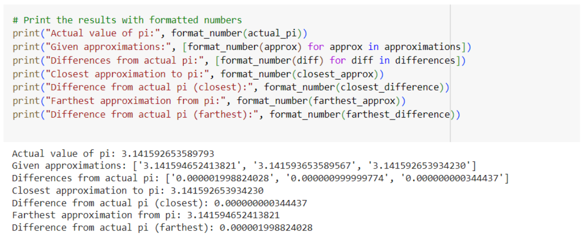
While the non-parallelized Riemann sum method is suitable for simple numerical integration tasks and demonstrations, parallelized implementations using multiprocessing or distributed computing techniques can significantly improve computational efficiency and scalability for larger-scale computations and real-world applications.

On the other hand, parallelization using multiprocessing significantly improves the computational efficiency, scalability, and accuracy of the Riemann sum method, making it a valuable technique for numerical integration and scientific computations in Python.



```python
import math

# Given approximations of pi
approximations = [
    3.1415946524138207,
    3.141593653589567,
    3.141592653934230390471330674
]

# Actual value of pi
actual_pi = math.pi

# Calculate the absolute differences between each approximation and actual_pi
differences = [abs(approx - actual_pi) for approx in approximations]

# Find the index of the closest and farthest approximations
closest_index = differences.index(min(differences))
farthest_index = differences.index(max(differences))

# Get the closest and farthest approximations and their differences from actual_pi
closest_approx = approximations[closest_index]
closest_difference = differences[closest_index]

farthest_approx = approximations[farthest_index]
farthest_difference = differences[farthest_index]

# Function to format numbers without scientific notation
def format_number(num):
    return "{:.15f}".format(num)
```

Fig. 6.



```python
# Print the results with formatted numbers
print("Actual value of pi:", format_number(actual_pi))
print("Given approximations:", [format_number(approx) for approx in approximations])
print("Differences from actual pi:", [format_number(diff) for diff in differences])
print("Closest approximation to pi:", format_number(closest_approx))
print("Difference from actual pi (closest):", format_number(closest_difference))
print("Farthest approximation from pi:", format_number(farthest_approx))
print("Difference from actual pi (farthest):", format_number(farthest_difference))
```

```
Actual value of pi: 3.141592653589793
Given approximations: ['3.141594652413821', '3.141593653589567', '3.141592653934230']
Differences from actual pi: ['0.000001998824028', '0.000000999999774', '0.000000000344437']
Closest approximation to pi: 3.141592653934230
Difference from actual pi (closest): 0.000000000344437
Farthest approximation from pi: 3.141594652413821
Difference from actual pi (farthest): 0.000001998824028
```

Fig. 7.

References

1) Riemann sum - two rules, approximations, and examples. (n.d.). The Story of Mathematics. Retrieved April 19, 2024, from https://www.storyofmathematics.com/riemann-sum/
2) Parallel Programming with Python. (n.d.). GitHub Repository. Retrieved April 19, 2024, from https://github.com/soniamendia/Parallel-Programming-with-Python.git