Parallel Programming with Python

Julio Dzul
Data Engineering
Universidad Politécnica de Yucatán
Ucú, Yucatán, México
2009048@upy.edu.mx

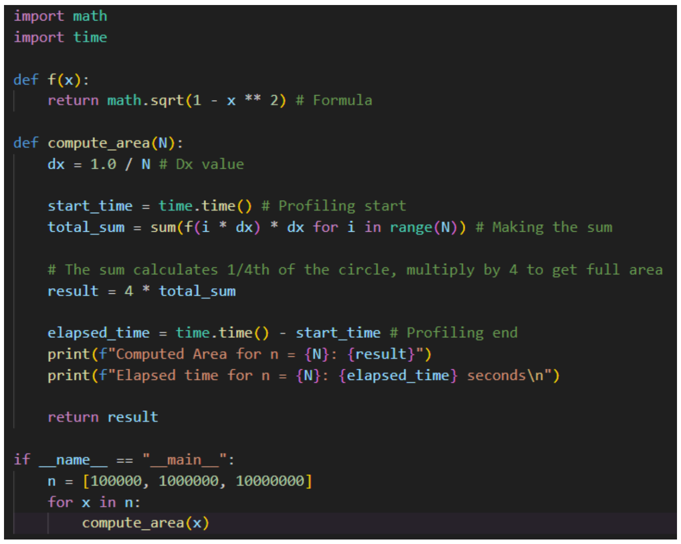
I. Introduction

Numerical integration is a fundamental method in computational mathematics, providing an approach to approximate the definite integrals when analytical solutions are difficult or impossible to find. This report explores numerical integration to approximate the value of π by calculating the area under the curve of a quarter circle using the Riemann sum method. The Riemann sum approximates the integral by dividing the area under a curve into rectangles and summing their areas.

The primary goal of this study is to evaluate the performance of three different implementations: a non-parallel approach, a parallel approach using Python's `multiprocessing` library, and a distributed parallel approach utilizing the `mpi4py` library for MPI-based parallel computing. We seek to understand the efficiency, scalability, and practicality of each method, especially as the problem size increases. Performance metrics were collected and analyzed to determine the most effective approach under varying computational loads and resources.

II. Solutions

A. No Parallelization Solution



```python
import math
import time

def f(x):
    return math.sqrt(1 - x ** 2) # Formula

def compute_area(N):
    dx = 1.0 / N # Dx value

    start_time = time.time() # Profiling start
    total_sum = sum(f(i * dx) * dx for i in range(N)) # Making the sum

    # The sum calculates 1/4th of the circle, multiply by 4 to get full area
    result = 4 * total_sum

    elapsed_time = time.time() - start_time # Profiling end
    print(f"Computed Area for n = {N}: {result}")
    print(f"Elapsed time for n = {N}: {elapsed_time} seconds\n")

    return result

if __name__ == "__main__":
    n = [100000, 1000000, 10000000]
    for x in n:
        compute_area(x)
```

Fig. 1. No Parallelization Solution code

The initial code provided within this report outlines a sequential method for approximating the value of π utilizing the Riemann sum. This particular implementation does not employ any form of parallelization. The function $f(x) = \sqrt{1 - x^2}$ describes the upper right quarter of a unit circle. The Riemann sum is calculated by summing the areas of rectangles under this curve, with the width $\Delta x = \frac{1}{N}$ and height given by $f(x_i)$ at each subdivision $x_i = i\Delta x$.

The Python function `compute_area(N)` executes this calculation across N subdivisions. It is structured as follows:

- **Defining Δx:** The width of each rectangle is set to $\Delta x = \frac{1}{N}$.
- **Profiling Start:** The `time.time()` function records the current time before the summation begins.
- **Computing the Riemann Sum:** A summation loop calculates the total area using a list comprehension, multiplying the function value by Δx for each x_i.
- **Scaling the Result:** Since the calculation only accounts for a quarter of the circle's area, the result is multiplied by 4.
- **Profiling End:** After the summation, the elapsed time is calculated and printed alongside the result.

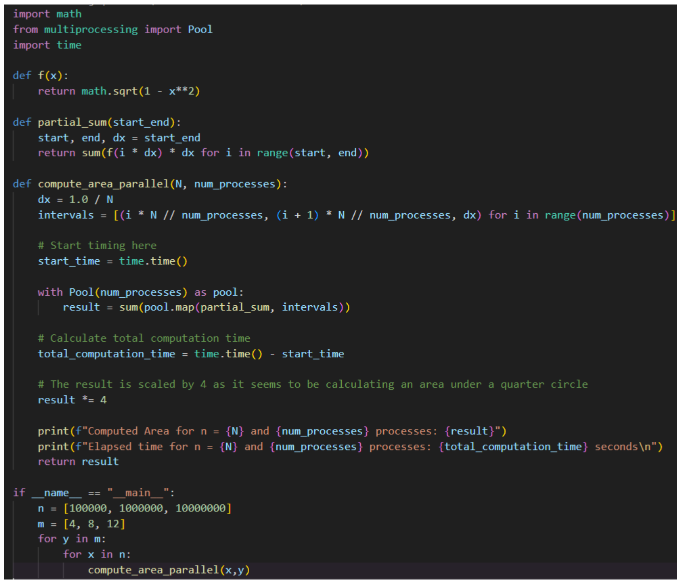
The procedure is encapsulated within the Python idiom `if __name__ == "__main__":` to ensure proper module behavior and is tested with various values of N, specifically 100000, 1000000, and 10000000. This allows for an analysis of the implementation's performance as N increases.

B. Parallelization Solution (multiprocessing)

Following the sequential computation, a parallelized version using Python's `multiprocessing` library is presented. This method leverages the concept of parallel processing, where the computation is divided across multiple CPU cores to execute.

The core components of this implementation are as follows:

- **The Partial Sum Function:** A `partial_sum` function computes a segment of the Riemann sum, receiving start and end indices along with Δx to calculate a localized sum of areas.
- **Division of Work:** The workload is divided into chunks, each assigned to a separate process. This division is encapsulated within the `intervals` list.



```python
import math
from multiprocessing import Pool
import time

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

def partial_sum(start_end):
    start, end, dx = start_end
    return sum(f(i * dx) * dx for i in range(start, end))

def compute_area_parallel(N, num_processes):
    dx = 1.0 / N
    intervals = [(i * N // num_processes, (i + 1) * N // num_processes, dx) for i in range(num_processes)]

    # Start timing here
    start_time = time.time()

    with Pool(num_processes) as pool:
        result = sum(pool.map(partial_sum, intervals))

    # Calculate total computation time
    total_computation_time = time.time() - start_time

    # The result is scaled by 4 as it seems to be calculating an area under a quarter circle
    result *= 4

    print(f"Computed Area for n = {N} and {num_processes} processes: {result}")
    print(f"Elapsed time for n = {N} and {num_processes} processes: {total_computation_time} seconds\n")
    return result

if __name__ == "__main__":
    n = [100000, 1000000, 10000000]
    m = [4, 8, 12]
    for y in m:
        for x in n:
            compute_area_parallel(x,y)
```

Fig. 2. Parallelization Solution (multiprocessing) code

- **Process Pool:** A pool of worker processes is created using the `Pool` class, with the number of processes specified by the user.
- **Map and Compute:** The `pool.map()` function delegates segments of the workload (intervals) to the worker processes and collects the results.
- **Timing and Scaling:** Execution time is measured using the `time` module. The resulting sum is scaled to represent the entire area of the unit circle.

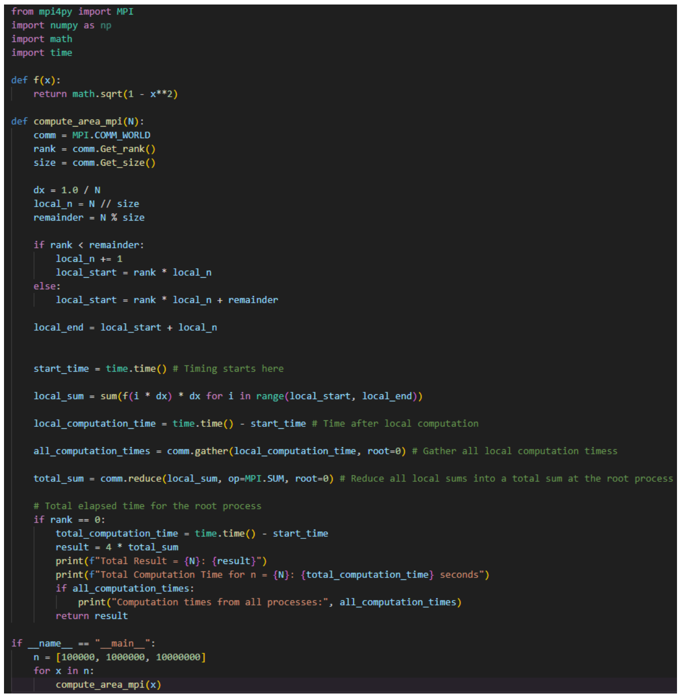
The parallel computation is executed over a range of N values and process counts, assessing the performance across different scales of problem size and parallelism. The observed data provides insights into the effectiveness and scalability of using multiprocessing for numerical integration tasks, serving as a comparative benchmark against both the non-parallel baseline and the subsequent MPI-based distributed parallel computation.

C. *Parallelization Via MPI4PY*

The final code segment shows a distributed parallel approach to compute the Riemann sum for approximating π, utilizing the `mpi4py` library. MPI, or Message Passing Interface, is a standardized and portable message-passing system designed to function on a wide variety of parallel computing architectures. This implementation showcases how to leverage MPI for distributing the computation across multiple processors potentially located on different nodes in a cluster.

The distributed computation is conducted as follows:

- **Process Communication:** The `MPI.COMM_WORLD` object facilitates communication between the processes. Each process identifies its role in the computation through its ‘rank‘, and the total number of processes is given by ‘size‘.



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

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

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

    dx = 1.0 / N
    local_n = N // size
    remainder = N % size

    if rank < remainder:
        local_n += 1
        local_start = rank * local_n
    else:
        local_start = rank * local_n + remainder

    local_end = local_start + local_n


    start_time = time.time() # Timing starts here

    local_sum = sum(f(i * dx) * dx for i in range(local_start, local_end))

    local_computation_time = time.time() - start_time # Time after local computation

    all_computation_times = comm.gather(local_computation_time, root=0) # Gather all local computation timess

    total_sum = comm.reduce(local_sum, op=MPI.SUM, root=0) # Reduce all local sums into a total sum at the root process

    # Total elapsed time for the root process
    if rank == 0:
        total_computation_time = time.time() - start_time
        result = 4 * total_sum
        print(f"Total Result = {N}: {result}")
        print(f"Total Computation Time for n = {N}: {total_computation_time} seconds")
        if all_computation_times:
            print("Computation times from all processes:", all_computation_times)
        return result

if __name__ == "__main__":
    n = [100000, 1000000, 10000000]
    for x in n:
        compute_area_mpi(x)
```

Fig. 3. Parallelization Via MPI4PY code

- **Workload Distribution:** The interval [0, 1] is divided into N subdivisions. Each process computes a local sum for a subset of these subdivisions. To handle any uneven division of work, the remainder is distributed among the first few processes.
- **Local Computation:** Every process calculates the local sum of the areas of rectangles within its assigned interval. Timing for this local computation is performed to analyze the efficiency of the distributed workload.
- **Aggregation of Results:** Once local computations are complete, all local sums are reduced (aggregated) to a single sum at the root process (rank 0) using MPI's reduce operation.
- **Profiling and Final Result:** The root process records the total computation time, including the reduction time, and multiplies the aggregated sum by 4 to estimate the entire area of the unit circle. The final result and the computation times from all processes are printed.

This method is evaluated for a set of subdivision values N, similar to the non-parallel and multiprocessing implementations, to provide a comprehensive comparison. The profiling results offer a direct insight into the scalability and performance of the distributed computing model implemented via MPI, highlighting the advantages and limitations of such a distributed approach for numerical integration tasks.

III. Profiling

.

A. No Parallelization Solution

TABLE I
Execution times (in seconds) for 1 process across different values of N

Processes / N	100,000	1,000,000	10,000,000
1	0.03142	0.33146	3.15566

The non-parallel implementation's execution time increases linearly with the number of sub intervals N. This is expected as the computation load increases proportionally to N.

B. Parallelization Solution (multiprocessing)

TABLE II
Execution times (in seconds) for 4, 8, and 12 processes across different values of N

Processes / N	100,000	1,000,000	10,000,000
4	0.18242	0.27762	1.26400
8	0.20641	0.25869	0.96692
12	0.19538	0.28659	0.90865

The multiprocessing implementation shows an improvement in execution time as the number of processes increases. However, this improvement does not scale linearly due to other processes and inter-process communication. Interestingly, for $N = 1000000$, increasing the number of processes from 8 to 12 does not make significant time savings, in the contrary, it reduces effectiveness slightly. This is likely due to inter-process communication.

C. Parallelization Via MPI4PY

TABLE III
Corrected execution times (in seconds) for 4, 8, and 12 processes across different values of N

Processes / N	100,000	1,000,000	10,000,000
4	0.13953	0.10119	0.94908
8	0.01296	0.08423	0.70523
12	0.01595	0.07426	0.70423

The Parallel approach that used MPI4PY presents the most significant efficiency improvements, even with a higher number of processes. This suggests that for large-scale computations distributed across potentially different nodes, MPI offers superior scalability and performance. However, it is important to note that the MPI environment's configuration and network communication overhead can influence these results.

Efficiency and Scalability

Upon reviewing the execution times, the MPI-based implementation proves to be more efficient than the multiprocessing and non-parallel versions, especially as N becomes very large. The efficiencies gained from the MPI approach are notable when the problem size and the computational resources (processes) are increased.

The non-parallel version, while simple, does not benefit from additional CPU cores and exhibits the longest execution times for large N. Multiprocessing offers a middle ground, providing a speedup over the non-parallel implementation without requiring a distributed computing environment.

IV. Conclusions

The computational experiments conducted in this report have provided insights into the performance characteristics of numerical integration using Riemann sums under three different computational implementations. The non-parallel implementation, while being the simplest, scales linearly with the number of subdivisions and is outperformed by parallel computing techniques.

The multiprocessing approach has a noticeable performance boost, demonstrating the benefits of parallel computation on multi-core systems.

The MPI-based distributed parallel implementation appears to be better in terms of scalability and efficiency, particularly for large problem sizes and when distributed across multiple computing nodes. This approach has the potential to harness the full power of modern distributed computing environments, making it the preferred method for large-scale numerical computations.

In conclusion, the choice of implementation must be chosen according to the specific requirements of the task at hand, considering factors such as problem size, the computational capacity of the hardware, and the available computational environment. For single-node systems, multiprocessing can be an effective strategy, but for large-scale computations that span multiple nodes, MPI provides superior performance and should be utilized to its full potential.

References

[1] Python (2024). multiprocessing — Process-based parallelism. Retrieved from Python Documentation website: https://docs.python.org/3/library/multiprocessing.html

[2] Lisandro Dalcin (2024). MPI for Python . Retrieved from MPI for Python: https://mpi4py.readthedocs.io/en/stable/