UNIVERSIDAD POLITÉCNICA DE YUCATÁN

BIS Universities

Parallel Programming with Python

High performance computing

Student:
Karina Campos Almeida

Teacher:
Didier Omar Gamboa Angulo

Data engineering 7°B

April 20, 2024

Parallel Programming with Python

I. Introduction

This report details the development and analysis of three programs written in Python to calculate the value of π by numerical integration. The basis of the calculation uses Riemann sums to approximate the area of a unit quarter circle. Three different approaches are explored: a non-parallelized approach, one using parallel computing with multiprocessing, and a third using distributed parallel computing.

Parallel computing involves executing multiple tasks simultaneously to improve performance, in the context of programming, it means dividing a large task into smaller subtasks that can run concurrently [1]. This approach utilizes multiple CPU cores or processors effectively.

Distributed computing a model in which computation is spread across multiple interconnected computers to achieve a common goal [2]. Unlike traditional centralized computing, where one powerful machine handles all tasks, distributed computing decentralizes the processing load, distributing it among multiple nodes or machines that communicate and collaborate over a network.

II. Objective

Evaluate and compare the effectiveness and efficiency of three different methods of computing π: sequential computing, parallelization with multiprocessing, and distributed parallel computing using mpi4py.

III. Methods and tools

The methodology adopted and proposed by the professor to calculate the approximation of pi is:

1) Function to integrate: Define a function $f(x) = 1 - x^2$, which is the equation for the upper semicircle of a circle with radius 1 centered at the origin, since we want to calculate the area of a quarter circle, we only need to integrate this function from $x = 0$ to $x = 1$.

2) Circle Area and π: The idea is that the integral of $f(x)$ from 0 to 1 corresponds to one quarter of the area of the unit circle, which is $\frac{\pi}{4}$, so if we multiply the result of the integral by 4, we get an approximation of π.

3) Riemann Sums: To approximately calculate the integral of $f(x)$, we use a Riemann sum. This involves dividing the integration interval (from 0 to 1) into N equal subintervals of width Δx, where $\Delta x = \frac{1}{N}$. For each subinterval, we calculate the value of the function f at the point x_i, which is at the left endpoint of the subinterval (for a left-hand Riemann sum).

4) Summation Process:: We sum all the values of $f(x_i)$ and multiply by Δx to obtain the sum of the areas of the rectangles approximating the curve of the quarter circle.

The sum $\sum_{i=0}^{N-1} \Delta x f(x_i)$ is the approximation of the integral of $f(x)$, and therefore approximately equal to $\frac{\pi}{4}$.

5) Final Calculation of π: To obtain the final approximation of π, we multiply the result of the Riemann sum by 4.

The tools and technologies used include:

- **Python Programming:** Utilized for writing server and client-side scripts that manage data transactions over the network.
- **Math library:** Standard library providing mathematical functions for common operations like trigonometric functions, exponentials, logarithms, and rounding operations.
- **Multiprocessing library:** Library for creating and managing parallel processes in Python, useful for leveraging multi-core CPU systems and performing computationally intensive tasks more efficiently.
- **mpi4py library:** Python interface for the Message Passing Interface (MPI) standard, enabling communication between processes in distributed and parallel systems, crucial for high-performance computing and distributed computing.
- **timeit:** Tool for measuring the execution time of small code snippets in Python, useful for evaluating the performance of different implementation approaches and optimizing code.
- **cProfile:** Python module providing a code profiler for measuring the performance and resource utilization of Python programs, allowing identification of bottlenecks and performance optimization.

IV. Development

A. Without any parallelization

```
import math
def f(x):
    """ Function to integrate, representing the
    upper half of a unit circle. """
    return math.sqrt(1 - x**2)

def riemann_sum(n):
    """ Computes the Riemann sum approximation for
    the area under the curve. """
    total_sum = 0
    delta_x = 1.0 / n

    for i in range(n):
        x_i = i * delta_x
        total_sum += f(x_i) * delta_x

    return total_sum

def approximate_pi(n):
    """ Approximates pi using the Riemann sum
    approach with n subdivisions. """
    return 4 * riemann_sum(n)

# Example usage:
n = 10000  # Number of subdivisions
```

```
approximation = approximate_pi(n)
print(f"Approximation of pi with {n} subdivisions: {approximation}")
```

This code exemplifies a sequential computing approach to approximating π through numerical integration techniques, it utilizes a straightforward method where the Riemann sum approximation is computed sequentially using a loop. This method iterates through subdivisions of the integration interval, calculating the contribution of each subdivision to the total area under the curve, while simple to implement, this approach may not fully exploit the available computational resources.

B. Parallel computing via multiprocessing

```
import math
import multiprocessing

def f(x):
    """ Function to integrate, representing the upper half of a unit circle. """
    return math.sqrt(1 - x**2)

def partial_riemann_sum(start, end, delta_x):
    """ Computes a partial Riemann sum for the given range. """
    total_sum = 0
    x_i = start
    while x_i < end:
        total_sum += f(x_i) * delta_x
        x_i += delta_x
    return total_sum

def approximate_pi(n, num_processes):
    """ Approximates i using Riemann sums with multiprocessing. """
    delta_x = 1.0 / n
    chunk_size = n // num_processes

    # Creating a pool of processes
    pool = multiprocessing.Pool(processes=num_processes)

    results = [pool.apply_async(partial_riemann_sum,
                                (i * chunk_size * delta_x, (i + 1) * chunk_size * delta_x, delta_x))
               for i in range(num_processes)]

    # Wait for all processes to complete and sum the results
    total_sum = sum(result.get() for result in results)

    pool.close()
    pool.join()

    return 4 * total_sum

# Example usage:
if __name__ == '__main__':
    n = 10000  # Number of subdivisions
    num_processes = 6  # Number of processes
    approximation = approximate_pi(n, num_processes)
    print(f"Approximation of pi with {n} subdivisions and {num_processes} processes: {approximation}")
```

This second code showcases parallelization with multiprocessing, allowing for the distribution of the computation among multiple processes, by leveraging the multiprocessing library, this code divides the integration interval into chunks and assigns each chunk to a separate process. This enables concurrent computation across multiple CPU cores or processors, potentially resulting in faster execution, particularly on multi-core systems.

C. Distributed parallel computing

```
from mpi4py import MPI
import math
def f(x):
    """ Function to integrate, representing the upper half of a unit circle. """
    return math.sqrt(1 - x**2)

def compute_segment(start, end, delta_x):
    """ Computes the Riemann sum for a segment of the integration interval. """
    total_sum = 0
    x_i = start
    while x_i < end:
        total_sum += f(x_i) * delta_x
        x_i += delta_x
    return total_sum

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

    n = 10000  # Total number of subdivisions
    delta_x = 1.0 / n

    # Determine the portion of the interval each process will handle
    chunk_size = n // size
    start = rank * chunk_size * delta_x
    end = start + chunk_size * delta_x

    # Handle the last process to take any remaining subdivisions
    if rank == size - 1:
        end = 1.0

    local_sum = compute_segment(start, end, delta_x)

    # Gather all the local sums at the root process
    total_sum = comm.reduce(local_sum, op=MPI.SUM, root=0)

    # Calculate and print pi approximation at the root
    if rank == 0:
        pi_approx = 4 * total_sum
        print(f"Approximation of pi with {n} subdivisions across {size} processes: {pi_approx}")

if __name__ == '__main__':
    main()
```

Finally, this code illustrates distributed parallel computing using mpi4py, where the computation is spread across interconnected processes, each process handles a segment of the integration interval, and communication between processes is facilitated through the Message Passing Interface (MPI). This approach is well-suited for high-performance computing tasks that require coordination among distributed resources, making it particularly advantageous for large-scale computations.

```
envkarinacampos@192 U4 % python3 NOparallelization.py
Approximation of π with 10000 subdivisions: 3.1417914776113167
envkarinacampos@192 U4 % python3 multipr.py
Approximation of π with 10000 subdivisions and 6 processes: 3.143405579669381
envkarinacampos@192 U4 % mpiexec -n 6 python dpc.py
Approximation of π with 10000 subdivisions across 6 processes: 3.1434290332828585
```

Fig. 1. Execution codes

V. Profiling

A. Timeit

To conduct profiling, I initially employed 'timeit' to measure the execution time of each of the three previous codes, averaging over 30 executions. For all codes, 100000 subdivisions were used, and for the last two codes, 6 processes were employed, Implemented in a jupyternotebook.

```
import timeit
import NOparallelization

number_of_executions = 30
n = 100000

def run_code():
    NOparallelization.approximate_pi(n)

execution_timen = timeit.timeit(run_code, number=
    number_of_executions)
approximation = NOparallelization.approximate_pi(n)
```

```
import timeit
import multipr

number_of_executions = 30
n = 100000
num_processes = 6

def run_code():
    multipr.approximate_pi(n, num_processes)

execution_timem = timeit.timeit(run_code, number=
    number_of_executions)
```

```
import timeit
import dpc

number_of_executions = 30
n = 100000
num_processes = 6

def run_code():
    return dpce.approximate_pi(n, num_processes)

execution_timed = timeit.timeit(run_code, number=
    number_of_executions)
result = run_code()
```

After executing each of these codes in the notebook, I made a bar chart to visualize the behavior of the time.

- **Non-Parallelization**: Represented by a blue bar, this approach has the longest average execution time, recorded at approximately 1.93 seconds.
- **Multiprocessing**: Illustrated by a green bar, the average execution time for this method is markedly less, at roughly 0.78 seconds.
- **Parallel Distributed**: Showcased by a red bar, this method exhibits the shortest average execution time, which is about 0.22 seconds.

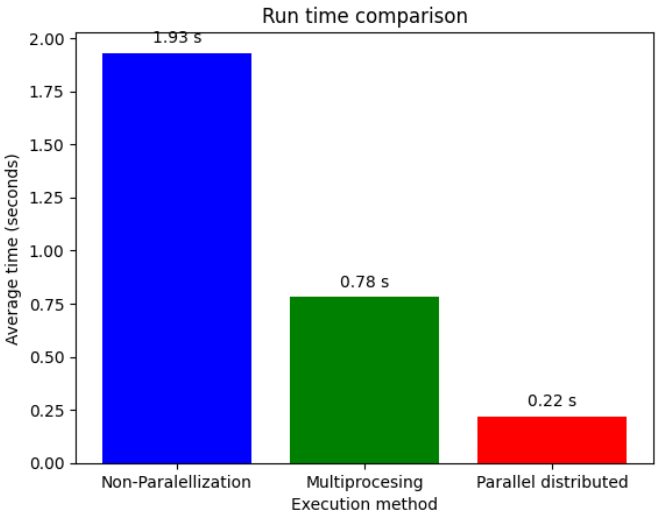

Fig. 2. Execution codes

The bar chart graphically demonstrates the advantages of employing parallel processing techniques. There is a significant reduction in execution time as the strategy shifts from non-parallelized to parallel distributed systems, underscoring that the division of tasks across multiple processors or computers can substantially enhance operational performance and efficiency.

B. Cprofile

Using cprofile within the code we can use the mprof tool in the terminal as follows:

1) mprof run myscript.py: :this line executes my script (myscript.py) while logging memory usage information at regular time intervals.

2) mprof plot: : after running my script with mprof run, this line generates a graph showing how memory usage changes over time during the execution of your script, this allows to identify memory usage spikes and possible memory leaks.

Here is the execution of this with the 3 codes:

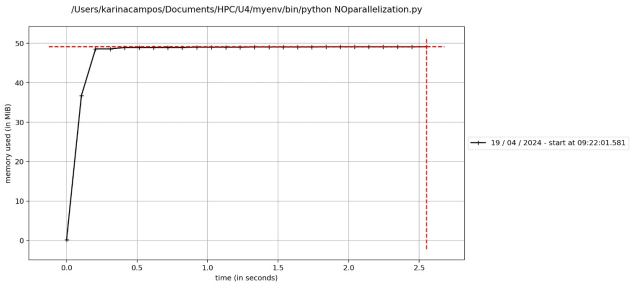

Fig. 3. Cprofile non-parallelization

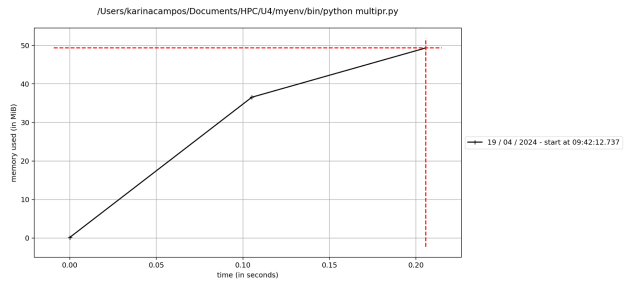

Fig. 4. Cprofile multiprocessing

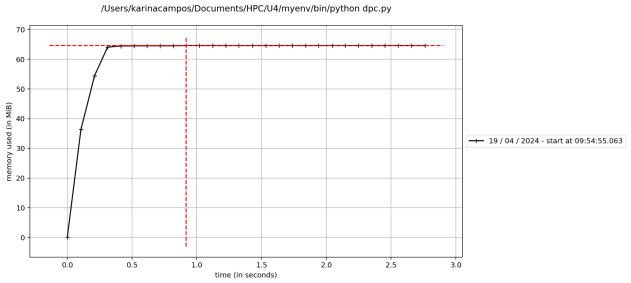

Fig. 5. Cprofile distributed parallel

Non-parallel (NOparallelization.py): Memory usage increases rapidly to just over 50 MB and then stabilizes, it seems efficient in terms of consistent memory usage after the initial allocation. This might be suitable for tasks that do not benefit significantly from parallel or distributed processing.

Multiprocessing (multipr.py): It hows a steady increase and reaches the peak of memory usage faster, suggesting a quick initialization of parallel tasks. it may be more efficient for tasks that can be easily divided into concurrent processes on a single machine, providing a speed advantage over non-parallel execution without significantly increasing memory usage.

Distributed Parallel Computing (dpc.py): Memory usage rises to a higher level (above 60 MB) and then remains stable. This could indicate that it is handling a more complex setup, possibly involving communication across a network, which is inherent to distributed computing.

VI. Conclusion

This report examines different methods for calculating π using Python, focusing on how various computing techniques affect efficiency, it covers three primary methods: sequential computing, parallel computing with multiprocessing, and distributed parallel computing using the mpi4py library.

Sequential computing, where tasks are executed one by one, is straightforward but slow as it doesn't fully utilize the computer's capabilities, in contrast, parallel computing through multiprocessing divides a task into smaller parts processed simultaneously using multiple CPU cores, greatly speeding up computation.

Distributed parallel computing extends this by linking multiple computers to work together on tasks, significantly reducing computation times, especially for large, complex tasks, the report finds that moving from sequential to distributed computing decreases computation times, highlighting the benefits of advanced techniques for handling multiple tasks simultaneously across different computers.

Overall, it emphasizes that employing parallel and distributed computing can dramatically enhance the speed and efficiency of computational tasks, crucial for data-intensive or complex calculations like estimating π. This demonstrates the importance of selecting the appropriate computing approach to optimize programming efficiency.

References

[1] "A Guide to Python Multiprocessing and Parallel Programming — SitePoint," Aug. 2022, available: https://www.sitepoint.com/python-multiprocessing-parallel-programming.

[2] J. Roller, "Exploring the Differences Between Parallel and Distributed Computing," *IEEE Computer Society*, Oct. 2023. [Online]. Available: https://www.computer.org/publications/tech-news/trends/differences-between-parallel-and-distributed-computing