E3. Parallel Programming with Python

Student:
Jair Armando Martinez Castillo

April 19, 2024

1 Introduction

The goal is to develop an algorithm that calculates the value of π with distinct approaches.
With advancements in computational technology, various methods have been developed to compute π more efficiently. This report explores three different computational strategies for approximating π:

- A **sequential approach**, where computations are carried out one after the other, offering simplicity but potentially lower performance on large-scale problems.
- A **multiprocessing approach**, which utilizes multiple processors concurrently to enhance performance by dividing the task into smaller, manageable parts that are processed simultaneously.
- A **distributed computing approach**, By using the Message Passing Interface (MPI) paradigm, computations are spread out among multiple networked computers. This allows for increased scalability and efficiency, making it ideal for handling large calculations.

Each of these methods uses a numerical integration technique to estimate π by calculating the area under the curve of the function $\sqrt{1-x^2}$, which forms a quarter-circle. The precision and efficiency of these methods vary based on the number of computations and the computational resources available. This report analyzes the performance of each method, providing insights into their practical applications and limitations.

2 Solutions

2.1 Sequential Approach

The sequential approach implemented in `Pure_python.py` calculates π by numerically integrating the function $\sqrt{1-x^2}$ over the interval from 0 to 1. This function represents a quarter-circle, and the integral is approximated using the Riemann sum method, where the interval is divided into N equal parts.

The core of this computation is a simple loop that iterates N times, calculating the area of a thin rectangle at each step and summing these areas to approximate the integral. The key part of the code is the loop within the `compute_pi_sequential` function:

```
def compute_pi_sequential(N):
    delta_x = 1.0 / N
    sum = 0.0
    for i in range(N):
        xi = i * delta_x
        sum += f(xi) * delta_x
    return sum * 4
```

This method is straightforward but can be slow for very large N as each calculation depends on the completion of the previous one.

2.2 Multiprocessing Approach

The multiprocessing approach in `Multiprocess_python.py` enhances the computation of π by dividing the task among multiple processes running concurrently. This approach aims to utilize multiple cores of a processor to reduce computation time significantly compared to the sequential approach.

The function `compute_pi_parallel` sets up a pool of worker processes, assigns each a segment of the integral to compute, and collects the results. The most relevant part of this code involves setting up and managing the multiprocessing pool, and aggregating the results:

```
def compute_pi_parallel(N, num_processes):
    delta_x = 1.0 / N
    pool = mp.Pool(num_processes)
    results = [pool.apply_async(partial_sum, (i * N // num_processes, (i + 1) * N // num_pro
    pool.close()
    pool.join()
    pi_approx = sum([result.get() for result in results]) * 4
    return pi_approx
```

This segment demonstrates efficient parallel computation, allowing the function to handle larger N values more effectively by distributing the workload.

2.3 Distributed Approach

The distributed approach in `Distributed_python.py` uses the MPI framework for distributed computing across multiple networked machines, which can significantly enhance performance, especially for extremely large-scale computations.

The `compute_pi_distributed` function coordinates multiple processes across potentially multiple nodes. Each process computes a portion of the total sum independently, and the results are combined using MPI's reduce function. The relevant code section is:

```
def compute_pi_distributed(N):
    comm = MPI.COMM_WORLD
    size = comm.Get_size()
    rank = comm.Get_rank()
    delta_x = 1.0 / N
    local_sum = sum(f(i * delta_x) * delta_x for i in range(rank * local_n, ...
    total_sum = comm.reduce(local_sum, op=MPI.SUM, root=0)
    if rank == 0:
        print("Approximation of pi:", total_sum * 4)
```

This method effectively leverages the computational power of multiple nodes, reducing the time required for large computations by distributing the workload evenly.

3 Profiling

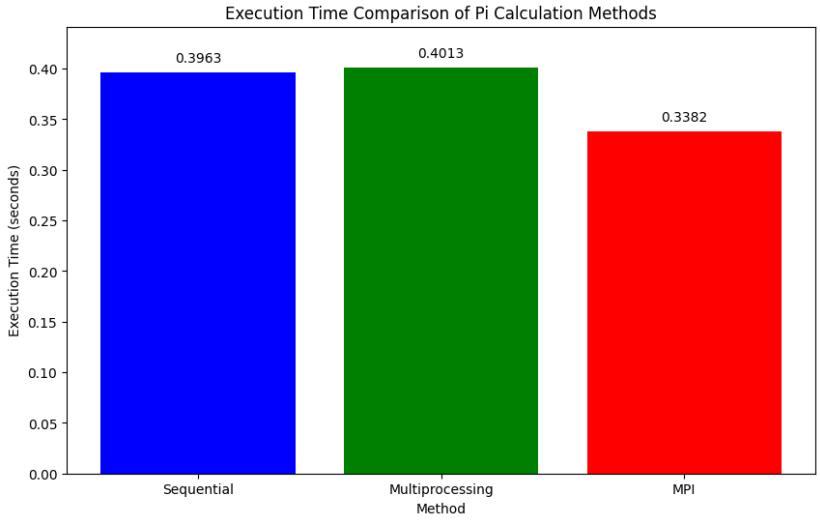
The profiling was made using the time library in each code, so the output is the actual time taken but also the approximation to pi that each one of the versions got.
For starters each one was tested on my machine, running it straight from the terminal, so I can make sure the time it takes and the approximation of pi.

```
~/Doc/UPY/7th/HPC/HPC-practice/PALLPROGRAM  HPC-practice at 12:34:03 PM
❯ python3 Multiprocess_python.py
Approximation of π: 3.141594652413769
Execution time: 0.15747308731079102 seconds

~/Doc/UPY/7th/HPC/HPC-practice/PALLPROGRAM  HPC-practice at 12:34:06 PM
❯ python3 Pure_python.py
Approximation of π: 3.1415946524138207
Execution time: 0.1858360767364502 seconds

~/Doc/UPY/7th/HPC/HPC-practice/PALLPROGRAM  HPC-practice at 12:34:10 PM
❯ mpiexec -np 4 python3 Distributed_python.py
Approximation of π: 3.1415946524138114
Execution time: 0.063778 seconds
```

Continuing with the testing, there was a problem trying to implement the mpi4py into my machine, so I decided to utilize google colab, upload my two versions, sequential and the multiprocessing, and inside the colab I wrote the MPI code. I did the testing, and for no apparent reason, the multiprocessing took more time than the pure python one. Although this could be for a lot of reasons related to the colab environment, it is clearly shown that, in my machine, it takes less time than the pure python implementation.



To clarify for the three versions the number for N was 1000000 for the whole testing so I could get an accurate value.

4 Conclusions

In this study, I explored three different approaches for approximating the value of π: sequential computation, multiprocessing, and Message Passing Interface (MPI) parallelization. The primary goal was to investigate the performance characteristics of each method and understand how they scale with increasing computational load.
To summarize, sequential computation works well for small problems, but using parallelization techniques is crucial for tackling more complex and computationally demanding tasks. Although multiprocessing had limited success in my tests, making optimizations and fine-tuning task granularity and process management could lead to improved outcomes.