High Performance Computing
”E3 Parallel Programming with Python”

Data Engineering

Class 7-A

Carlo Alejandro Ek Palomo

Didier Omar Gamboa Angulo

UNIVERSIDAD POLITÉCNICA DE YUCATÁN

April 19, 2024

Approximating Pi using Numerical Integration with Sequential and Parallel Approaches

Carlo Alejandro Ek Palomo

***Abstract*—This report presents the computational techniques used to approximate the value of π via numerical integration of a quarter circle. The methods explored range from a basic sequential algorithm to more advanced parallel computing approaches using multiprocessing and distributed computing with MPI.**

I. Introduction

The numerical approximation of mathematical constants has profound implications in various scientific and engineering disciplines. Among these constants, π is paramount due to its ubiquitous presence in mathematical expressions, particularly those describing circular and spherical shapes. This report addresses the problem of approximating π using numerical integration, a fundamental technique in computational mathematics.

Specifically, we approximate π by integrating the function $f(x) = \sqrt{1 - x^2}$, which represents the upper semicircle of the unit circle centered at the origin. By integrating this function from $x = 0$ to $x = 1$, we effectively calculate the area of a quarter circle. Multiplying this result by four provides an estimate for π.

To compute this integral, we employ the Riemann sum approach, a method that approximates the area under a curve by dividing the domain into a series of small segments or rectangles. This method is characterized by its simplicity and adaptability to functions that are otherwise challenging to integrate analytically. In our approach, the interval [0,1] is subdivided into N segments, each of width $\Delta x = \frac{1}{N}$. The height of each rectangle is determined by the function value at each subinterval, and the sum of the areas of these rectangles serves as the approximation for the quarter-circle's area.

II. Solutions

A. Sequential Approach

The sequential algorithm serves as the foundational method for approximating π in our study. This approach is grounded in the simplicity of executing a single-threaded process, which iterates through a predetermined number of subdivisions to calculate the area under the curve $f(x) = \sqrt{1 - x^2}$. The choice to start with a sequential approach allows us to establish a baseline for accuracy and performance, which is essential for comparative analysis with more complex parallel implementations.

The core of the sequential method involves computing the Riemann sum, where the domain of integration is divided into N equal intervals, each of width $\Delta x = \frac{1}{N}$. The algorithm computes the function value at each subinterval, then multiplies this value by Δx to estimate the area of a rectangle under the curve. These areas are then summed to approximate the area of a quarter circle, which is subsequently multiplied by four to estimate π.

```
delta_x = 1 / N
sum_area = 0
for i in range(N):
  xi = i * delta_x
  fi = np.sqrt(1 - xi**2)
  sum_area += fi * delta_x
pi = 4 * sum_area
```

This method is straightforward but computationally intensive as N increases. Each increase in N enhances the accuracy of the approximation by reducing the width of each interval and thereby increasing the number of function evaluations. However, this also leads to longer computation times, as each function evaluation is processed sequentially.

The key expectation from the sequential approach is to understand how effectively the Riemann sum approximates π when computation is confined to a single processor. This setup provides a benchmark for assessing the benefits of more advanced computational strategies such as multiprocessing and distributed computing. By comparing the accuracy and execution time of the sequential approach with parallel techniques, we can evaluate the trade-offs between simplicity and computational speed, setting the stage for exploring how parallel processing can significantly reduce execution times while maintaining or improving the approximation's accuracy.

B. Parallel Computing with Multiprocessing

As computational demands increase, particularly with a higher number of subdivisions N, the sequential approach becomes less efficient due to the linear increase in processing time. To address this limitation and significantly enhance performance, we implemented a parallel computing solution using Python's *multiprocessing* module. This approach leverages multiple CPU cores to concurrently execute calculations, reducing overall computation time.

In the multiprocessing method, the primary challenge is to effectively distribute the computational load among several processes. This is accomplished by dividing the task into equal segments, each of which is handled by a separate process. The domain [0, 1] is partitioned into N intervals, and these intervals are grouped into chunks corresponding to the number of available processes. Each process computes the area for its assigned segment independently, parallelizing the workload.

```
pool = Pool(processes=num_processes)
```

```
tasks = [(i * chunk_size, (i + 1) *
    ↪ chunk_size, N) for i in
    ↪ range(num_processes)]
results = pool.map(worker, tasks)
pi = 4 * sum(results)
```

The 'Pool' object from the multiprocessing library manages a pool of worker processes, distributing tasks among them. Each task consists of a tuple specifying the start and end indices for the sub-intervals and the total number of intervals N. The function 'worker', not shown here, performs the actual computation of the Riemann sum over the specified segment. The results from all workers are then aggregated using the 'sum' function, and multiplied by four to obtain the approximation of π.

This parallel approach is designed to exploit the hardware capabilities effectively, particularly on multi-core systems. By distributing the work among several cores, the algorithm decreases the wall-clock time required to compute the result, compared to the sequential execution. The expectation is that as the number of processes increases, the computation time will decrease, demonstrating near-linear or super-linear speedup depending on the overhead of process management and data aggregation.

However, the efficiency of this approach can be limited by the overhead involved in creating and managing multiple processes and the cost of aggregating results across process boundaries. These factors are particularly significant when the number of processes is very large or when each process computes a very small amount of work. As such, there is an optimal number of processes beyond which performance gains may diminish or even degrade.

Through experimental analysis, this study aims to determine the optimal balance between the number of processes and the size of computation tasks, thereby maximizing the efficiency of the multiprocessing approach.

C. Distributed Parallel Computing with MPI

For tackling even more computationally intensive tasks, particularly those benefiting from distribution across multiple computational nodes, we employed the *mpi4py* library. This Python library interfaces with MPI (Message Passing Interface), a standard used for high-performance distributed computing. The distributed nature of MPI is well-suited for large-scale problems that exceed the memory and processing capabilities of a single machine, allowing for a significant scale-up in computational resources.

In this approach, the computation of π is distributed across multiple processors or nodes, each handling a segment of the overall task. The key concept here is the division of the integration domain into parts that can be processed independently, similar to the chunk division in the multiprocessing approach but on a potentially much larger scale.

```
local_sum = 0
for i in range(local_start, local_end):
  xi = i * delta_x
  fi = np.sqrt(1 - xi**2)
  local_sum += fi * delta_x
global_sum = comm.reduce(local_sum,
    ↪ op=MPI.SUM, root=0)
```

The script begins by calculating the local sum of areas in each process. Each node computes its portion of the Riemann sum from `local_start` to `local_end`, which are calculated based on the process rank and the total number of processes. This parallel computation is a pivotal advantage, as each processor performs its calculations independently without the need for inter-process communication during the summation phase.

Once all processes have computed their respective sums, the results are aggregated using the MPI `reduce` operation. This function collects and sums the `local_sum` values from all processes, storing the final result in the process designated as `root`. This aggregation is a critical step, as it combines the individual contributions into the global sum which is then used to compute the approximation of π.

The distributed MPI approach is expected to offer substantial performance improvements, especially as the number of nodes increases. This method not only speeds up the computation but also distributes the memory load, which can be crucial for very large values of N. However, the efficiency of this approach can be affected by the overhead associated with data communication between nodes. The balance between computation and communication overhead is therefore a key factor in the effectiveness of the MPI-based solution.

This method demonstrates how distributed computing can address scalability challenges inherent in numerical integration tasks, potentially achieving significant reductions in computation time while handling larger datasets that a single system might not be able to process effectively.

Through this advanced computational strategy, we aim to explore the boundaries of parallel computing efficiency, investigating how effectively large-scale distributed systems can cooperate to solve complex mathematical problems.

III. Profiling

A. Test Setup

The execution time analysis was conducted under controlled conditions to ensure comparability between the different methods. The tests were performed on a standard computational platform with the same hardware specifications for all methods to avoid any hardware-induced performance biases. Each method was executed with an approximation granularity set by $N = 1000000$ intervals, representing a high-resolution computation scenario intended to stress the computational capabilities of each approach.

B. Execution Time Analysis

The execution times were meticulously recorded for each method, capturing not only the total time taken to complete the calculation but also the user and system times, which provide insights into the CPU time utilized by the processes and the system overhead involved, respectively. The following table summarizes these metrics:

Method	Execution Time (s)	User Time (s)	System Time (s)
Sequential	1.055	1.189	1.219
Multiprocessing	0.314	0.458	0.921
MPI	0.253	0.760	1.489

The results highlight the significant performance gains achieved through parallel computing techniques. The multiprocessing method demonstrated a substantial decrease in execution time compared to the sequential approach. Similarly, the distributed MPI approach further reduced the time, outperforming the multiprocessing method, which suggests effective scalability and utilization of distributed resources.

C. Analysis of Results

The recorded execution times reveal critical insights into the performance scalability and efficiency of parallel computing architectures. The user and system times offer a deeper understanding of how computational tasks are handled by the operating system and hardware:

1. **Sequential Method**: Exhibited the longest execution and system times, indicating substantial CPU and system resource utilization without the benefits of parallel execution. 2. **Multiprocessing**: Significantly reduced the execution time by leveraging multiple cores, though it showed increased system time, reflecting the overhead involved in managing multiple processes. 3. **MPI**: Although MPI had the shortest execution time, its high user and system times suggest a considerable amount of communication and coordination overhead among the distributed nodes, which is a typical characteristic of distributed computing environments.

This detailed examination aids in identifying optimal computing strategies for tasks requiring high computational resources, guiding future implementations towards more efficient computational designs.

IV. Conclusions

This study has systematically explored various computational strategies to approximate π using numerical integration, ranging from a straightforward sequential method to more sophisticated parallel and distributed computing techniques. The results affirm that all methods are capable of approximating π with a high degree of accuracy. However, the efficiency and scalability offered by parallel computing methods, particularly under heavy computational loads, are pronounced and significant.

The sequential approach, while simplest, proved to be the least efficient in terms of computation time. This method is best suited for environments where simplicity and resource availability are more critical than execution speed. On the other hand, the multiprocessing approach demonstrated a considerable reduction in execution time by effectively utilizing multiple cores of a single machine. This approach strikes a balance between performance and resource utilization, making it suitable for mid-scale computational tasks that require faster processing but do not have access to distributed systems.

The most advanced approach, using the *mpi4py* library for distributed parallel computing, exhibited the highest scalability and performance, particularly in environments that facilitate distributed computing. The MPI method not only reduced the execution time further than the multiprocessing approach but also allowed for the handling of much larger datasets by distributing the memory and computational load across multiple nodes. This capability makes it exceptionally well-suited for large-scale scientific computations where the volume of data or the complexity of the problem exceeds the capacity of single-system resources.

Furthermore, the detailed profiling and analysis indicate that as the number of processors increases, the MPI approach tends to maintain or even improve its efficiency, demonstrating super-linear speedup in some cases. This attribute is particularly valuable in a research or industrial setting where time is a critical factor, and the computational demand often surpasses the capabilities of conventional single-node systems.

In conclusion, while each method has its merits and optimal use cases, the distributed parallel computing approach using MPI stands out for tasks that demand high scalability and rapid execution. Future work should focus on optimizing inter-process communication and reducing overhead to further enhance the performance benefits of distributed computing. Additionally, exploring hybrid models that combine the strengths of multiprocessing and MPI might yield even better efficiencies for a broader range of computational scenarios.