# E3 Parallel Programming with Python

Christian Javier Can Montero
Data Engineering 7°B
Universidad Politécnica de Yucatán
Ucú, Yucatán. México
2009017@upy.edu.mx

## I. Introduction

In order to quantitatively approximate the constant $\pi$, we use a fundamental geometric property: $\pi$ is the area of a unit circle. Riemann sums can be used to estimate $\pi$ by breaking this region apart and concentrating on a quarter circle. This quarter circle's boundary is represented by the function $f(x) = \sqrt{1-x^2}$. We seek to compute $\pi/4$ as $\pi/4 \approx \sum_{i=0}^{N-1} \Delta x f(x_i)$, where $x_i = i\Delta x$ and $\Delta x = 1/N$. This approach offers a simple, yet effective, way to comprehend numerical integration.

Beyond simple mathematical interest, the computational problem of determining $\pi$ offers an opportunity to investigate the efficiency of algorithmic methods. Without using parallelization, the task starts and establishes a baseline for execution time in relation to N, the number of subdivisions. As we move forward, focus shifts to using distributed and parallel computing strategies. These cutting-edge techniques are at the forefront of increasing computational efficiency, showing a trend from simple algorithms to complex, optimal computational frameworks.

## II. Solving the problem without any parallelization

In order to determine how the execution time scales with the number of subdivisions (N), this solution will first profile the algorithm in order to approximate the value of $\pi$ by numerical integration, specifically utilising the Riemann sum approach. The selected approach takes advantage of the geometric meaning of $\pi$, which is the area of a unit circle.

### A. Source Code

- Solution:

```python
import numpy as np

# Define the function to integrate
def f(x):
    return np.sqrt(1 - x**2)

# Define the numerical integration function using Riemann sums
def compute_pi(N):
    riemann_sum = 0 # Initialize the sum
    delta_x = 1 / N # Compute  x

    # Compute the sum
    for i in range(N):
        xi = i * delta_x
        riemann_sum += f(xi) * delta_x

    # Multiply by 4 to get the approximation of

    return riemann_sum * 4

# Number of rectangles
N = 100000  # Using a large N for a better approximation

# Compute
pi_approximation = compute_pi(N)
print(f"Approximation of pi: {pi_approximation}")
```

- Output:

```
Approximation of pi: 3.1416126164019564
```

- Profiling:

```python
import time

# Function to profile the compute_pi function for different values of N
def profile_compute_pi(N_values):
    times = []
    for N in N_values:
        start_time = time.time()
        compute_pi(N)
        end_time = time.time()
        times.append(end_time - start_time)
    return times

# Different values of N to evaluate the execution time
N_values = [10**k for k in range(1, 7)]

# Profile the function
execution_times = profile_compute_pi(N_values)


def display_execution_times(N_values, execution_times):
    for N, time in zip(N_values, execution_times):
        print(f"For N = {N:,}, the execution time was approximately {time:.6f} seconds.")

display_execution_times(N_values, execution_times)
```

- Output:

```
For N = 10, the execution time was approximately 0.000061 seconds.
For N = 100, the execution time was approximately 0.000206 seconds.
For N = 1,000, the execution time was approximately 0.002332 seconds.
For N = 10,000, the execution time was approximately 0.028887 seconds.
For N = 100,000, the execution time was approximately 0.224685 seconds.
For N = 1,000,000, the execution time was approximately 2.259626 seconds.
```

## B. Key parts of the code

- Function Definition for f(x):

```python
def f(x):
    return np.sqrt(1 - x**2)
```

The equation for the upper segment of a unit circle is represented by this function f(x), which is important as the Riemann sums will utilise it to get the areas of the rectangles that are similar to the quarter-circle's area.

- Numerical Integration Function (compute_pi):

```python
def compute_pi(N):
    riemann_sum = 0
    delta_x = 1 / N
    for i in range(N):
        xi = i * delta_x
        riemann_sum += f(xi) * delta_x
    return riemann_sum * 4
```

The main function (compute_pi) divides the interval into N parts (delta_x), initialises a sum (riemann_sum), and iteratively adds the area of each rectangle under the curve f(x) from x = 0 to x = 1. This process performs the numerical integration, and the area of the quarter-circle is increased to the entire circle by multiplying at the end by 4.

- Profiling Function (profile_compute_pi):

```python
def profile_compute_pi(N_values):
    times = []
    for N in N_values:
        start_time = time.time()
        compute_pi(N)
        end_time = time.time()
        times.append(end_time - start_time)
    return times
```

The compute_pi execution time for various values of N is measured by the profile_compute_pi function. Utilising Python's time package, it collects the beginning and ending times and stores the difference in an array (times) to facilitate performance analysis by showing how increasing N impacts calculation time.

## C. Conclusion

The answer shows an easy numerical technique to approximate $\pi$. The source code is well-structured, with functions that are all well-defined, making it easier to understand and possibly modify for increased performance. A performance-driven development approach requires the profiling portion of the code, which shows how to optimise the approximation function or select the right value of N to strike a compromise between speed and accuracy.

## III. Solving the problem with parallel computing

This method uses the idea that the area under a quarter-circle of unit radius is $\pi/4$ to solve the difficulty of estimating $\pi$ using numerical integration via Riemann sums, especially with parallel computation. An approximation of $\pi$ is indirectly computed by utilising Riemann sums to approximate this region.

## A. Source Code

- Solution:

```python
import numpy as np
from math import sqrt
from multiprocessing import Pool

# Define the function to integrate
def f(x):
    return sqrt(1 - x**2)

# The integrand for the Riemann sum in the
    interval
def integrand(i, delta_x):
    x = i * delta_x
    return f(x) * delta_x

# Compute pi using numerical integration
def compute_pi(n_intervals):
    delta_x = 1 / n_intervals
    indices = range(n_intervals)
    with Pool() as pool:
        results = pool.starmap(integrand, [(i,
    delta_x) for i in indices])
    return 4 * sum(results)

# Number of intervals
n_intervals = 100000  # You can increase this
    number for a more accurate result

# Calculate pi
pi_approximation = compute_pi(n_intervals)
print(f"Approximation of pi: {pi_approximation
    }")
```

- Output:

```
Approximation of pi: 3.1416126164019564
```

- Profiling:

```python
import time

# Profiling function
def profile_computation(n_values):
    times = []
    for n_intervals in n_values:
        start_time = time.time()
        compute_pi(n_intervals)
        end_time = time.time()
        times.append(end_time - start_time)
        print(f"N = {n_intervals} took {end_time
 - start_time:.4f} seconds")
    return times

# Range of N values to profile
n_values = [1000, 10000, 100000, 1000000]

# Profile the compute_pi function
profile_computation(n_values)
```

- Output:

```
N = 1000 took 0.0279 seconds
N = 10000 took 0.0334 seconds
N = 100000 took 0.1277 seconds
N = 1000000 took 1.0931 seconds
[0.027869224548339844,
 0.03340935707092285,
 0.12768912315368652,
 1.0931406021118164]
```

## B. Key parts of the code

- The Integrating Function:

```python
def f(x):
    return sqrt(1 - x**2)
```

This is the quarter-circle's mathematical representation; this function is important since the numerical integration uses it as the integrand. To determine the correct area under the curve, it must be precisely defined.

- The Integrand Calculation for Riemann Sums:

```python
def integrand(i, delta_x):
    x = i * delta_x
    return f(x) * delta_x
```

In this case, we compute the area beneath the curve of a thin rectangle. The function computes the x position, returns the area of the rectangle at that segment, and accepts as inputs the segment index (i) and the width "delta_x" of each interval. This is the central region of the Riemann sum, where the sum of these discrete "rectangles" represents the overall area.

- Parallelizing the Computation:

```python
def compute_pi(n_intervals):
    delta_x = 1 / n_intervals
    indices = range(n_intervals)
    with Pool() as pool:
        results = pool.starmap(integrand, [(i,
 delta_x) for i in indices])
    return 4 * sum(results)
```

Parallelization takes place in this function, which splits the task of calculating the Riemann sum into manageable chunks, each of which is handled by a different processor core. A pool of worker processes is created in the context of the "with Pool() as pool" statement; the workload is then evenly distributed throughout the pool thanks to the "pool.starmap" method, which maps the integrand function to each interval. The approximation for $\pi$ is obtained by multiplying the area of the quarter-circle by 4, which increases it to the area of the whole circle.

- Profiling for Performance Analysis:

```python
def profile_computation(n_values):
    for n_intervals in n_values:
        start_time = time.time()
        compute_pi(n_intervals)
        end_time = time.time()
        print(f"N = {n_intervals} took {end_time
 - start_time:.4f} seconds")
```

This section measures the variation in execution time with varying numbers of intervals as we iterate over a range of values for N (number of intervals), track the execution time of "compute_pi(n_intervals)", and output the execution time. This analysis demonstrates the advantages of employing parallel computation for demanding numerical operations and helps us understand the performance implications of using more intervals for a more exact output.

## C. Conclusion

As the multiprocessing package allows us to use several processor cores concurrently, the solution effectively demonstrates how to use parallel processing in Python to do numerical integration and a major speedup in computation is achieved. Profiling also helps us understand the trade-offs between accuracy and calculation time. When dealing with computational issues that may be split up into separate, concurrently processing subtasks, this is a typical method.

## IV. Solving the problem with distributed parallel computing via MI4PY

The main idea behind the approach is to split up the computation of $\pi$ over several processors. Using numerical integration, each processor computes a portion of the integral that yields an area estimate for a quarter-circle. Subsequently, the output of every processor is combined to generate the ultimate approximation of $\pi$. The mpi4py package, which offers Python programs an interface to use the MPI standard for parallel computing, is used to accomplish this.

### A. *Source Code*

- Solution:

```
!pip install mpi4py

import math
from mpi4py import MPI

def calculate_part_of_pi(N, rank, size):
    # Calculates a portion of PI based on the rank of the process and total number of processes.
    # Uses a numerical integration method over a quarter circle.
    dx = 1.0 / N
    local_sum = sum(dx * math.sqrt(1 - (i * dx) ** 2) for i in range(rank, N, size))
    return local_sum

def distributed_calculate_pi(N):
    # Calculates the approximation of PI using multiple processes distributed across MPI.
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()

    local_sum = calculate_part_of_pi(N, rank, size)
    pi_approx = 4 * comm.reduce(local_sum, op=MPI.SUM, root=0)

    if rank == 0:
        print(f”Approximation of pi: {pi_approx}”)

if __name__ == ”__main__”:
    N = 10000
    distributed_calculate_pi(N)
```

- Output:

```
Approximation of pi: 3.1417914776113167
```

- Profiling:

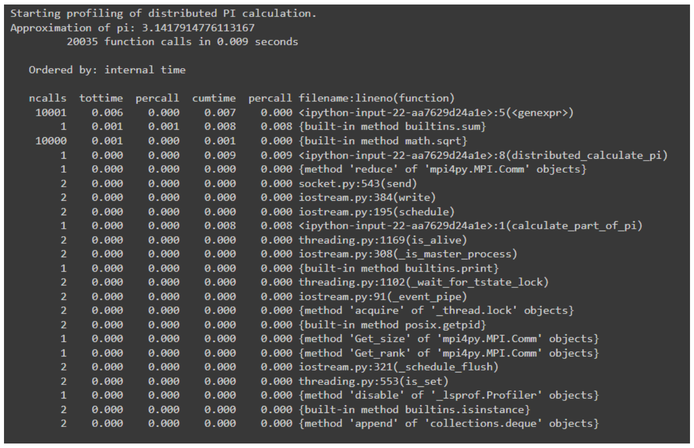
```
import cProfile

def profile_function(func, *args):
    #Profiles the specified function using cProfile, without printing results from the function itself.
    profiler = cProfile.Profile()
    profiler.enable()
    func(*args)  # Execute the function with the provided arguments.
    profiler.disable()
    if MPI.COMM_WORLD.Get_rank() == 0:  # Ensure only the root process prints the profiling stats
        profiler.print_stats(sort='time')

if __name__ == ”__main__”:
    N = 10000
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()

    if rank == 0:
        print(”Starting profiling of distributed PI calculation.”)
    profile_function(distributed_calculate_pi, N)  # Profile on all nodes, but only the root node will print profiling stats.
```

- Output:



```
Starting profiling of distributed PI calculation.
Approximation of pi: 3.1417914776113167
         20035 function calls in 0.009 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    10001    0.006    0.000    0.007    0.000 <ipython-input-22-aa7629d24a1e>:5(<genexpr>)
        1    0.001    0.001    0.008    0.008 {built-in method builtins.sum}
    10000    0.001    0.000    0.001    0.000 {built-in method math.sqrt}
        1    0.000    0.000    0.009    0.009 <ipython-input-22-aa7629d24a1e>:8(distributed_calculate_pi)
        1    0.000    0.000    0.000    0.000 {method 'reduce' of 'mpi4py.MPI.Comm' objects}
        2    0.000    0.000    0.000    0.000 socket.py:543(send)
        2    0.000    0.000    0.000    0.000 iostream.py:384(write)
        2    0.000    0.000    0.000    0.000 iostream.py:195(schedule)
        1    0.000    0.000    0.008    0.008 <ipython-input-22-aa7629d24a1e>:1(calculate_part_of_pi)
        2    0.000    0.000    0.000    0.000 threading.py:1169(is_alive)
        2    0.000    0.000    0.000    0.000 iostream.py:308(_is_master_process)
        1    0.000    0.000    0.000    0.000 {built-in method builtins.print}
        2    0.000    0.000    0.000    0.000 threading.py:1102(_wait_for_tstate_lock)
        2    0.000    0.000    0.000    0.000 iostream.py:91(_event_pipe)
        2    0.000    0.000    0.000    0.000 {method 'acquire' of '_thread.lock' objects}
        2    0.000    0.000    0.000    0.000 {built-in method posix.getpid}
        1    0.000    0.000    0.000    0.000 {method 'Get_size' of 'mpi4py.MPI.Comm' objects}
        1    0.000    0.000    0.000    0.000 {method 'Get_rank' of 'mpi4py.MPI.Comm' objects}
        2    0.000    0.000    0.000    0.000 iostream.py:321(_schedule_flush)
        2    0.000    0.000    0.000    0.000 threading.py:553(is_set)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
        2    0.000    0.000    0.000    0.000 {built-in method builtins.isinstance}
        2    0.000    0.000    0.000    0.000 {method 'append' of 'collections.deque' objects}
```

Fig. 1. Profiling output.

### B. *Key parts of the code*

- Function ”calculate_part_of_pi(N, rank, size)”:

```
def f(x):
    return sqrt(1 - x**2)
```

For a given process, this function computes a portion of the integral needed to approximate $\pi$. In order to approximate the area under a curve that depicts a quarter circle, one can use the formula $\sqrt{1-x^2}$ to calculate the height of the rectangles that are located beneath the curve.

Here, the width of each small rectangle (or interval) is indicated by $dx$, while the x-coordinate of each rectangle's midpoint is indicated by $i \times dx$. By iterating from rank to N with steps of size, the ”sum()” function combines all these little regions at intervals given to a specific process (identified by rank).

- Function ”distributed_calalculate_pi(N)”:

```python
def integrand(i, delta_x):
    x = i * delta_x
    return f(x) * delta_x
```

  Using the MPI environment, this function coordinates the distributed computing of $\pi$. All of the computation’s processes are communicated with using $MPI.COMM\_WORLD$, which facilitates data transmission and communication. The current process’s rank and the total number of processes are found using the ”comm.Get_rank()” and ”comm.Get_size()” functions, respectively. ”comm.reduce()”, an MPI collective communication action, is used to aggregate the partial results of the integral computation from each process. Since the integral only takes into consideration a quarter-circle, this total is then multiplied by 4 to get the final approximation of $\pi$, which the root procedure then prints.

- Function ”profile_function(func, *args)”:

```python
def compute_pi(n_intervals):
    delta_x = 1 / n_intervals
    indices = range(n_intervals)
    with Pool() as pool:
        results = pool.starmap(integrand, [(i,
    delta_x) for i in indices])
    return 4 * sum(results)
```

  This function creates a profiler object that tracks the execution of specified functions, encapsulating the profiling process using ”cProfile.Profile()”. The ”profiler.enable()” and ”profiler.disable()” methods are utilised to initiate and terminate profiling surrounding the execution of the targeted function (”func(*args)”). The root process is the only one that displays the profiling statistics in order to maintain output clarity and avoid data clutter. These data are sorted according to the execution time (sort=’time’), which helps detect possible performance bottlenecks by providing important information about where the majority of the computation is spent.

### C. *Conclusion*

This code is designed to run in a distributed computing environment where Python and the ”mpi4py” library are installed. To run this code, the MPI environment must be configured appropriately, which usually entails configuring a cluster or a multi-core system to have several nodes or processors. This distributed computation’s efficiency is highly dependent on the number of processes used (managed by the ”size” parameter) and the distribution of work among these processes. All in all, this code successfully illustrates how to use MPI for Python simultaneous numerical computations, offering critical performance profiling information that are essential for optimising these kinds of distributed systems.

## V. General Conclusion

From a basic non-parallelized approximation of $\pi$ to a complex distributed parallel computing approach using the ”mpi4py” package, this is, in conclusion, an evolutionary journey of computational optimisation. The paper demonstrates how large performance benefits may be obtained by putting distributed systems and parallel computing into practice, particularly for computationally demanding applications like numerical integration. Throughout the process, performance analysis and profiling tools are essential because they offer insights into the trade-offs between execution time and precision. The case study in question highlights the significance of utilising contemporary computing architectures and techniques to enhance the efficiency of solving traditional numerical problems.

## Appendix A
## Github Repository

E3 Parallel Programming with Python