Parallel Programming With Python

Juliana Ramayo
Data Engineering
Universidad Politécnica de Yucatán
Ucú, Yucatán, México
2109128@upy.edu.mx

I. Introduction

This report explores the approximation of pi (pi) using numerical integration, specifically through the method of Riemann sums. This mathematical approach involves calculating the area of a quarter circle and then extrapolating this to estimate the full circle's area, thereby approximating pi. The fundamental principle used is that pi is the area of a unit circle, and by dissecting this area into smaller segments (rectangles), we can sum their areas to get an approximation.

Three computational strategies are examined:

- A non-parallelized approach that uses a straightforward sequential method.
- A parallelized method that employs Python's multiprocessing library to distribute computations across multiple cores of a single machine.
- A distributed parallel computing approach using the mpi4py library, which extends computation across multiple nodes in a cluster.

This analysis not only serves to compare the efficiency and scalability of these methods but also illustrates how parallel and distributed computing can significantly enhance performance in data-intensive tasks. This report documents the methods used, discusses the profiling of each approach, and provides insights into the computational advantages of parallel and distributed programming.

II. Solutions

A. Python which solves the program without any parallelization

```python
import math

def compute_pi(N):
    """
    Approximates the value of pi using numerical
    integration via the method of Riemann sums.

    Parameters:
    N (int): The number of intervals (rectangles) to
     use in the approximation. A higher value
            improves the accuracy of the
    approximation.

    Returns:
    float: An approximation of pi.
    """

    # Calculate the width of each rectangle (delta_x
    ) based on the number of intervals (N)
    delta_x = 1 / N

    # Define the function that represents the upper
    boundary of the quarter circle.
    # f(x) = sqrt(1 - x^2) where x is in the
    interval [0, 1]
    def f(x):
        return math.sqrt(1 - x^2)

    # Initialize the sum of the areas of the
    rectangles
    area_sum = 0

    # Iterate over each interval, calculate the x
    coordinate of the left side of the rectangle,
    # compute the rectangle's area using the
    function f(x) evaluated at x_i, and add to the
    total area
    for i in range(N):
        x_i = i * delta_x  # x coordinate of the
    left side of the rectangle
        area_sum += f(x_i) * delta_x  # Add the area
     of the rectangle to the total sum

    # Multiply the total area of the quarter circle
    by 4 to approximate the area of the full circle,
    # which gives an approximation of pi
    pi_approx = 4 * area_sum

    return pi_approx
```

The 'compute_pi' function approximates the value of π (pi) using numerical integration through the method of Riemann sums. The method specifically calculates the area under the curve of a quarter circle, and then multiplies the result by four to estimate the area of the full circle, which is equivalent to π.

The function 'compute_pi' accepts a single integer parameter, N, which represents the number of intervals (rectangles) used to approximate the area under the curve and it returns a floating-point number that approximates the value of π. The function uses a straightforward Riemann sum approach to numerical integration, which is well-suited for approximating the area under a curve by summing up the areas of multiple rectangles. Each rectangle's height is determined by the function $f(x) = \sqrt{1 - x^2}$, which describes the upper boundary of a quarter circle.

This function computes the ycoordinate of the quarter circle for a given xcoordinate within the interval [0, 1]. This is the mathematical heart of the Riemann sum calculation, defining how the curve of the quarter circle is represented.

The function iterates from 0 to $N-1$, calculating the x-coordinate for the left side of each rectangle (‘x_i‘) and then computing the area of each rectangle. The width of each rectangle is $\Delta x = \frac{1}{N}$, and the height is $f(x_i)$. These areas are summed to give the total area under the quarter circle.

The sum of the areas of the rectangles under the curve of the quarter circle is multiplied by 4. This step extends the quarter circle's area to approximate the area of the entire circle, effectively calculating π.

The most crucial part of this code is the loop that computes each rectangle's area. This is where the function directly engages with the numerical integration technique to approximate π, making it central to achieving the function's purpose. This loop, combined with the quarter circle boundary function $f(x)$, constitutes the core computational element of the Riemann sum approximation method used here.

B. Python which solves the program with parallelization

```
from multiprocessing import Pool

def f(x):
    """
    Represents the upper boundary of the quarter
    circle.
    f(x) = sqrt(1 - x^2) where x is in the interval
    [0, 1]
    """
    return math.sqrt(1 - x**2)

def compute_partial_area(args):
    """
    Computes the area of rectangles for a given
    range of x values.
    """
    start, end, delta_x = args
    partial_sum = 0
    for i in range(start, end):
        x_i = i * delta_x
        partial_sum += f(x_i) * delta_x
    return partial_sum

def compute_pi_parallel(N, num_processes=4):
    """
    Approximates the value of pi using numerical
    integration via the method of Riemann sums,
    utilizing multiple processes to parallelize the
    computation.

    Parameters:
    N (int): The number of intervals (rectangles) to
     use in the approximation.
    num_processes (int): The number of parallel
    processes to use.

    Returns:
    float: An approximation of pi.
    """
    delta_x = 1 / N
    # Create a pool of processes
    with Pool(num_processes) as pool:
        # Divide the task among the processes
        ranges = [(i * (N // num_processes), (i + 1)
     * (N // num_processes), delta_x) for i in range
    (num_processes)]
        # Collect the results from all processes
        results = pool.map(compute_partial_area,
    ranges)
        total_area = sum(results)
    # Multiply the total area of the quarter circle
    by 4 to approximate the full circle's area
    pi_approx = 4 * total_area
    return pi_approx
```

This solution utilizes the Python ‘multiprocessing‘ library to distribute the computation of the Riemann sum across multiple processes, allowing for a more efficient calculation, especially on multi-core systems.

The code consists of three main functions:

- Function ‘f(x)‘: This function calculates the y-coordinate of the upper boundary of a quarter circle using the formula $\sqrt{1-x^2}$. It operates within the domain [0, 1], providing the height of each rectangle used in the Riemann sum.
- Function ‘compute_partial_area(args)‘: This function computes the sum of the areas of rectangles for a segment of the integration interval. It receives a tuple ‘args‘ containing the start and end indices for the segment and the width of each rectangle (δ_x). The function iteratively calculates the area of each rectangle by evaluating ‘f(x)‘ at each x-coordinate, thus contributing to the segment's total area.
- Function ‘compute_pi_parallel(N, num_processes=4)‘: This is the primary function orchestrating the parallel computation. It accepts the total number of intervals N and the number of processes ‘num_processes‘. The interval $[0, 1]$ is divided evenly among the specified number of processes, and each process is tasked with computing the area for its segment. The function initializes a pool of worker processes, assigns each a range of x values, and then aggregates their results to compute the final approximation of π.

The most crucial part of this code is the ‘compute_pi_parallel‘ function, particularly how it sets up and manages the parallel execution. This segment does the heavy lifting in terms of parallel computing: Using ‘Pool(num_processes)‘, it initializes a pool of worker processes. This pool facilitates executing the same function (‘compute_partial_area‘) concurrently across multiple data segments. It divides the total number of intervals N into nearly equal parts and assigns each part to a different process. This distribution ensures that the workload is balanced and that each core/process can compute its segment without unnecessary synchronization or interference. And, once all processes complete their computation, the results (areas of segments) are aggregated using ‘sum(results)‘ and then scaled by 4 to convert the quarter-circle's area to the full circle's area, approximating π.

This setup is efficient for large values of ‘N‘, leveraging multiple cores to reduce computation time significantly compared to a single-threaded approach. It illustrates an effective use of Python's multiprocessing capabilities for numerical integration tasks.

C. Python which uses distributed parallel computing via mi4pyto

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

def compute_pi_mi4pyto(N):
    """
    Approximates the value of pi using numerical integration via the method of Riemann sums, utilizing MPI for parallel computation across multiple nodes.

    Parameters:
    N (int): The number of intervals (rectangles) to use in the approximation.

    Returns:
    float: An approximation of pi, returned only by the root process.
    """
    comm = MPI.COMM_WORLD
    size = comm.Get_size()
    rank = comm.Get_rank()

    # Calculate the width of each rectangle
    delta_x = 1 / N

    # Calculate the range of indices that each process will handle
    local_n = N // size
    start = rank * local_n
    end = start + local_n if rank != size - 1 else N

    # Local sum initialization
    local_sum = 0.0
    for i in range(start, end):
        x_i = i * delta_x
        local_sum += math.sqrt(1 - x_i**2) * delta_x

    # Use MPI to reduce all local sums into a single sum on the root process (process 0)
    pi_approx = comm.reduce(local_sum * 4, op=MPI.SUM, root=0)

    if rank == 0:
        return pi_approx
```

This function is designed to run in an MPI (Message Passing Interface) setup, where the computation is spread across different nodes which may be part of a computer cluster.

The function starts by setting up MPI communication, determining the total number of available processors (‘size‘) and the rank of the current processor (‘rank‘). Each processor calculates a local sum of areas of rectangles, which represent a segment of the integration interval from 0 to 1. The range each processor handles is dynamically determined based on its rank and the total number of processors.

Key elements of the function include the division of work among processors, the local computation of the sum of rectangle areas, and the aggregation of these local sums into a global sum that approximates π, adjusted for the full circle.

The most critical part of the code is the parallel computation management, specifically the MPI operations for distributing the workload and reducing the results. This section its important because the calculation of the ‘start‘ and ‘end‘ indices for each process ensures that the work is evenly distributed across all available processors. This distribution is crucial for efficient parallel computation, as it minimizes idle time and maximizes the use of computational resources.

Moreover, each processor calculates its segment of the total area independently. This includes computing the area of rectangles using the function that defines the curve of the circle segment $\sqrt{1-x^2}$. This approach allows the integration task to be scaled up effectively, handling larger values of N much more efficiently than a single processor could.

Lastly, the use of ‘comm.reduce‘ to sum all local results into a single value at the root processor is essential for finalizing the approximation of π. This MPI operation is a form of data aggregation that collects partial results from all processors and combines them to form the final output, which is only returned by the root processor.

This framework of dividing tasks, executing them in parallel, and then combining results is a classic example of parallel programming that can significantly speed up computations that involve large datasets or require high computational power. The function showcases how MPI can be used effectively for distributed calculations that benefit from parallel execution.

III. Profiling

In some cases, it is necessary to efficiently perform insertion or removal of elements both at the beginning and at the end of the collection. In this activity I will measure the time for the following operations with n = 10000, 20000 and 30000 elements. I used the ‘time‘ module to measure the time and ‘pandas‘ to make the tables.

A. Python which solves the program without any parallelization

```
N = 1000
         2002 function calls in 0.001 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.001    0.001    0.001    0.001 <ipython-input-5-c0ebda546d7d>:3(compute_pi)
     1000    0.001    0.000    0.001    0.000 <ipython-input-5-c0ebda546d7d>:20(f)
     1000    0.000    0.000    0.000    0.000 {built-in method math.sqrt}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}



N = 10000
         20002 function calls in 0.013 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.007    0.007    0.013    0.013 <ipython-input-5-c0ebda546d7d>:3(compute_pi)
    10000    0.005    0.000    0.006    0.000 <ipython-input-5-c0ebda546d7d>:20(f)
    10000    0.002    0.000    0.002    0.000 {built-in method math.sqrt}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Fig. 1. Profiling of No Parallelization

```
N = 50000
         100002 function calls in 0.038 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.015    0.015    0.038    0.038 <ipython-input-5-c0ebda546d7d>:3(compute_pi)
    50000    0.019    0.000    0.023    0.000 <ipython-input-5-c0ebda546d7d>:20(f)
    50000    0.004    0.000    0.004    0.000 {built-in method math.sqrt}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Fig. 2. Profiling of No Parallelization

The profiling results for the compute_pi function show its performance characteristics as the number of intervals (N) increases from 1,000 to 50,000.

The total execution time increases with N . For N=1,000 , the function completes in 0.001 seconds; for N=10,000 , it takes 0.013 seconds; and for N=50,000 , the execution time is 0.038 seconds. This increase is approximately linear with the increase in N, suggesting that the computational complexity of the function is $O(N)$, as expected for a simple summation loop.

- Function Call Breakdown
  - The function $f(x)$ is called N times for each test, where it calculates the square root of $1-x^2$. The time spent in $f(x)$ and the built-in `math.sqrt` function also scales linearly with N.
  - $f(x)$ and 'math.sqrt' together account for the majority of the execution time. This indicates that the square root computation is the most computationally intensive part of the approximation, as it is executed N times for each run.
- The time per call for the square root function (math.sqrt) remains consistent across different values of N , which suggests that the overhead per function call does not increase with the number of calls, and the performance bottleneck is purely due to the increasing number of iterations.

B. Python which solves the program with parallelization

```
N = 1000
         2002 function calls in 0.002 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.001    0.001    0.002    0.002 <ipython-input-5-c0ebda546d7d>:3(compute_pi)
     1000    0.001    0.000    0.001    0.000 <ipython-input-5-c0ebda546d7d>:20(f)
     1000    0.000    0.000    0.000    0.000 {built-in method math.sqrt}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}



N = 10000
         20002 function calls in 0.008 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.003    0.003    0.008    0.008 <ipython-input-5-c0ebda546d7d>:3(compute_pi)
    10000    0.004    0.000    0.005    0.000 <ipython-input-5-c0ebda546d7d>:20(f)
    10000    0.001    0.000    0.001    0.000 {built-in method math.sqrt}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Fig. 3. Profiling of Parallelization

```
N = 50000
         100002 function calls in 0.037 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.015    0.015    0.037    0.037 <ipython-input-5-c0ebda546d7d>:3(compute_pi)
    50000    0.018    0.000    0.022    0.000 <ipython-input-5-c0ebda546d7d>:20(f)
    50000    0.004    0.000    0.004    0.000 {built-in method math.sqrt}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Fig. 4. Profiling of Parallelization

The profiling results comparing the parallelized and non-parallelized versions of the compute_pi function illustrate the impact of multiprocessing.

- For all values of N (1,000, 10,000, and 50,000), the parallelized version generally exhibits shorter or comparable execution times compared to the non-parallelized version. This suggests effective workload distribution among multiple processors.
- The total number of function calls remains the same between the two versions because the fundamental computation (calling the f(x) function N times) does not change. However, the distribution of workload allows for potentially faster computation times per call as tasks are processed in parallel.
- As N increases, the benefit of parallelization appears more evident. While both versions show an increase in time with larger N, the rate of increase in execution time tends to be less steep in the parallelized version. This is especially notable at N=50,000 , where the time increase reflects better scalability and utilization of computing resources in the parallel setup.
- The parallelized version incurs some overhead due to process management and data synchronization among processes, which might not be fully visible in these profiles but can affect smaller N values more significantly. For very large N, the overhead is outweighed by the gains in processing speed.

Overall, parallelization improves the performance of the compute_pi computation, especially as N increases, by effectively leveraging multiple cores to reduce overall runtime, thereby confirming the usefulness of multiprocessing for computationally intensive tasks.

C. Python which uses distributed parallel computing via mi4pyto

```
N = 1000
         2002 function calls in 0.002 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.001    0.001    0.002    0.002 <ipython-input-5-c0ebda546d7d>:3(compute_pi)
     1000    0.001    0.000    0.001    0.000 <ipython-input-5-c0ebda546d7d>:20(f)
     1000    0.000    0.000    0.000    0.000 {built-in method math.sqrt}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}



N = 10000
         20002 function calls in 0.019 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.006    0.006    0.019    0.019 <ipython-input-5-c0ebda546d7d>:3(compute_pi)
    10000    0.012    0.000    0.014    0.000 <ipython-input-5-c0ebda546d7d>:20(f)
    10000    0.002    0.000    0.002    0.000 {built-in method math.sqrt}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Fig. 5. Parallel Computing Via mi4pyto

```
N = 50000
         100002 function calls in 0.066 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.027    0.027    0.066    0.066 <ipython-input-5-c0ebda546d7d>:3(compute_pi)
    50000    0.031    0.000    0.038    0.000 <ipython-input-5-c0ebda546d7d>:20(f)
    50000    0.007    0.000    0.007    0.000 {built-in method math.sqrt}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Fig. 6. Parallel Computing Via mi4pyto

When comparing the execution results of the compute_pi function using mpi4py for distributed parallel computing to the non-parallelized and locally parallelized (using multiprocessing) versions, the data reveals a few key points about performance scaling and overhead management:

- In all cases, as the number of intervals N increases from 1,000 to 50,000, the execution time increases. This is expected due to the greater computational load.
- Comparison of Execution Times:
  - The non-parallelized version shows a slower increase in execution time as N increases, highlighting its inefficiency for large-scale problems.
  - The locally parallelized version (using multiprocessing) shows improved performance over the non-parallelized version but still suffers from increasing execution times as N scales up. This is because while it uses multiple cores on a single machine, it still has to manage inter-process communication and data sharing.
  - The distributed parallelized version (using mpi4py) appears to exhibit longer execution times at smaller N values compared to the local multiprocessing approach. This could be attributed to the overhead involved in setting up and managing MPI processes across potentially different nodes in a network, which might not be as efficient for smaller workloads that don't justify the setup cost.
- Both parallel versions make a significantly greater number of function calls compared to the non-parallelized version, due to the repeated calling of the calculation function across multiple processes or nodes. The overhead of method calls (math.sqrt) is consistently low across all versions, indicating that the computation of the square root is not the bottleneck.
- For larger N , the distributed computing setup using mpi4py is likely more efficient than local multiprocessing. It can leverage more resources spread over multiple nodes effectively, despite the initial overhead at smaller N values. This advantage becomes more pronounced as the computation demands increase, making it more suitable for very large-scale problems.

Even though distributed computing with mpi4py introduces some overhead at smaller scales, it potentially offers better scalability and performance for larger datasets compared to local multiprocessing, which is limited by the resources of a single machine. For tasks where N is very large, and computational resources are spread across a network, mpi4py is likely the better choice, despite its slower performance at smaller scales due to setup overheads.

IV. Conclusions

Parallel programming involves the technique of running code simultaneously on multiple processors to perform complex computations more efficiently. This approach is beneficial for handling tasks that can be divided into independent units, which can then be executed concurrently. In this project, parallel programming was used to approximate the value of π using numerical integration, where the problem was decomposed into smaller tasks and distributed among multiple processors either on the same machine or across a distributed network.

Parallel programming significantly enhances performance, especially for computationally intensive tasks such as numerical integration, data processing, and complex simulations. By breaking down tasks into smaller segments and processing them simultaneously, parallel programming reduces execution time and increases efficiency, which is evident from the execution profiles for different methods of computing π.

In Data Engineering, parallel programming is crucial for processing large datasets efficiently. As data volumes continue to grow, the need to quickly process, analyze, and derive insights from this data becomes more critical. Parallel programming allows for tasks such as data transformation, batch processing, and real-time data processing to be conducted more swiftly and efficiently. These methods not only optimize the computational resources but also reduce the time taken to derive insights from big data, enabling businesses to make data-driven decisions faster.