




04/18/2024

# HIGH PERFORMANCE PYTHON

## FINAL PROJECT

MARIANO MIRANDA SANCHEZ  
DIDIER OMAR GAMBOA ANGULO  
Universidad Politécnica de Yucata



# INTRODUCTION

In this work, we use numerical integration techniques to estimate the value of  $\pi$ , the famous mathematical ratio.  $\pi$  is commonly recognized as the area enclosed by a circle having a radius of one unit. Our approximation procedure is dividing this unit circle into quarters and calculating the area using a Riemann sums-based technique.

To describe the arc of this quarter circle, we employ the function  $f(x) = \sqrt{1-x^2}$  which holds true for the interval from 0 to 1 along the x-axis. The Riemann sum approximation entails dividing this interval into  $NN$  segments of equal length, evaluating the function  $f(x)$  at these divisions to determine the heights of the corresponding rectangles, and then calculating the sum of their areas. The formula encapsulating this operation is:

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

Our goal with this report is to delve into various computational strategies for this approximation, detail their execution in programming constructs, and evaluate their computational efficiency by measuring execution times relative to different choices of  $N$ .

## Solutions and Profiling

### 1. Without any parallelization

Code

```
import math
import cProfile

def calculate_pi(N):
    dx = 1.0 / N
    sum = 0.0
    for i in range(N):
        xi = i * dx
        sum += dx * math.sqrt(1 - xi**2)
    return 4 * sum

N = 10000 # número de subdivisiones

cProfile.run('calculate_pi(N)')
```

The relevant part of the implementation of the code is the 'calculate\_pi' function which computes the sum of areas of rectangles under the quarter circle curve of  $f(x) = \sqrt{1-x^2}$  in the 'for' application, that is the loop.

```
for i in range(N):
    xi = i * dx
    sum += dx * math.sqrt(1 - xi**2)
return 4 * sum
```

Iteratively calculates the area of each rectangle and accumulates the total area. After completing the loop, the sum is multiplied by 4 to estimate  $\pi$ .

The simplicity of this solution is the strong part, as it makes it easily understandable and modifiable. But, the sequential structure means that it does not use the parallel processing capabilities of the CPU, which could result in longer computation times for large values of N (I think that happened before in another code and almost killed my lap).

## Profiling

```
10004 function calls in 0.007 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      1   0.006    0.006    0.007    0.007 <ipython-input-45-5980700521e2>:4(calculate_pi)
      1   0.000    0.000    0.007    0.007 <string>:1(<module>)
      1   0.000    0.000    0.007    0.007 {built-in method builtins.exec}
    10000  0.001    0.000    0.001    0.000 {built-in method math.sqrt}
      1   0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

## 2. Multiprocessing solution

### Code

```
import math
from multiprocessing import Pool
import cProfile

def f(i, N):
    dx = 1.0 / N
    xi = i * dx
    return dx * math.sqrt(1 - xi**2)

def parallel_calculate_pi(N):
    with Pool() as pool:
        results = pool.starmap(f, [(i, N) for i in range(N)])
    return 4 * sum(results)

N = 10000

def main():
    print("Aproximation with multiprocessing:", parallel_calculate_pi(N))

cProfile.run('main()')
```

The relevant section of the code is in the 'Pool()' of the 'parallel\_calculate\_pi' that focus in the processes.

```
with Pool() as pool:
    results = pool.starmap(f, [(i, N) for i in range(N)])
    return 4 * sum(results)
```

The block uses the 'Pool()' class to manage multiple processes and the starmap method to distribute the workload, calculating the area of each rectangle corresponding to a quarter-circle segment. The distribution is performed by passing the function 'f' and an iterable of arguments [(i, N) for i in the range of N], which instructs each process to compute a portion of the Riemann sums in parallel. The function 'f' represents a task that each process performs. It computes the area of a rectangle for a given segment 'i', which is part of the Riemann sum. Once all processes are completed, the results are summed using 'sum(results)', and the total is multiplied by 4 to approximate  $\pi$ .

## Profiling

```
Aproximation de pi con multiprocessing: 3.1417914776113167
943 function calls in 0.055 seconds

Ordered by: standard name
```

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
12	0.000	0.000	0.000	0.000	<frozen importlib._bootstrap>:404(parent)
1	0.000	0.000	0.054	0.054	<ipython-input-42-b12c0a06e735>:10(parallel_calculate_pi)
1	0.004	0.004	0.004	0.004	<ipython-input-42-b12c0a06e735>:12(<listcomp>)
1	0.000	0.000	0.055	0.055	<ipython-input-42-b12c0a06e735>:17(main)
1	0.000	0.000	0.055	0.055	<string>:1(<module>)
2	0.000	0.000	0.000	0.000	__init__.py:219(_acquireLock)
2	0.000	0.000	0.000	0.000	__init__.py:228(_releaseLock)
5	0.000	0.000	0.000	0.000	_weakrefset.py:39(_remove)
5	0.000	0.000	0.000	0.000	_weakrefset.py:86(add)
6	0.000	0.000	0.000	0.000	connection.py:117(__init__)
6	0.000	0.000	0.000	0.000	connection.py:130(__del__)
3	0.000	0.000	0.000	0.000	connection.py:134(_check_closed)
3	0.000	0.000	0.000	0.000	connection.py:142(_check_writable)
3	0.000	0.000	0.000	0.000	connection.py:181(send_bytes)
6	0.000	0.000	0.000	0.000	connection.py:360(_close)
3	0.000	0.000	0.000	0.000	connection.py:365(_send)
3	0.000	0.000	0.000	0.000	connection.py:390(_send_bytes)
3	0.000	0.000	0.000	0.000	connection.py:516(Pipe)
3	0.000	0.000	0.001	0.000	context.py:110(SimpleQueue)
1	0.000	0.000	0.024	0.024	context.py:115(Pool)
9	0.000	0.000	0.000	0.000	context.py:187(get_context)
6	0.000	0.000	0.000	0.000	context.py:197(get_start_method)
1	0.000	0.000	0.000	0.000	context.py:237(get_context)
2	0.000	0.000	0.019	0.010	context.py:278(_Popen)
6	0.000	0.000	0.000	0.000	context.py:65(Lock)
13	0.000	0.000	0.003	0.000	iostream.py:195(schedule)
4	0.000	0.000	0.000	0.000	iostream.py:308(_is_master_process)
4	0.000	0.000	0.000	0.000	iostream.py:321(_schedule_flush)

### 3. *Mpi4py*

#### Code

```
from mpi4py import MPI
import math
import cProfile

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

    profiler = cProfile.Profile()
    profiler.enable()

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

    profiler.disable()
    profiler.print_stats(sort='time')

    if rank == 0:
        print(f"Aproximación with mpi4py: {pi_approx}")

if __name__ == "__main__":
    N = 10000
    if MPI.COMM_WORLD.Get_rank() == 0:
        print("Starting distributed PI calculation with profiling")
    distributed_calculate_pi(N)
```

‘Calculate\_part\_of\_pi()’ It is the detailed task that each process in the distributed system performs. It takes the process's rank and the total size of the communicator to calculate the correct segments.

The 'range' function in the list comprehension within this function, 'range(rank, N, size)', ensures that each process calculates a unique part of the  $\pi$  approximation without any overlap. This is an important part of the distribution where tasks are divided among processors.

```
def calculate_part_of_pi(N, rank, size):
    dx = 1.0 / N
    local_sum = sum(dx * math.sqrt(1 - (i * dx) ** 2) for i in range(rank, N, size))
    return local_sum
```

'MPI.COMM\_WORLD.reduce()' collects the partial sums from all processes and aggregates them with a sum operation '(MPI.SUM)', a critical point in the distribution where the individual calculations of all processes are combined to obtain the final result.

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

    profiler = cProfile.Profile()
    profiler.enable()

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

    profiler.disable()
    profiler.print_stats(sort='time')
```

## Profiling

```
Starting distributed PI calculation with profiling
20005 function calls in 0.011 seconds

Ordered by: internal time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
10001   0.008    0.000    0.009    0.000 <ipython-input-46-cc25e2be2c4d>:7(<genexpr>)
1       0.001    0.001    0.011    0.011 {built-in method builtins.sum}
10000   0.001    0.000    0.001    0.000 {built-in method math.sqrt}
1       0.000    0.000    0.000    0.000 {method 'reduce' of 'mpi4py.MPI.Comm' objects}
1       0.000    0.000    0.011    0.011 <ipython-input-46-cc25e2be2c4d>:5(calculate_part_of_pi)
1       0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Aproximación with mpi4py: 3.1417914776113167

## Conclusion

The first solution proved to be fast and efficient for smaller datasets where the overhead of parallel processing is not justified, having an execution time of 0.007 sec with 10,004 calls. Most of the execution time is spent calculating the Riemann sums. the calculate\_pi function itself, which makes the calculation of the Riemann sums for the  $\pi$  approximation the most computationally intensive part of this code, of these function calls, 10,000 were calls to the math.sqrt function. Each of these calls represents a calculation of the square root needed for the area of a single rectangle in the Riemann sum.

The second method had a time of 0.055 seconds with 943 calls being the slowest for the dataset size of  $N=10,000$  due to the overhead of managing multiple processes. Although the actual computation is fast, the setup and synchronisation of the processes is time consuming, taking away the benefits of parallel execution at this scale. There were fewer total function calls than were used in the first method. This reduction is likely due to the clustering and distribution of the workload across multiple processes. The result concludes that, although the multiprocessing approach can benefit from parallel execution, there is a significant overhead associated with the initiation and management of multiple processes. This overhead may not help in performance benefits for computations that are not sufficiently large or complex.

The third method had a time of 0.011 seconds with 20,005 function calls, offering better performance than multiprocessing but is slightly slower than the sequential method for this problem size. The solution shows the potential for efficient scaling across multiple nodes, especially as computational demand increases. The communication overhead that would be computational time is minimal, which is beneficial for larger distributed systems.

As a general conclusion the use and performance of each code varies, for small datasets the sequential approach is preferable due to its simplicity and lower overhead, being in some ways the most "optimal". For larger datasets and scalable systems, the MPI approach gives better results, particularly in environments where tasks can be distributed among many nodes, giving a good balance between computational efficiency and overhead management. Finally multiprocessing can be beneficial for intermediate scales where multiple cores are available but setting up a distributed environment is not feasible or necessary and would be the worst case of the 3 cases given.