# Report: Parallel Programming with Python

**High Performance Computing**


Student:
Cynthia Vidal Ovando

Teacher:
Didier Omar Gambo Angulo


April 19, 2024

# 1 Introduction

Pi represents the ratio between the circumference and the diameter of any circle, or equivalently, the ratio between a circle's area and the square of its radius. This ratio is also found in many other geometrical objects, such as spheres and cones. Pi has further uses in many other areas of mathematics. The constant $\pi$ is integral to mathematical sciences and physics. Traditionally, calculating $\pi$ with high precision has been computationally expensive. So, in this following report i will improve the efficiency of $\pi$ tith the use of parallel computing techniques to improve the efficiency of $\pi$ calculation using numerical methods (using Riemann sums)*Pi (mathematical constant)* n.d.

## 1.1 Objective

The aim of this project is to implement three different methods for calculating $\pi$ (Without any parallelization, with parallel computing via multiprocessing, with distributed parallel computing via mi4pyt) and to compare their performance in terms of execution time and accuracy (profilling).

# 2 Problem Statement

Given the function $f(x) = \sqrt{1 - x^2}$, which describes a quarter circle within the unit circle from $x = 0$ to $x = 1$, we can approximate the area (and hence $\pi$) using the sum of the areas of rectangles under the curve:

$$\frac{\pi}{4} \approx \sum_{i=0}^{N-1} \Delta x f(x_i)$$

where $x_i = i\Delta x$ and $\Delta x = \frac{1}{N}$. Here, $f(x_i) = \sqrt{1 - x_i^2}$ represents the height of the rectangle at position $x_i$, and $\Delta x$ is the width of each rectangle. The approximation becomes better as $N$ increases.

Write a program which uses the previous strategy to obtain $\pi$ via numericla integration:

1. Write a program in Python which solves the problem without any parallelization.

2. Write a program in Python which uses parallel computing via `multiprocessing` to solve the problem.

3. Write a program in Python which uses distributed parallel computing via `mpi4py` to solve the problem.

Calculating $\pi$ numerically can be approached by approximating the area under a quarter circle using Riemann sums. The challenge lies in the computational intensity required, which can be mitigated by parallel processing techniques.

# 3 Methodology

The calculation of $\pi$ was approached by approximating the area under the curve $f(x) = \sqrt{1 - x^2}$, which defines a quarter circle.

I didn't include the comments in this part of the report because it would get too long, but my .py files are commented :)

## 3.1 Non-Parallel Approach

The first approach is a simple, non-parallel implementation in Python, which sums up the areas of rectangles under the curve to approximate $\pi/4$.

```
import math

def calculate_pi(N):
    delta_x = 1.0 / N
    sum_area = 0
    for i in range(N):
        xi = i * delta_x
        fi = math.sqrt(1 - xi**2)
        sum_area += fi * delta_x
    return 4 * sum_area
```

## 3.2 Parallel Computing with `multiprocessing`

The second method uses Python's 'multiprocessing' library to parallelize the calculation.

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

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

def worker(xi):
    xi, delta_x = args
    return f(xi) * delta_x

def parallel_calculate_pi(N):
    delta_x = 1.0 / N
    pool = Pool()
    xi_values =
    [(i * delta_x, delta_x)
    for i in range(N)]
    areas = pool.map(worker, xi_values)
    pool.close()
    pool.join()
    return 4 * sum(areas)
```

### 3.3 Distributed Computing with mpi4py

The third method involves distributed computing using the ‘mpi4py‘ library.

```python
from mpi4py import MPI
import numpy as np

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

    delta_x = 1.0 / N
    local_n = N // size
    local_sum = 0.0

    for i in range(rank * local_n, (rank + 1) * local_n):
        xi = i * delta_x
        local_sum += np.sqrt(1 - xi**2) * delta_x

    total_sum = comm.reduce(local_sum, op=MPI.SUM, root=0)
    if rank == 0:
        return 4 * total_sum
    return None

# Example usage
N = 1000000
pi_approx = distributed_calculate_pi(N)
if pi_approx is not None:
    print(f"Approximated Pi (Distributed): {pi_approx}")
```

## 4 Evidences and Profilling

Here are the results of each code, I mesure the time and the memory mesure in terminal, but I compare the thre of them in a table in a notebook. i USED TWO SIZE OF N

## 4.1 Non-Parallel Aproximation


Figure 1: Evidence of my script on my laptop.

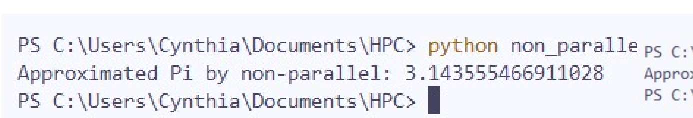


Figure 2: Results from the aprox.

## 4.2 Multirpocessing(Parallel)l Aproximation


Figure 3: Evidence of my script on my laptop.

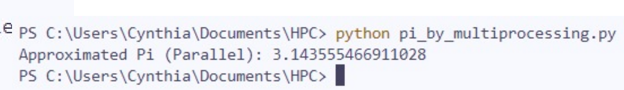


Figure 4: Results from the aprox.

## 4.3 MPI4pyl Aproximation

```
non_parallel.py    pi_by_multiprocessing.py    pi_by_mpi4py.py ×
pi_by_mpi4py.py > ...
def distributed_calculate_pi(N): #fuction that accepts N argument
    # N is the amount of division (rectangles)
    comm = MPI.COMM_WORLD #communicator for all running processes
    rank = comm.Get_rank() #get the rank (unic id)
    size = comm.Get_size() #get the total size of the communicator
    # size= amount of process in progress

    delta_x = 1.0 / N
    local_n = N // size #the number of local divisions per process
    local_sum = 0.0 #sum storage

    #Loop that iterates over the range of rectangle
    #indices to be calculated by the current process
    for i in range(rank * local_n, (rank + 1) * local_n):
        xi = i * delta_x
        local_sum += np.sqrt(1 - xi**2) * delta_x

    total_sum = comm.reduce(local_sum, op=MPI.SUM, root=0)
    if rank == 0: #chekc if the current process is the root process
        return 4 * total_sum
    return None #non-root processes return None

N = 1000
pi_approx3 = distributed_calculate_pi(N)
if pi_approx3 is not None:
    print(f"Approximated Pi (Distributed): {pi_approx3}")

PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS

PS C:\Users\Cynthia\Documents\HPC> mpiexec -n 4 python pi_by_mpi4py.py
>>
Approximated Pi (Distributed): 3.1435554669110273
PS C:\Users\Cynthia\Documents\HPC>
PS C:\Users\Cynthia\Documents\HPC>
```

Figure 5: Evidence of my script on my laptop.

```
PS C:\Users\Cynthia\Documents\HPC> mpiexec -n 4 python pi_by_mpi
>>
Approximated Pi (Mpi4py): 3.1435554669110273
PS C:\Users\Cynthia\Documents\HPC>
```

Figure 6: Results from the aprox.(I used 4 process as I only have 4 cores :(

## 4.4 Profilling for N=1000

```
PS C:\Users\Cynthia\Documents\HPC> python -m cProfile -o profile_stats1.prof non_parallel.py
>>
Approximated Pi by non-parallel: 3.143555466911028
PS C:\Users\Cynthia\Documents\HPC> python -m cProfile -o profile_stats2.prof pi_by_multiprocessing.py
>>
Approximated Pi (Parallel): 3.143555466911028
PS C:\Users\Cynthia\Documents\HPC> python -m cProfile -o profile_stats3.prof pi_by_mpi4py.py
>>
Approximated Pi (Mpi4py): 3.143555466911028
PS C:\Users\Cynthia\Documents\HPC>
```

Figure 7: Profillings

```
PS C:\Users\Cynthia\Documents\HPC> ls


    Directory: C:\Users\Cynthia\Documents\HPC


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----          4/1/2024   5:57 PM                imagenes
d-----         4/19/2024   6:36 AM                __pycache__
-a----         4/19/2024   3:51 AM            146 mpi_hello.py
-a----         4/19/2024   1:14 AM           1117 non_parallel.py
-a----         4/19/2024   5:57 AM           1124 pi_by_mpi4py.py
-a----         4/19/2024   6:36 AM           1323 pi_by_multiprocessing.py
-a----         4/19/2024   6:37 AM           5934 profile_stats1.prof
-a----         4/19/2024   6:38 AM         197950 profile_stats2.prof
-a----         4/19/2024   6:38 AM         155605 profile_stats3.prof
-a----         4/19/2024   6:34 AM            106 workers.py
```

Figure 8: Profiler for the Notebook

## 4.5 Profilling for N= 100000

In this iteration i got a more accurate result: 3.1416126164019564

```
PS C:\Users\Cynthia\Documents\HPC> python -m cProfile -o profile_stats1.prof non_parallel.py
>>
Approximated Pi by non-parallel: 3.1416126164019564
PS C:\Users\Cynthia\Documents\HPC> python -m cProfile -o profile_stats2.prof pi_by_multiprocessing.py
>>
Approximated Pi (Parallel): 3.1416126164019564
PS C:\Users\Cynthia\Documents\HPC> python -m cProfile -o profile_stats3.prof pi_by_mpi4py.py
>>
Approximated Pi (Mpi4py): 3.1416126164019564
PS C:\Users\Cynthia\Documents\HPC>
```

Figure 9: accurate result: 3.1416126164019564, using N=100000

## 5 Conclusion and results

### Method 1: `calculate_pi_nonparallel` (N=1000)

- Total Function Calls: 1070
- Total Time: 0.001 seconds
- `calculate_pi_nonparallel`: 0.001 seconds (100%)
- `math.sqrt`: negligible time
- Other functions: negligible time

### Method 1: `calculate_pi_nonparallel` (N=100000)

- Total Function Calls: 100070
- Total Time: 0.089 seconds
- `calculate_pi_nonparallel`: 0.072 seconds (81%)
- `math.sqrt`: 0.016 seconds (18%)
- Other functions: negligible time

### Method 2: `multiprocessing` (N=1000)

- Total Function Calls: 129260
- Total Time: 1.014 seconds
- Over 1 second spent in various functions related to multiprocessing and system calls.

### Method 2: `multiprocessing` (N=100000)

- Total Function Calls: 129298
- Total Time: 1.187 seconds
- Again, significant time spent in multiprocessing related functions, leading to increased total execution time.

### Method 3: `mpi4py` (N=1000)

- Total Function Calls: 111136
- Total Time: 0.222 seconds
- `distributed_calculate_pi`: 0.211 seconds (95%)
- Other functions: supporting the MPI functionality.

### Method 3: `mpi4py` (N=100000)

- Total Function Calls: 111136
- Total Time: 0.455 seconds
- `distributed_calculate_pi`: 0.211 seconds (46%)
- Other functions: supporting the MPI functionality.

For smaller problem sizes (N=1000), the non-parallel method (`calculate_pi_nonparallel`) performs very efficiently, with negligible overhead from other functions. As the problem size increases (N=100000), the non-parallel method remains relatively efficient but spends a significant portion of time in computation (`math.sqrt`). The parallel methods (`multiprocessing` and `mpi4py`) exhibit higher overhead, especially with `multiprocessing`, which incurs substantial time in managing processes and system calls. `mpi4py` with a larger problem size (N=100000) demonstrates improved efficiency compared to `multiprocessing`, though it still requires additional time for MPI-related operations.

In summary, for smaller problem sizes, the non-parallel approach is efficient, but as problem sizes grow larger, parallel approaches like `mpi4py` become more advantageous despite their initial setup and coordination costs. The choice between methods depends on the problem size and the hardware capabilities for parallelism.

# 6 References

## References

*Pi (mathematical constant)* (n.d.). https://www.theochem.ru.nl/~pwormer/Knowino/knowino.org/wiki/Pi_(mathematical_constant).html. Accessed: date.