



UNIVERSIDAD
POLITÉCNICA
DE YUCATÁN



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 π is integral to mathematical sciences and physics. Traditionally, calculating π with high precision has been computationally expensive. So, in this following report i will improve the efficiency of π tith the use of parallel computing techniques to improve the efficiency of π 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 π (Without any parallelization, with parallel computing via multiprocessing, with distributed parallel computing via mpi4py) 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 π) 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 Δx is the width of each rectangle. The approximation becomes better as N increases.

Write a program which uses the previous strategy to obtain π 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 π 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 π 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.

```

1 from multiprocessing import Pool
2 import numpy as np
3
4 def f(x):
5     return np.sqrt(1 - x**2)
6
7 def worker(xi):
8     xi, delta_x = args
9     return f(xi) * delta_x
10
11 def parallel_calculate_pi(N):
12     delta_x = 1.0 / N
13     pool = Pool()
14     xi_values =
15     [(i * delta_x, delta_x)
16     for i in range(N)]
17     areas = pool.map(worker,
18                       xi_values)
19     pool.close()
20     pool.join()
21     return 4 * sum(areas)
22
23 total_sum = comm.reduce(
24     local_sum, op=MPI.SUM, root
25     =0)
26
27 if rank == 0:
28     return 4 * total_sum
29
30 return None
31
32 # Example usage
33 N = 1000000
34 pi_approx = distributed_calculate_pi
35 (N)
36
37 if pi_approx is not None:
38     print(f"Approximated Pi (
39           Distributed): {pi_approx}")

```

3.3 Distributed Computing with mpi4py

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

```

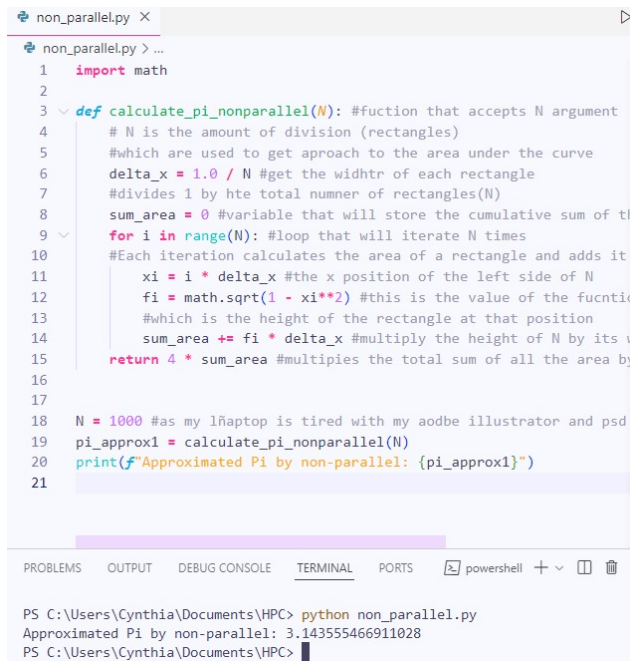
1 from mpi4py import MPI
2 import numpy as np
3
4 def distributed_calculate_pi(N):
5     comm = MPI.COMM_WORLD
6     rank = comm.Get_rank()
7     size = comm.Get_size()
8
9     delta_x = 1.0 / N
10    local_n = N // size
11    local_sum = 0.0
12
13    for i in range(rank * local_n, (
14    rank + 1) * local_n):
15        xi = i * delta_x
16        local_sum += np.sqrt(1 - xi
17                            **2) * delta_x

```

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

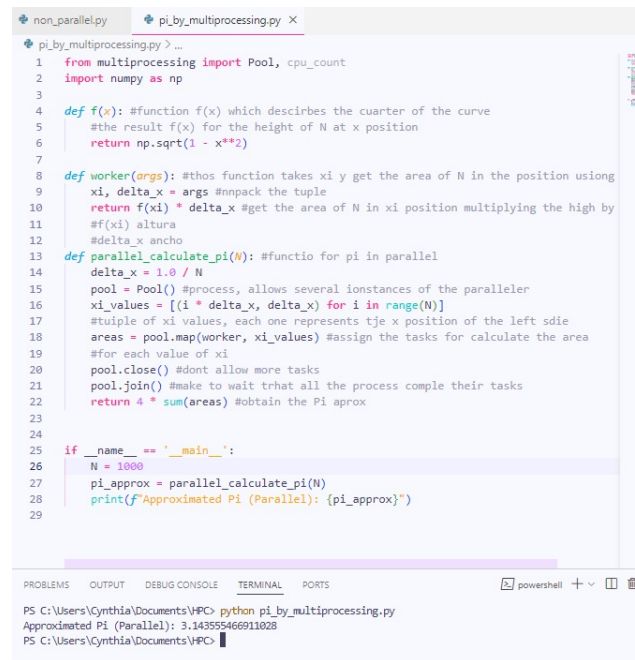


```
1 import math
2
3 def calculate_pi_nonparallel(N): #function that accepts N argument
4     # N is the amount of division (rectangles)
5     # which are used to get approach to the area under the curve
6     delta_x = 1.0 / N #get the width of each rectangle
7     # divides 1 by the total number of rectangles (N)
8     sum_area = 0 #variable that will store the cumulative sum of the area
9     for i in range(N): #loop that will iterate N times
10        # Each iteration calculates the area of a rectangle and adds it to the sum
11        xi = i * delta_x #the x position of the left side of N
12        fi = math.sqrt(1 - xi**2) #this is the value of the function
13        # which is the height of the rectangle at that position
14        sum_area += fi * delta_x #multiply the height of N by its width
15    return 4 * sum_area #multiplies the total sum of all the area by 4
16
17
18 N = 1000 #as my laptop is tired with my aodbe illustrator and psd
19 pi_approx1 = calculate_pi_nonparallel(N)
20 print(f'Approximated Pi by non-parallel: {pi_approx1}')
```

PS C:\Users\Cynthia\Documents\HPC> python non_parallel.py
Approximated Pi by non-parallel: 3.143555466911028
PS C:\Users\Cynthia\Documents\HPC>

Figure 1: Evidence of my script on my laptop.

4.2 Multiprocessing(Parallel) Aproximation



```
1 from multiprocessing import Pool, cpu_count
2 import numpy as np
3
4 def f(x): #function f(x) which describes the quarter of the curve
5     # the result f(x) for the height of N at x position
6     return np.sqrt(1 - x**2)
7
8 def worker(args): #this function takes xi y get the area of N in the position using
9     xi, delta_x = args #unpack the tuple
10    return f(xi) * delta_x #get the area of N in xi position multiplying the height by
11    # f(xi) altura
12    # delta_x ancho
13
14 def parallel_calculate_pi(N): #function for pi in parallel
15     delta_x = 1.0 / N
16     pool = Pool() #process, allows several instances of the parallel
17     xi_values = [(i * delta_x, delta_x) for i in range(N)]
18     # tuple of xi values, each one represents the x position of the left side
19     areas = pool.map(worker, xi_values) #assign the tasks for calculate the area
20     # for each value of xi
21     pool.close() #don't allow more tasks
22     pool.join() #make to wait that all the process complete their tasks
23     return 4 * sum(areas) #obtain the Pi aprox
24
25 if __name__ == '__main__':
26     N = 1000
27     pi_approx = parallel_calculate_pi(N)
28     print(f'Approximated Pi (Parallel): {pi_approx}')
```

PS C:\Users\Cynthia\Documents\HPC> python pi_by_multiprocessing.py
Approximated Pi (Parallel): 3.143555466911028
PS C:\Users\Cynthia\Documents\HPC>

Figure 3: Evidence of my script on my laptop.



```
PS C:\Users\Cynthia\Documents\HPC> python non_parallel.py
Approximated Pi by non-parallel: 3.143555466911028
PS C:\Users\Cynthia\Documents\HPC>

PS C:\Users\Cynthia\Documents\HPC> python pi_by_multiprocessing.py
Approximated Pi (Parallel): 3.143555466911028
PS C:\Users\Cynthia\Documents\HPC>
```

Figure 2: Results from the aprox.

Figure 4: Results from the aprox.

4.3 MPI4pyl Aproximation

```

4 def distributed_calculate_pi(N): #fuction that accepts N argument
5     # N is the amount of division (rectangles)
6     comm = MPI.COMM_WORLD #communicator for all running processes
7     rank = comm.Get_rank() #get the rank (unic id)
8     size = comm.Get_size() #get the total size of the communicator
9     # size= amount of process in progress
10
11     delta_x = 1.0 / N
12     local_n = N // size #the number of local divisions per process
13     local_sum = 0.0 #sum storage
14
15     #Loop that iterates over the range of rectangle
16     #indices to be calculated by the current process
17     for i in range(rank * local_n, (rank + 1) * local_n):
18         xi = i * delta_x
19         local_sum += np.sqrt(1 - xi**2) * delta_x
20
21     total_sum = comm.reduce(local_sum, op=MPI.SUM, root=0)
22     if rank == 0: #check if the current process is the root process
23         return 4 * total_sum
24     return None #non-root processes return None
25
26 N = 1000
27 pi_approx3 = distributed_calculate_pi(N)
28 if pi_approx3 is not None:
29     print(f"Approximated Pi (Distributed): {pi_approx3}")
30

```

```

PS C:\Users\Cynthia\Documents\HPC> mpiexec -n 4 python pi_by_mpi4py.py
>>
Approximated Pi (Distributed): 3.143554669110273
PS C:\Users\Cynthia\Documents\HPC>

```

Figure 5: Evidence of my script on my laptop.

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

4.4 Profiling for N=1000

```

PS C:\Users\Cynthia\Documents\HPC> python -m cProfile -o profile_stats1.prof non_parallel.py
>>
Approximated Pi by non-parallel: 3.14355466911028
PS C:\Users\Cynthia\Documents\HPC> python -m cProfile -o profile_stats2.prof pi_by_multiprocessing.py
>>
Approximated Pi (Parallel): 3.14355466911028
PS C:\Users\Cynthia\Documents\HPC> python -m cProfile -o profile_stats3.prof pi_by_mpi4py.py
>>
Approximated Pi (/mpi4py): 3.14355466911028
PS C:\Users\Cynthia\Documents\HPC>

```

Figure 7: Profilings

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

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 Profiling 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](https://www.theochem.ru.nl/~pwormer/Knowino/knowino.org/wiki/Pi_(mathematical_constant).html). Accessed: date.