Name: Rosemary Fernandes **Course: BDCC**

UID: 225020 **Rollno: 13**

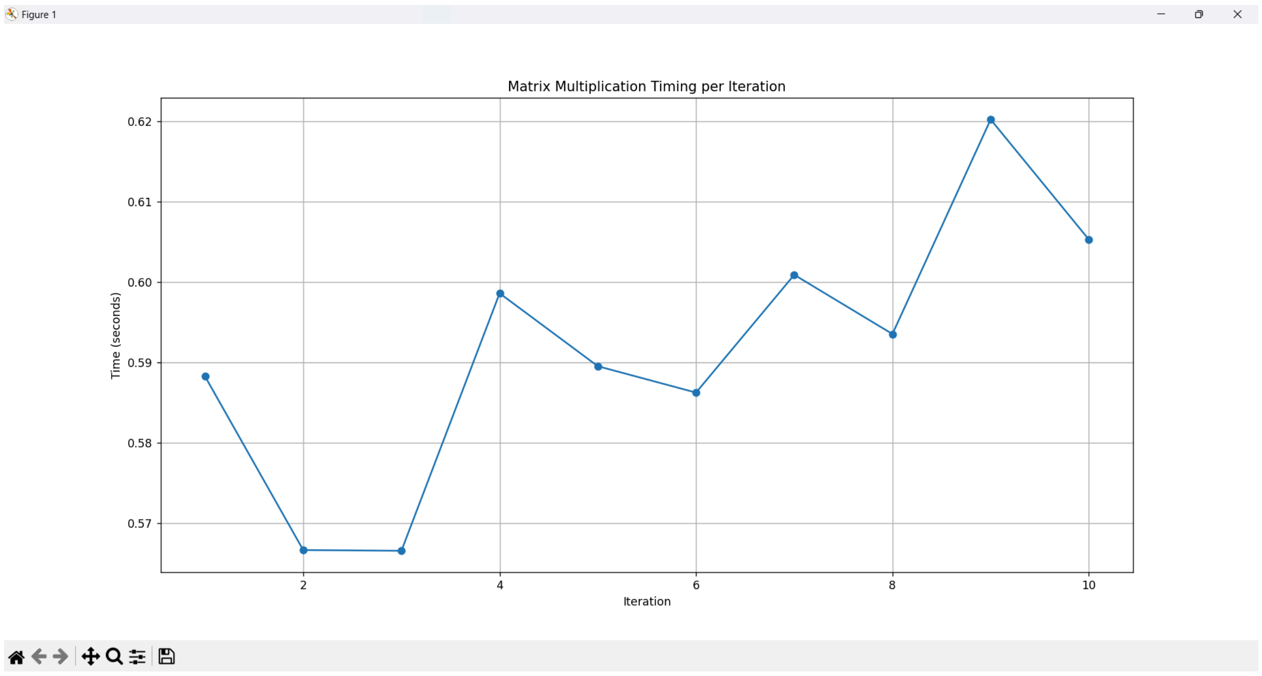
PART 1: ***Performance Comparison of Python Implementations***

The analysis is based on the performance data of matrix multiplication timings across multiple iterations for three Python implementations: **CPython**, **PyPy**, and **Jython**.

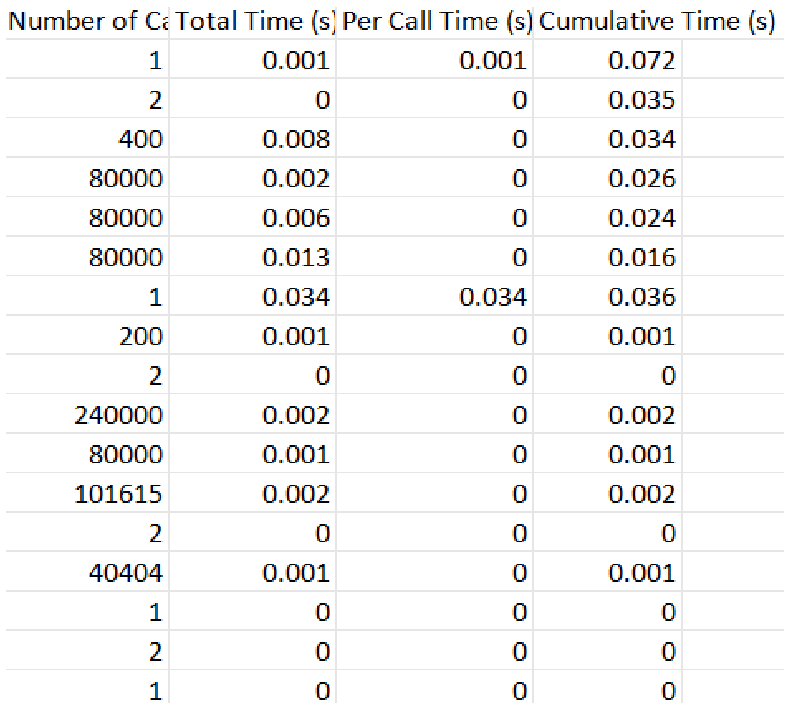
For each iteration, time taken is recorded and plotted against it in the graphs below:

TEST OUTPUT for PYTHON(CPython):

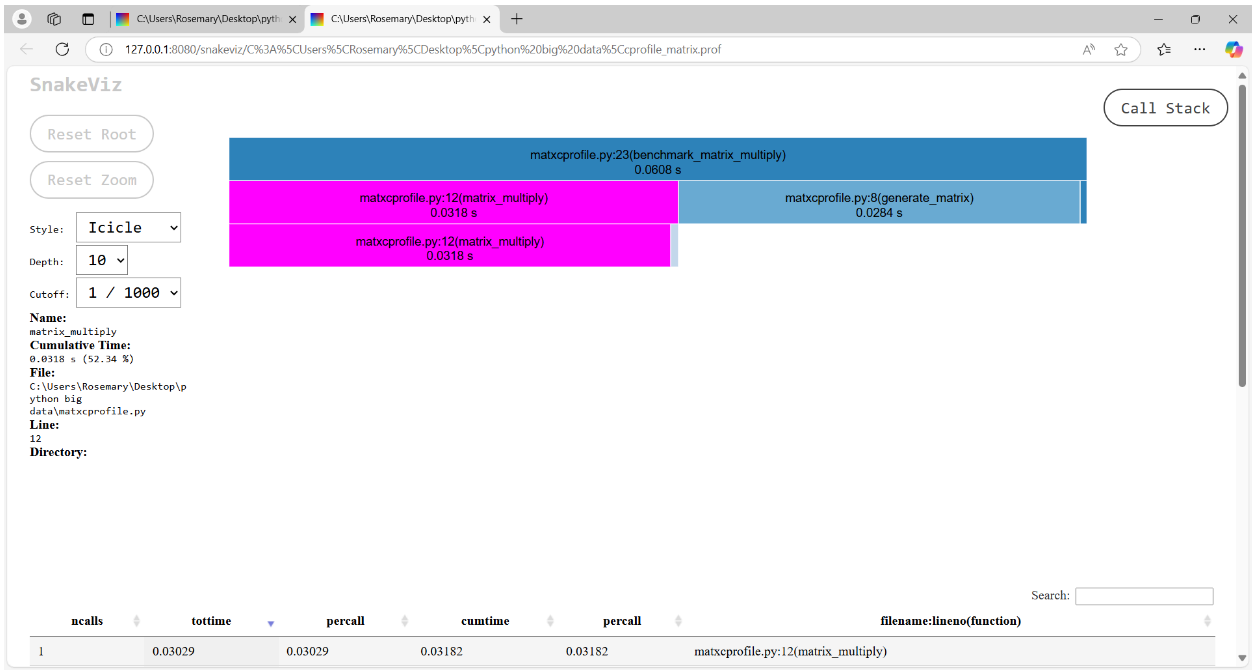
1.1:



1.2:



Number of Ca	Total Time (s	Per Call Time (s)	Cumulative Time (s)
1	0.001	0.001	0.072
2	0	0	0.035
400	0.008	0	0.034
80000	0.002	0	0.026
80000	0.006	0	0.024
80000	0.013	0	0.016
1	0.034	0.034	0.036
200	0.001	0	0.001
2	0	0	0
240000	0.002	0	0.002
80000	0.001	0	0.001
101615	0.002	0	0.002
2	0	0	0
40404	0.001	0	0.001
1	0	0	0
2	0	0	0
1	0	0	0



C:\Users\Rosemary\Desktop\pyth
C:\Users\Rosemary\Desktop\pyth
127.0.0.1:8080/snakeviz/C%3A%5CUsers%5CRosemary%5CDesktop%5Cpython%20big%20data%5Cprofile_matrix.prof
SnakeViz
Call Stack
Reset Root
Reset Zoom

matxcprofile.py:23(benchmark_matrix_multiply) 0.0608 s
matxcprofile.py:12(matrix_multiply) 0.0318 s
matxcprofile.py:8(generate_matrix) 0.0284 s
matxcprofile.py:12(matrix_multiply) 0.0318 s

Style: Icicle
Depth: 10
Cutoff: 1 / 1000

Name: matrix_multiply
Cumulative Time: 0.0318 s (52.34 %)
File: C:\Users\Rosemary\Desktop\python big data\matxcprofile.py
Line: 12
Directory:

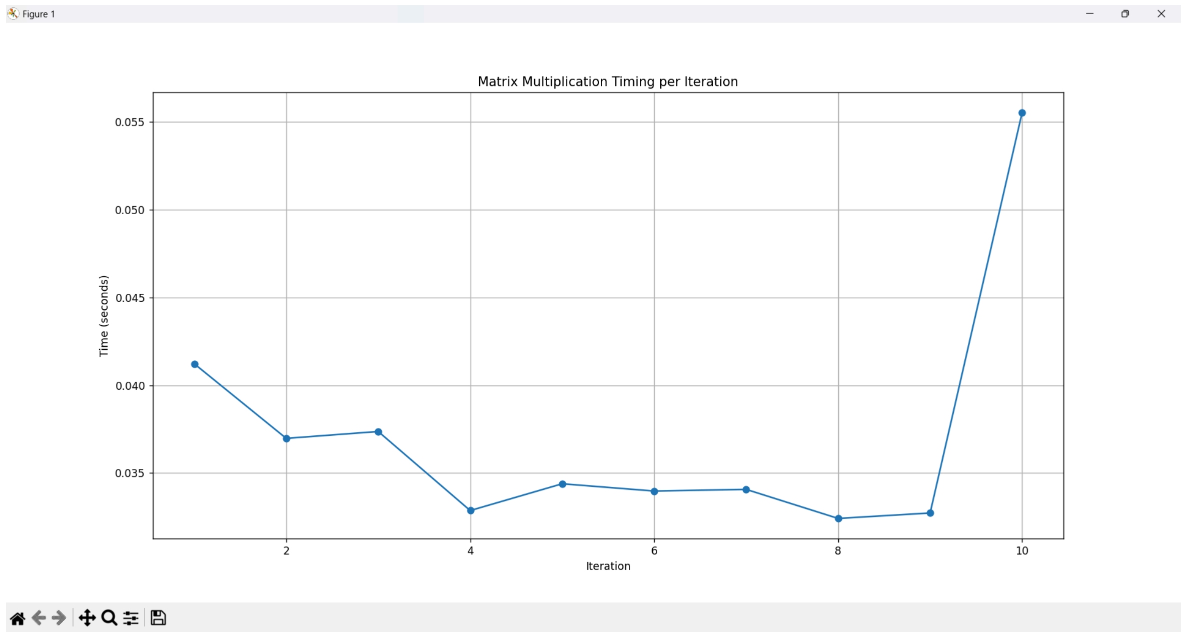
Search:

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.03029	0.03029	0.03182	0.03182	matxcprofile.py:12(matrix_multiply)

The graph shows noticeable fluctuations in the time taken per iteration.
Time values vary around **0.57 to 0.62 seconds**, indicating moderate consistency but no clear trend in execution speed.

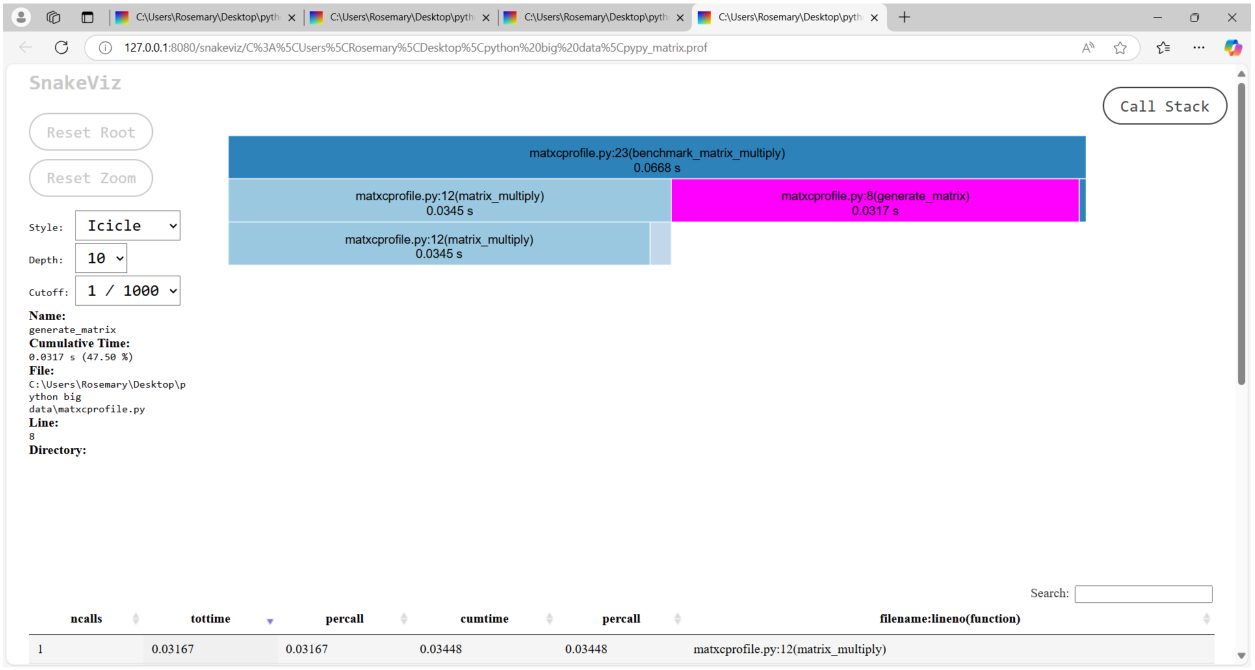
TEST OUTPUT for PYPY:

1.3:



1.4:

Number of Calls	Total Time (s)	Per Call Time (s)	Cumulative Time (s)
1	0.001	0.001	0.07
2	0	0	0.033
400	0.01	0	0.033
80000	0.002	0	0.023
80000	0.006	0	0.021
80000	0.01	0	0.013
1	0.034	0.034	0.036
200	0.001	0	0.001
2	0	0	0
240000	0.002	0	0.002
80000	0.001	0	0.001
101488	0.002	0	0.002
2	0	0	0
40404	0.001	0	0.001
1	0	0	0
2	0	0	0
1	0	0	0


SnakeViz
Reset Root
Reset Zoom
Call Stack
matxcprofile.py:23(benchmark_matrix_multiply) 0.0668 s
matxcprofile.py:12(matrix_multiply) 0.0345 s
matxcprofile.py:8(generate_matrix) 0.0317 s
matxcprofile.py:12(matrix_multiply) 0.0345 s

Style: Icicle

Depth: 10

Cutoff: 1 / 1000

Name: generate_matrix

Cumulative Time: 0.0317 s (47.50 %)

File: C:\Users\Rosemary\Desktop\python big data\matxcprofile.py

Line: 8

Directory:

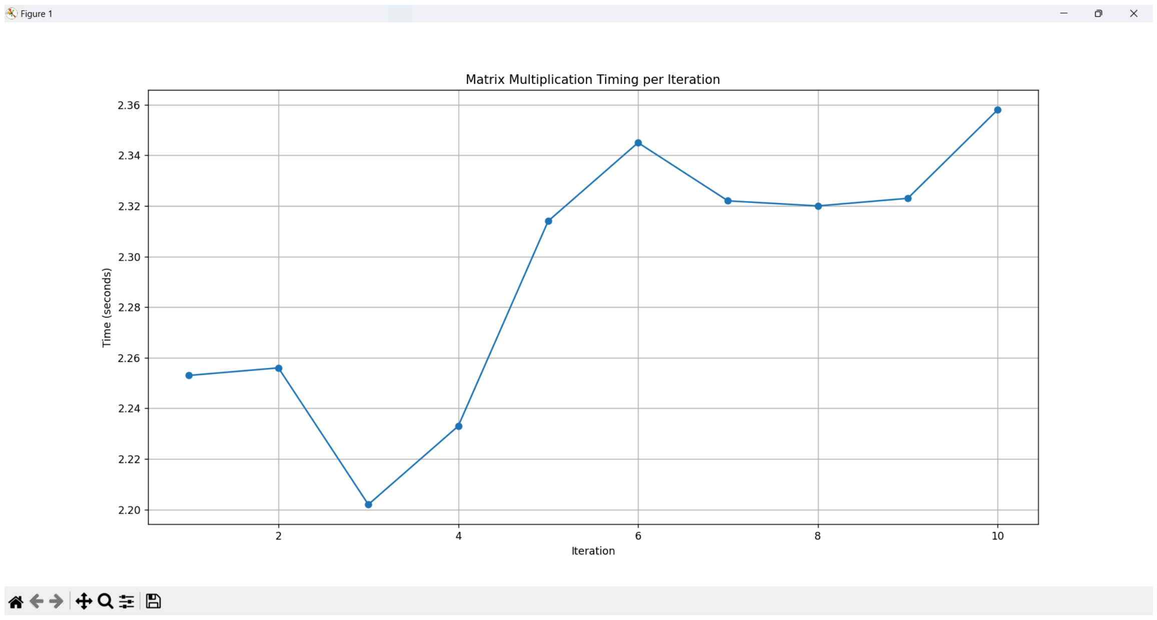
Search:

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.03167	0.03167	0.03448	0.03448	matxcprofile.py:12(matrix_multiply)

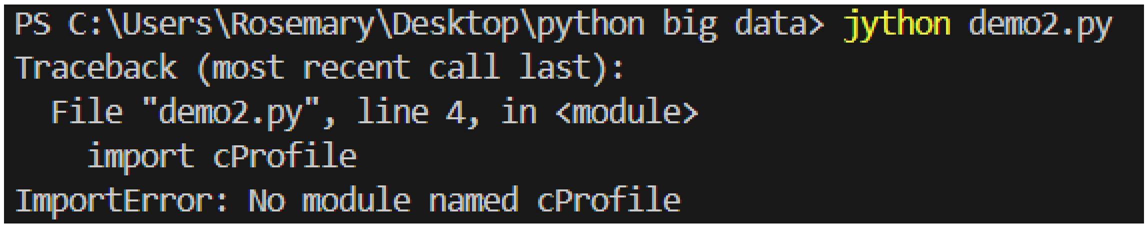
The graph for PyPy indicates a significant reduction in processing time per iteration, hovering between **0.02 and 0.04 seconds**.
The performance is consistently better compared to CPython, with smaller variations across iterations.

<u>FOR JYTHON :</u>

1.5:



1.6:



```
PS C:\Users\Rosemary\Desktop\python big data> jython demo2.py
Traceback (most recent call last):
  File "demo2.py", line 4, in <module>
    import cProfile
ImportError: No module named cProfile
```

Jython does not natively support essential libraries like matplotlib, as a result, you need to save the time recorded into a text file and then run default python to visualize it.
The recorded data, saved externally and visualized, shows times ranging from **2.20 to 2.36 seconds**, much slower than CPython and PyPy.

Overall speed comparison:

(refer images 1.1, 1.3 and 1.5 above)
CPython executes at a relatively slower speed compared to PyPy but remains reliable as the standard implementation. PyPy's Just-In-Time (JIT) compilation significantly enhances speed for repetitive operations like matrix multiplication.Jython's reliance on Java Virtual Machine (JVM) introduces overhead, making it less suitable for compute-heavy tasks.

cProfile comparison:

(refer images 1.2, 1. 4 and 1.6 above)
The matrix_multiply function:

- CPython: 0.037 seconds cumulative time.
- PyPy: 0.049 seconds cumulative time.

CPython appears more efficient in handling list comprehensions and basic numerical operations compared to PyPy for this workload.
Both implementations involved significant calls to randint and related methods (randrange and _randbelow_with_getrandbits).
PyPy generally handled these operations slightly faster, as shown by the lower cumulative times for random-related functions.

For this specific task, **CPython outperformed PyPy** in terms of matrix multiplication speed, which is unusual but could be attributed to:

1. The **small matrix size** or lack of optimizable hot spots, reducing the benefits of PyPy's JIT compilation.
2. **Profiling overhead in PyPy**, which may skew timings due to the JIT's additional operations during profiling.

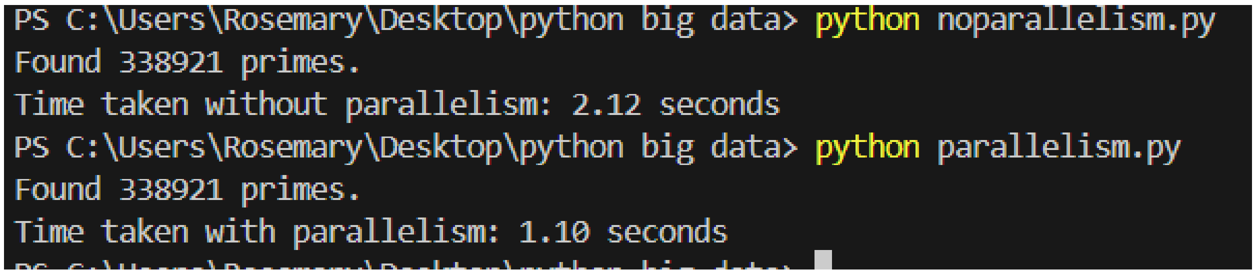
Jython does not support cProfile because cProfile relies on CPython-specific APIs written in C, which are absent in Jython.This limitation restricts its use for detailed performance analysis using Python's native profiling tools.

Overview:

- **For high-performance needs**, especially heavy computations or repetitive operations, **PyPy** is the optimal choice.
- **For general-purpose scripting**, **CPython** is the standard and most versatile implementation.
- **Jython** is not recommended for intensive computations,due to the limitations of JVM and lack of compatibility with key Python libraries, making it less practical for modern Python projects.

PART 2: ***Algorithm Parallelization***

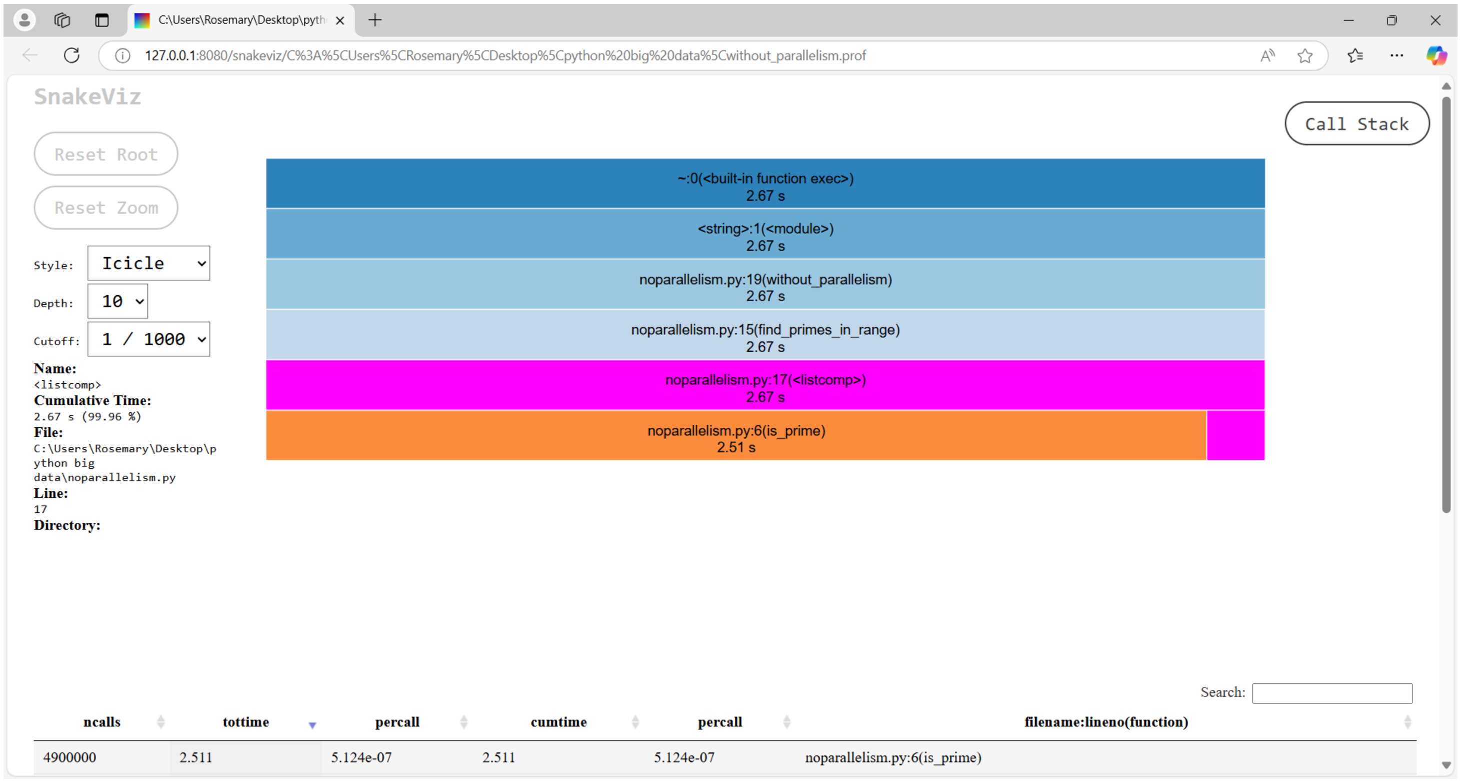
Program: Find prime numbers between a given range



```
PS C:\Users\Rosemary\Desktop\python big data> python noparallelism.py
Found 338921 primes.
Time taken without parallelism: 2.12 seconds
PS C:\Users\Rosemary\Desktop\python big data> python parallelism.py
Found 338921 primes.
Time taken with parallelism: 1.10 seconds
```

(csv files attached for cprofile results, for only table data)

Range Processing Complexity(For no parallelism):



C:\Users\Rosemary\Desktop\pyth
127.0.0.1:8080/snakeviz/C%3A%5CUsers%5CRosemary%5CDesktop%5Cpython%20big%20data%5Cwithout_parallelism.prof
SnakeViz
Reset Root
Reset Zoom
Call Stack
~:0(<built-in function exec>)
2.67 s
<string>:1(<module>)
2.67 s
noparallelism.py:19(without_parallelism)
2.67 s
noparallelism.py:15(find_primes_in_range)
2.67 s
noparallelism.py:17(<listcomp>)
2.67 s
noparallelism.py:6(is_prime)
2.51 s
Search:

Style: Icicle
Depth: 10
Cutoff: 1 / 1000

Name:
<listcomp>
Cumulative Time:
2.67 s (99.96 %)
File:
C:\Users\Rosemary\Desktop\python big data\noparallelism.py
Line:
17
Directory:

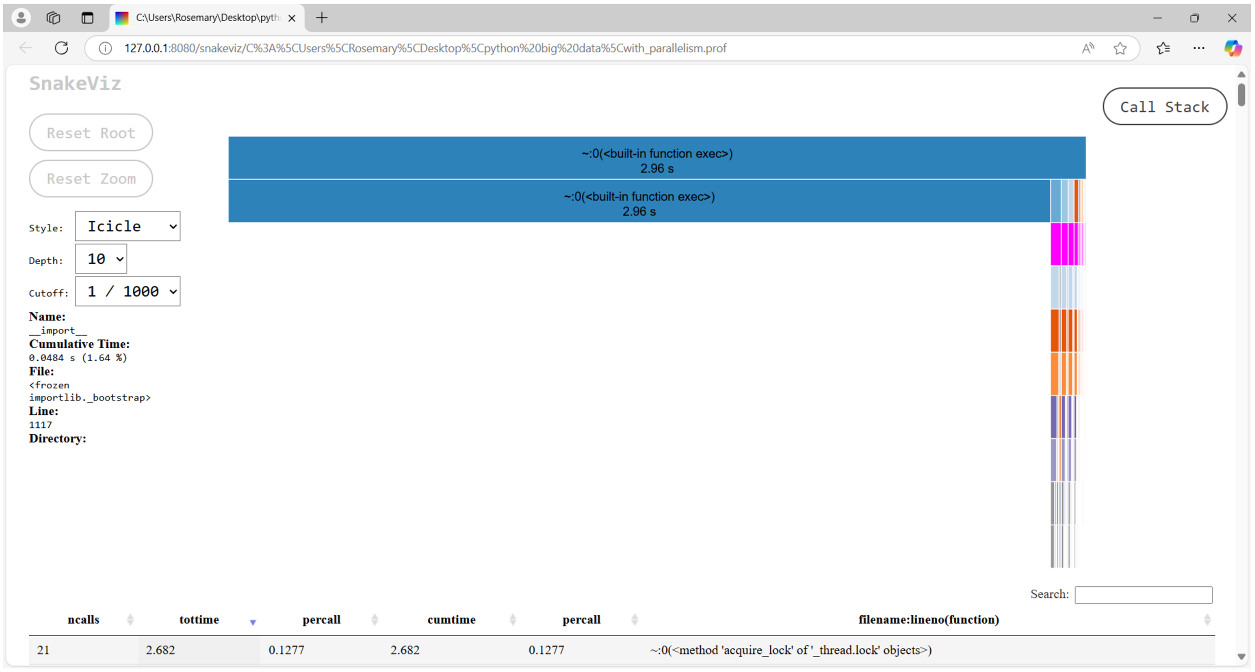
ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
4900000	2.511	5.124e-07	2.511	5.124e-07	noparallelism.py:6(is_prime)

- For a range of size N (numbers between start and end), the total complexity is: **O(N/sqrt(M))**

- Where:
  - N is the number of integers in the range.
  - M is the average size of the numbers in the range.

P.T.O.

Parallelized Version Complexity:


C:\Users\Rosemary\Desktop\pyth
127.0.0.1:8080/snakeviz/C%3A%5CUsers%5CRosemary%5CDesktop%5Cpython%20big%20data%5Cwith_parallelism.prof
SnakeViz
Reset Root
Reset Zoom
Call Stack
~:0(<built-in function exec>)
2.96 s
~:0(<built-in function exec>)
2.96 s
Search:

Style: Icicle

Depth: 10

Cutoff: 1 / 1000

Name:
__import__
Cumulative Time:
0.0484 s (1.64 %)
File:
<frozen importlib._bootstrap>
Line:
1117
Directory:

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
21	2.682	0.1277	2.682	0.1277	~:0(<method 'acquire_lock' of '_thread.lock' objects>)

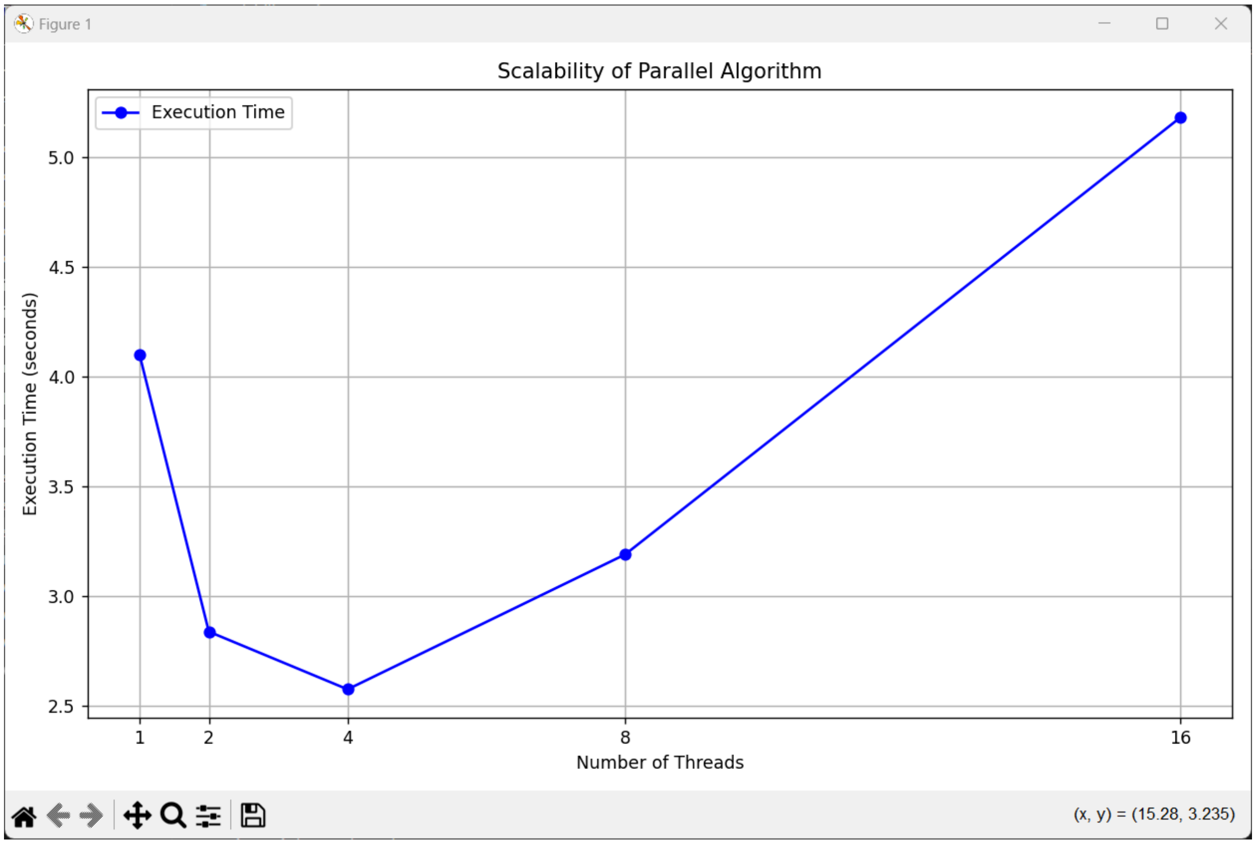
- The range is divided into T chunks for T workers (threads or processes).
- Each chunk processes approximately **N/T** numbers, resulting in:

$$O\left(\frac{N \cdot \sqrt{M}}{T}\right)$$

- Overhead: Additional time for dividing the range, inter-process communication, and combining results.

P.T.O.

Scalability analysis:



Number of Threads	Execution Time (seconds)	
1	4.1	
2	2.836	
4	2.573	
8	3.188	
16	5.179	

Big O:

$$O(f(n)/p) + O(c)$$

Where:

- p: Number of threads.

- c: Overhead from thread communication and synchronization.

Overview:

The results suggest diminishing returns after 4 threads, indicating increasing overhead as the thread count rises.

The algorithm scales well up to **4 threads**, achieving a speedup of **1.60x** (39% of ideal scalability). Beyond 4 threads, performance degrades due to overhead.

For **8 threads**, speedup drops to **1.28x**, highlighting inefficiencies in thread management.

For **16 threads**, the performance is worse than sequential execution, likely due to thread contention and synchronization bottlenecks.