



**UNIVERSIDAD  
POLITÉCNICA  
DE YUCATÁN**



# **Parallel Programming with Python**

**High performance computing**

Student:

Karina Campos Almeida

Teacher:

Didier Omar Gamboa Angulo

Data engineering 7°B

April 19, 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  $\pi$  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  $\pi$ : 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  $\pi$* : 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  $\pi$ .

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  $\Delta x$ , 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  $\Delta x$  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  $\pi$* : To obtain the final approximation of  $\pi$ , 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

```
1 import math
2 def f(x):
3     """ Function to integrate, representing the
4     upper half of a unit circle. """
5     return math.sqrt(1 - x**2)
6
7 def riemann_sum(n):
8     """ Computes the Riemann sum approximation for
9     the area under the curve. """
10    total_sum = 0
11    delta_x = 1.0 / n
12
13    for i in range(n):
14        x_i = i * delta_x
15        total_sum += f(x_i) * delta_x
16
17    return total_sum
18
19 def approximate_pi(n):
20     """ Approximates pi using the Riemann sum
21     approach with n subdivisions. """
22     return 4 * riemann_sum(n)
23
24 # Example usage:
25 n = 10000 # Number of subdivisions
```

```

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

```

This code exemplifies a sequential computing approach to approximating  $\pi$  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

```

1 import math
2 import multiprocessing
3
4 def f(x):
5     """ Function to integrate, representing the
6     upper half of a unit circle. """
7     return math.sqrt(1 - x**2)
8
9 def partial_riemann_sum(start, end, delta_x):
10     """ Computes a partial Riemann sum for the given
11     range. """
12     total_sum = 0
13     x_i = start
14     while x_i < end:
15         total_sum += f(x_i) * delta_x
16         x_i += delta_x
17     return total_sum
18
19 def approximate_pi(n, num_processes):
20     """ Approximates pi using Riemann sums with
21     multiprocessing. """
22     delta_x = 1.0 / n
23     chunk_size = n // num_processes
24
25     # Creating a pool of processes
26     pool = multiprocessing.Pool(processes=
27     num_processes)
28
29     results = [pool.apply_async(partial_riemann_sum,
30     (i * chunk_size *
31     delta_x, (i + 1) * chunk_size * delta_x, delta_x
32     ))
33     for i in range(num_processes)]
34
35     # Wait for all processes to complete and sum the
36     results
37     total_sum = sum(result.get() for result in
38     results)
39
40     pool.close()
41     pool.join()
42
43     return 4 * total_sum
44
45 # Example usage:
46 if __name__ == '__main__':
47     n = 10000 # Number of subdivisions
48     num_processes = 6 # Number of processes
49     approximation = approximate_pi(n, num_processes)
50     print(f"Approximation of pi with {n}
51     subdivisions and {num_processes} processes: {
52     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

```

1 from mpi4py import MPI
2 import math
3 def f(x):
4     """ Function to integrate, representing the
5     upper half of a unit circle. """
6     return math.sqrt(1 - x**2)
7
8 def compute_segment(start, end, delta_x):
9     """ Computes the Riemann sum for a segment of
10     the integration interval. """
11     total_sum = 0
12     x_i = start
13     while x_i < end:
14         total_sum += f(x_i) * delta_x
15         x_i += delta_x
16     return total_sum
17
18 def main():
19     comm = MPI.COMM_WORLD
20     rank = comm.Get_rank()
21     size = comm.Get_size()
22
23     n = 10000 # Total number of subdivisions
24     delta_x = 1.0 / n
25
26     # Determine the portion of the interval each
27     process will handle
28     chunk_size = n // size
29     start = rank * chunk_size * delta_x
30     end = start + chunk_size * delta_x
31
32     # Handle the last process to take any remaining
33     subdivisions
34     if rank == size - 1:
35         end = 1.0
36
37     local_sum = compute_segment(start, end, delta_x)
38
39     # Gather all the local sums at the root process
40     total_sum = comm.reduce(local_sum, op=MPI.SUM,
41     root=0)
42
43     # Calculate and print pi approximation at the
44     root
45     if rank == 0:
46         pi_approx = 4 * total_sum
47         print(f"Approximation of pi with {n}
48         subdivisions across {size} processes: {pi_approx
49         }")
50
51 if __name__ == '__main__':
52     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  $\pi$  with 10000 subdivisions: 3.1417914776113167
envkarinacampos@192 U4 % python3 multipr.py
Approximation of  $\pi$  with 10000 subdivisions and 6 processes: 3.143405579669381
envkarinacampos@192 U4 % mpiexec -n 6 python dpc.py
Approximation of  $\pi$  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.

```

1 import timeit
2 import NOparallelization
3
4 number_of_executions = 30
5 n = 100000
6
7 def run_code():
8     NOparallelization.approximate_pi(n)
9
10 execution_time = timeit.timeit(run_code, number=
    number_of_executions)
11 approximation = NOparallelization.approximate_pi(n)

```

```

1 import timeit
2 import multipr
3
4 number_of_executions = 30
5 n = 100000
6 num_processes = 6
7
8 def run_code():
9     multipr.approximate_pi(n, num_processes)
10
11 execution_time = timeit.timeit(run_code, number=
    number_of_executions)

```

```

1 import timeit
2 import dpc
3
4 number_of_executions = 30
5 n = 100000
6 num_processes = 6
7
8 def run_code():
9     return dpc.approximate_pi(n, num_processes)
10
11 execution_time = timeit.timeit(run_code, number=
    number_of_executions)
12 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.

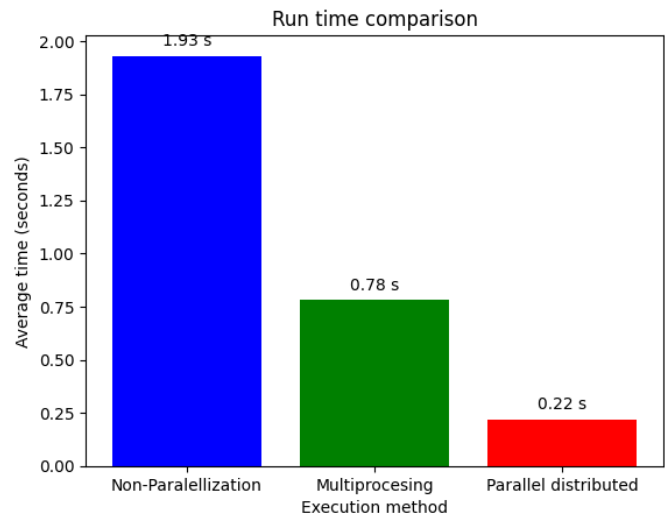


Fig. 2. Execution codes

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:

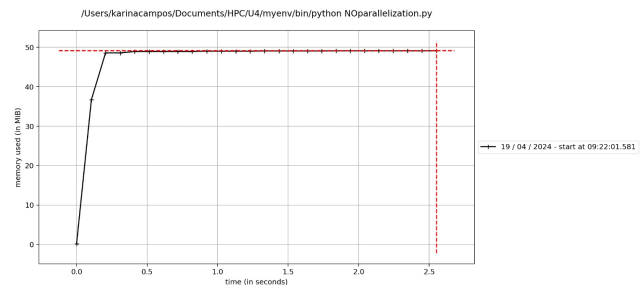


Fig. 3. Cprofile non-parallelization

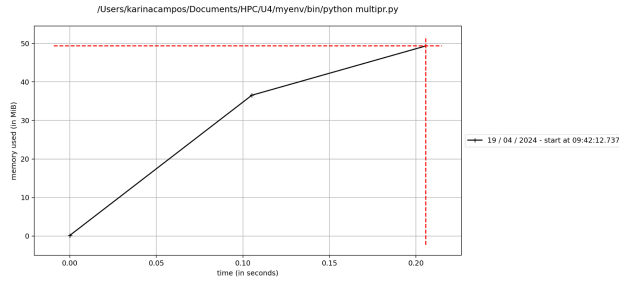


Fig. 4. Cprofile multiprocessing

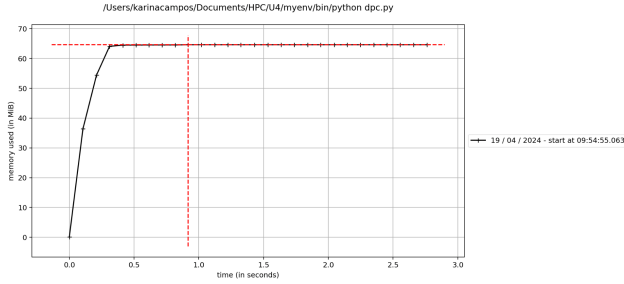


Fig. 5. Cprofile distributed parallel

**Non-parallel (NOpallelization.py):** El uso de memoria aumenta rápidamente a poco más de 50 MB y luego se estabiliza. Parece eficiente en términos de uso de memoria consistente después de la asignación inicial. Esto podría ser adecuado para tareas que no se benefician significativamente de un procesamiento paralelo o distribuido.

**Multiprocessing (multipr.py):** Este script muestra un aumento constante y alcanza el pico de uso de memoria más rápido, lo que sugiere una inicialización rápida de tareas paralelas. Puede ser más eficiente para tareas que pueden dividirse fácilmente en procesos concurrentes en una sola máquina, proporcionando una ventaja de velocidad sobre la ejecución no paralela sin aumentar significativamente el uso de memoria.

**Distributed Parallel Computing (dpc.py):** El uso de memoria aumenta a un nivel más alto (por encima de 60 MB) y luego se mantiene estable. Esto podría indicar que está manejando una configuración más compleja, posiblemente involucrando comunicación a través de una red, que es inherente a la computación distribuida. Esta podría ser la mejor opción para tareas que requieren más recursos computacionales de los que puede proporcionar un solo sistema o cuando la tarea puede acelerarse enormemente utilizando múltiples sistemas.

## VI. CONCLUSION

This report examines different methods for calculating  $\pi$  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  $\pi$ , 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>