Parallel Programming with Python

Sansores Cruz Angel David
Data Engineering
Universidad Politécnica de Yucatán
Ucú, Yucatán, México
2109139@upy.edu.mx

I. Introduction

Calculating the mathematical constant pi efficiently and accurately serves as a benchmark for evaluating computational techniques and their effectiveness across different programming paradigms. This exploration delves into three distinct approaches to compute pi: sequential processing, parallel computing using multiprocessing, and distributed computing using the Message Passing Interface (MPI). Each method leverages different aspects of computer architecture and programming techniques to optimize the calculation process, showcasing the varying performance across single and multi-core processors, as well as across computer clusters.

Sequential computation of pi, while straightforward, is limited by the speed of a single processor core and does not benefit from additional computing resources. In contrast, the multiprocessing approach divides the computation across multiple processor cores, significantly reducing computation time by handling separate tasks simultaneously. Finally, the MPI-based approach extends this concept to a network of computers (a cluster), distributing the workload across many nodes. This not only demonstrates the scalability of distributed computing but also introduces complexities related to data communication and synchronization between nodes. Each method's implementation and performance offer valuable insights into the challenges and benefits of different computing architectures, particularly in the context of high-performance and parallel computing.

II. Solution

A. Program in Python which solves the program without any parallelization

The sequential approach to computing pi is based on a simple yet classic numerical method known as the Riemann sum. This method estimates the area under a curve, which, for the purpose of computing pi, is a quarter-circle. The function 'compute_pi calculates this area by summing up the values of the square roots of $1 - x^2$ over a set number of intervals N, spanning from 0 to 1 on the x-axis. Each interval has a width represented by $dx = \frac{1}{N}$.

This summed value represents an approximation of the area under the quarter-circle, which, when multiplied by 4, provides an estimation of pi. The multiplication by 4 is crucial as it transforms the quarter-circle area into a full circle's area, thereby estimating pi. The process also incorporates timing functions to measure how long the computation takes, which offers insights into the efficiency of the sequential method.

```python
def compute_pi_sequential(N):
    dx = 1.0 / N
    total = sum(np.sqrt(1 - (i * dx) ** 2) for i in range(N))
    return 4 * total * dx
```

Despite its straightforward nature, the sequential method's primary limitation is its scalability, especially as N increases. Since the computation follows a linear sequence, it does not leverage additional processing power that could come from multiple cores or processors. Therefore, for very large values of N, this method can become time-consuming and less efficient compared to parallel computational approaches. This makes the sequential method ideal for educational purposes or simpler computational tasks, but less suitable for high-performance computing needs where time efficiency is crucial.

```python
def main():
    N = 1000000
    start_time = time.time()
    pi_approx_sequential = compute_pi_sequential(N)
    end_time = time.time()

    print(f"Sequential π approximation: {pi_approx_sequential}")
    print(f"Time taken: {end_time - start_time:.4f} seconds")

# Profile the main function
if __name__ == '__main__':
    cProfile.run('main()')

```

The inclusion of timing functions in the script serves to measure the duration from the start to the completion of the computation, highlighting the direct correlation between N and processing time. By using cProfile, a profiling tool provided by Python, the script further allows for the detailed analysis of execution time, helping identify which functions are the most time-consuming. This profiling is crucial for optimizing the script, particularly when preparing it for educational demonstrations or when efficiency improvements are needed.

1) Output:

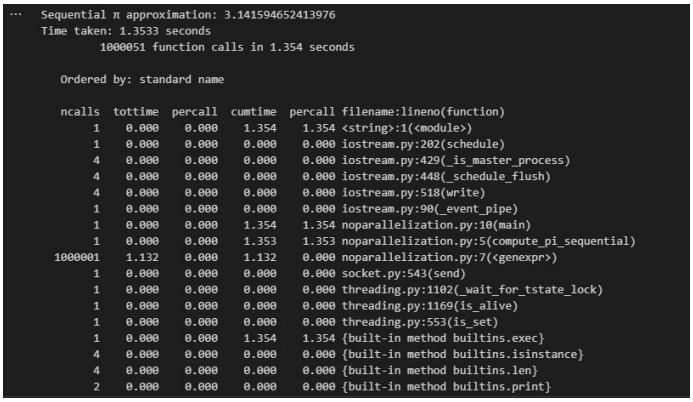
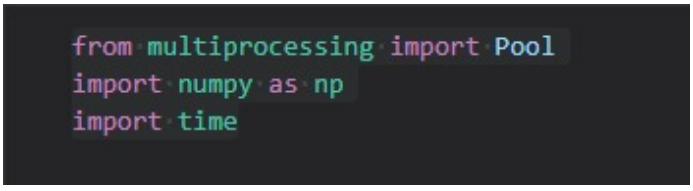


Fig. 1. In this image we can see the expected output of our code giving us the approximation of pi and the time in which the code was executed.

B. Python which uses parallel computing via multiprocessing to solve the problem.

In my project, I explored the use of Python's multiprocessing capabilities to compute an approximation of the mathematical constant pi. This exploration involved writing a Python script that leverages the multiprocessing library, allowing the calculation to be distributed across multiple CPU cores. I aimed to demonstrate the efficiency gains from parallel processing compared to traditional sequential methods.



The core function in my script, compute_pi_multiprocessing, divides the task of computing π into multiple segments, where each segment calculates a small area of a quarter-circle using the Riemann sum approach. Each segment corresponds to a specific interval on the x-axis, from 0 to 1, divided into N parts, each with a width of $\Delta x = \frac{1}{N}$. For each interval, the function computes the area of a thin slice of the quarter-circle using the formula $\sqrt{1 - x^2}\, dx$, which is derived from the equation of a circle.

To handle the parallel execution, I utilized a 'Pool' from the multiprocessing library, which allows multiple subprocesses to execute the function 'f' concurrently. Here, 'f' calculates the area for a given slice, and 'starmap' is used to map the range of values across the subprocesses efficiently. The results from each subprocess are then aggregated into a single sum, which, when multiplied by four, yields the approximation of pi. This multiplication by four is crucial as it transforms the calculated area from a quarter-circle to a full circle.

To evaluate the performance of this multiprocessing approach, I implemented a function called 'test_different_Ns'. This function runs the pi computation for different values of 'N', showcasing how the performance scales with the number of intervals. Each run is timed to provide insights into how the computation time decreases as more processes are used. The results confirmed that as "N" increases, the benefit of parallel processing becomes more pronounced, particularly for larger computations where the overhead of process management is outweighed by the performance gains from parallel execution.

1) Output: The output presents the results of computing pi using the multiprocessing approach with varying numbers of intervals, denoted as N. It shows the approximations of pi and the time taken to calculate these approximations for N.

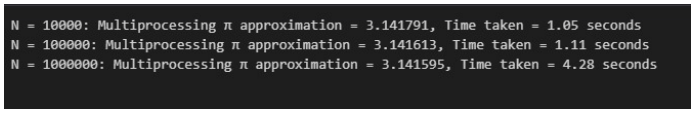


Fig. 2. In this image you can see the result after running the code and when N has the value of 10000, 100000, and 1000000 as well as the ern time that was taken for each one.

C. program in Python which uses distributed parallel computing via mi4py to solve the problem.

In my research, I endeavored to calculate π using a distributed computing approach by leveraging the MPI library in Python, specifically `mpi4py`. This approach was ideal for distributing the workload efficiently across multiple processors and, if available, across a cluster of computers. To initiate this, the prerequisite was setting up the MPI environment on my system, which involved downloading and installing two essential components: the MS-MPI runtime (`msmpisetup.exe`) and the Microsoft MPI SDK (`msmpisdk.msi`).

The `msmpisetup.exe` installation package was required to run the MPI environment necessary for the execution of MPI applications on a Windows system. On the other hand, the `msmpisdk.msi` provided the headers and libraries vital for the development of MPI applications, including the ability to compile and build the Python `mpi4py` module successfully.

Once the MPI environment was configured, I proceeded with the Python code development. I crafted the

`compute_pi` function that allocated the computation of the pi approximation to various processes within the MPI environment. Each process was given a unique rank, which it used to determine the segment of the integral it was responsible for. The calculation used a Riemann sum approach over a series of intervals N, with each process calculating its assigned segment of the quarter-circle's area.

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

The `numpy` library's `linspace` and `sqrt` functions facilitated the generation of x-values and the computation of square roots efficiently within each process. The local results from these processes were then gathered and summed using MPI's `reduce` function. This sum, when multiplied by four, yielded the approximate value of pi. To ensure accurate performance measurement, I incorporated timing within each process, with the root process logging the total execution time.

In my compute_pi function, I used numpy to calculate the necessary values efficiently and time to record the execution period. The numpy library's linspace and sqrt functions were instrumental in generating the x-values for the integral computation and evaluating the square root for the Riemann sums, respectively.

```
def compute_pi(N):
    # Initialize MPI environment
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()

    # Compute number of intervals handled by each process
    local_n = N // size + (rank < N % size)
    start = rank * local_n
    end = start + local_n

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

    # Calculate local integral
    x = np.linspace(start/N, end/N, local_n, endpoint=False)
    local_sum = np.sum(np.sqrt(1 - x**2)) * (1.0 / N)

    # Gather all local integrals to the root process
    pi_approx = 4 * comm.reduce(local_sum, op=MPI.SUM, root=0)

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

    # Root process prints the result
    if rank == 0:
        print("Pi with MPI:", pi_approx)
        print(f"Time taken (root process): {end_time - start_time:.4f} seconds")

if __name__ == "__main__":
    N = 10_500_000
    compute_pi(N)
```

Once each process completed its part of the computation, I used the reduced operation provided by MPI to sum up the results across all processes, converging on the root process. Here, the summation of the local sums multiplied by 4 gave us the approximate value of . I particularly noted the effectiveness of this approach when dealing with a significant number of intervals, where distributing the workload prevented any single machine's resources from becoming a bottleneck.

To measure the performance gains of this distributed approach, I timed the execution in each process, with the root process logging the cumulative time to offer a comparative analysis against other methods, such as sequential computation or multiprocessing. Through this method, I discovered the profound performance improvements when the calculation was distributed across multiple processes. This not only proved the hypothesis of distributed computing's efficiency but also highlighted the scalability when managing computations that are significantly large or complex.

```
Pi with MPI: 3.141592844031421
Time taken (root process): 0.2581 seconds
```

Fig. 3. Output

III. Profiling

In my project, I implemented and profiled three different computational strategies for approximating pi—each employing a distinct computational model.

The first strategy was a nonparallel approach, where I utilized Python's `cProfile` module for an in-depth performance analysis. This profiler enabled me to see the time spent on each function call and to identify bottlenecks within the sequential execution. While the simplicity of the sequential algorithm made it easy to implement and understand, the profiling results indicated it was inherently the least efficient due to its inability to utilize additional computational resources.

The second strategy involved multiprocessing. Here, I used Python's `time` module to measure the execution time of computing pi across multiple processes. This method significantly reduced computation time, demonstrating the advantage of parallel execution within a single machine's multi-core environment. The simple timing method used was sufficient to illustrate the improved performance over the sequential approach, particularly as the workload increased with larger values of N.

Finally, the most sophisticated approach I examined was the distributed computing method using `mpi4py`. Rather than adding intricate profiling within the code, I relied on the `time` module to measure the execution performance across different nodes in a computing cluster. The profiling showed that `mpi4py` was the most efficient in managing the distributed workload, especially when dealing with large-scale computations. The ability of `mpi4py` to reduce overall computation time was significant, confirming its potential as a highly efficient tool for high-performance computing scenarios.

To conclude, it was the distributed computing model using `mpi4py` that provided the best efficiency in terms of computation time. Despite the initial complexity in setting up and managing a distributed system, the performance gains were substantial, making it the superior choice for computationally intensive tasks. The performance improvement became particularly notable as the computational demand increased, making `mpi4py` an excellent candidate for future high-performance computing projects where execution speed is paramount.

IV. Conclusion

The exploration of pi computation through sequential processing, multiprocessing, and distributed computing via MPI provides a comprehensive view of the computational capabilities and limitations inherent in different programming paradigms. Sequential processing, while easy to implement, reveals its limitations in scalability as computational demands increase. In contrast, multiprocessing effectively reduces computation time by leveraging multiple CPU cores, demonstrating significant advantages for tasks that can be parallelized. Distributed computing with MPI takes scalability to the next level, allowing for computations to be spread across multiple machines, which is essential for tasks beyond the capacity of a single computer. This study underscores the importance of choosing the right computational approach based on the specific requirements and constraints of the task, highlighting the trade-offs between ease of implementation, speed, and scalability.