Python Performance Analysis:

A Comparative Study with Parallelization

Rahul Gupta

TY BSc IT

225039

26

1. Performance Comparison of Python Implementations

Introduction

- In this section, we will analyze the performance of four Python implementations — CPython, PyPy, Jython, and MicroPython — by running a factorial calculation for numbers from 1 to 100.
- We will compare the execution time for each implementation, display the output, and present a comparative analysis using a table and a matplotlib graph.

Python Flavors Used

1. CPython
   - Description: The default and most widely used Python implementation written in C. It is known for its extensive library support and compatibility.
   - Execution Time: 0.001529 seconds
   - Output:

```
rahul@pop-os:~/Documents/bdcc_assignment/task_1/python-flavor$ source cpython_env/bin/activate
(cpython_env) rahul@pop-os:~/Documents/bdcc_assignment/task_1/python-flavor$ python3 ../code/factorial.py
Factorial calculated of each number from 1 to  100
Execution time: 0.001529 seconds
(cpython_env) rahul@pop-os:~/Documents/bdcc_assignment/task_1/python-flavor$ deactivate
rahul@pop-os:~/Documents/bdcc_assignment/task_1/python-flavor$
```

2. PyPy
   - Description: A fast, JIT (Just-In-Time) compiled Python implementation that focuses on speed and efficiency.
   - Execution Time: 0.015046 seconds
   - Output:

```
rahul@pop-os:~/Documents/bdcc_assignment/task_1/python-flavor$ source pypy_env/bin/activate
(pypy_env) rahul@pop-os:~/Documents/bdcc_assignment/task_1/python-flavor$ pypy3 ../code/factorial.py
Factorial calculated of each number from 1 to  100
Execution time: 0.015046 seconds
(pypy_env) rahul@pop-os:~/Documents/bdcc_assignment/task_1/python-flavor$ deactivate
rahul@pop-os:~/Documents/bdcc_assignment/task_1/python-flavor$
```

3. Jython
    - Description: A Python implementation written in Java that runs on the JVM (Java Virtual Machine), allowing interaction with Java libraries.
    - Execution Time: 0.041000 seconds
    - Output:

```
rahul@pop-os:~/Documents/bdcc_assignment/task_1/python-flavor$ source jython_env/bin/activate
(jython_env) rahul@pop-os:~/Documents/bdcc_assignment/task_1/python-flavor$ jython ../code/factorial.py
('Factorial calculated of each number from 1 to ', 100)
Execution time: 0.041000 seconds
(jython_env) rahul@pop-os:~/Documents/bdcc_assignment/task_1/python-flavor$ deactivate
rahul@pop-os:~/Documents/bdcc_assignment/task_1/python-flavor$ 
```

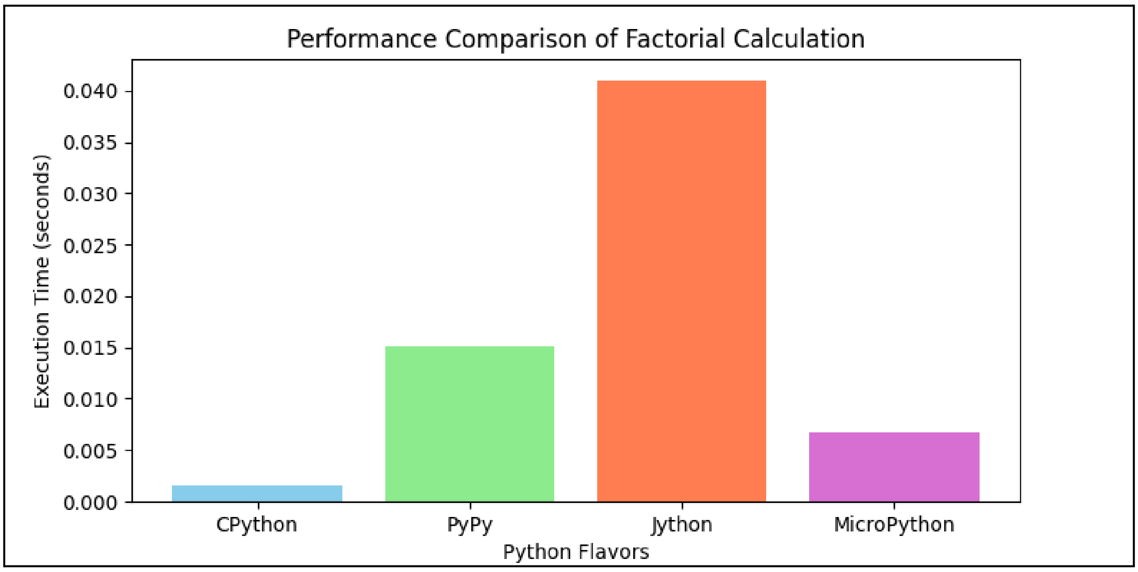
4. Micropython
    - Description: A lightweight Python implementation designed for microcontrollers and embedded systems.
    - Execution Time: 0.006702 seconds
    - Output:

```
rahul@pop-os:~/Documents/bdcc_assignment/task_1/python-flavor$ source micropython_env/bin/activate
(micropython_env) rahul@pop-os:~/Documents/bdcc_assignment/task_1/python-flavor$ micropython ../code/factorial.py
Factorial calculated of each number from 1 to  100
Execution time: 0.006702 seconds
(micropython_env) rahul@pop-os:~/Documents/bdcc_assignment/task_1/python-flavor$ deactivate
rahul@pop-os:~/Documents/bdcc_assignment/task_1/python-flavor$ 
```

<u>Table: Execution Time of Factorial Calculation (1-100)</u>

Python Flavor	**Execution Time (seconds)**
CPython	0.001529
PyPy	0.015046
Jython	0.041000
Micropython	0.006702

Performance Comparison Graph



Analysis and Inference

1. Fastest Implementation: CPython is the fastest, taking 0.001529 seconds to execute.
2. Slowest Implementation: Jython is the slowest, taking 0.041000 seconds to execute.
3. Reason for Differences:
    - CPython: Fastest as it is the standard implementation with C-level optimizations.
    - PyPy: Slightly slower than CPython despite JIT compilation, likely due to the overhead of initial JIT setup for a small recursive task like factorial.
    - Jython: Slow because it runs on the Java Virtual Machine (JVM), adding overhead compared to direct C execution.
    - MicroPython: Designed for embedded systems with limited processing power, so it is slower than CPython but faster than Jython in this test.

Key Takeaways

1. Virtual Environments ensure that each Python flavor is isolated for testing.
2. The factorial.py script calculates the factorial and measures execution time for each Python flavor.
3. Matplotlib visualizes Python performance differences.
4. CPython is the best overall for general computation, while PyPy might outperform it for larger or more complex iterative/recursive tasks.

2. Algorithm Parallelization

Introduction

- This project demonstrates how parallelization can improve the performance of a computationally intensive algorithm.
- We use a prime number finder as the test algorithm and analyze its execution with various optimizations.
- The analysis is performed through profiling, multi-threading, and multiprocessing to measure execution time improvements.
- The project aims to showcase the benefits of parallelization using the following steps:
  - cProfile Analysis to identify time-consuming parts of the algorithm.
  - Timeit Analysis to measure the precise execution time of the function.
  - Multi-threading to see how multiple threads can speed up execution.
  - Multi-processing to explore how dividing the workload across processors impacts performance.
- We also evaluate the execution times for 2, 4, 6, and 8 processors, visualize the results, and provide inferences from the findings.

Standard Version

- File Name: main.py
- Description:
  - This is the standard version of the prime finder program.
  - It finds all prime numbers in the range 1 to 1,000,000 sequentially without any parallelization.
  - The execution time is measured using the time module.
- Execution Time: 5.641796 seconds
- Output:

```
rahul@pop-os:~/Documents/bdcc_assignment/task_2$ python3 main.py
Total number of primes in range(1, 1000000): 78498
Execution time: 5.641796 seconds
rahul@pop-os:~/Documents/bdcc_assignment/task_2$
```

cProfile Analysis

- File Name: cprofile_version.py
- Description:
  - This file uses the cProfile module to identify which parts of the algorithm consume the most time.
  - It helps in finding bottlenecks in the algorithm.
- Execution Time: 5.824426 seconds
- Output:

```
rahul@pop-os:~/Documents/bdcc_assignment/task_2$ python3 cprofile_version.py
Total number of primes in range(1, 1000000): 78498
Execution time: 5.824426 seconds
         2078499 function calls in 20.047 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.001    0.001   20.047   20.047 <string>:1(<module>)
        1    2.905    2.905   20.046   20.046 main.py:14(find_primes_in_range)
   999999   14.527    0.000   16.989    0.000 main.py:5(is_prime)
        1    0.000    0.000   20.047   20.047 {built-in method builtins.exec}
   999998    2.461    0.000    2.461    0.000 {built-in method math.sqrt}
    78498    0.153    0.000    0.153    0.000 {method 'append' of 'list' objects}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Timeit Analysis

- File Name: timeit_version.py
- Description:
  - This file uses the timeit module to measure the execution time of the find_primes_in_range() function with high precision.
- Execution Time: 5.487442 seconds
- Output:

```
rahul@pop-os:~/Documents/bdcc_assignment/task_2$ python3 timeit_version.py
Total number of primes in range(1, 1000000): 78498
Execution time: 5.580345 seconds
Execution time using timeit: 5.487442 seconds
rahul@pop-os:~/Documents/bdcc_assignment/task_2$
```

Multi-Threading Implementation

- File Name: multithreading_version.py
- Description:
  - This version divides the range of numbers into smaller parts and assigns them to multiple threads.
- Execution Time: 6.550390 seconds
- Output:

```
rahul@pop-os:~/Documents/bdcc_assignment/task_2$ python3 multithreading_version.py
Total number of primes in range(1, 1000000): 78498
Execution time: 5.866524 seconds
Total Number of Primes: 78498
Execution time with multithreading: 6.550390 seconds
rahul@pop-os:~/Documents/bdcc_assignment/task_2$
```

Multiprocessing

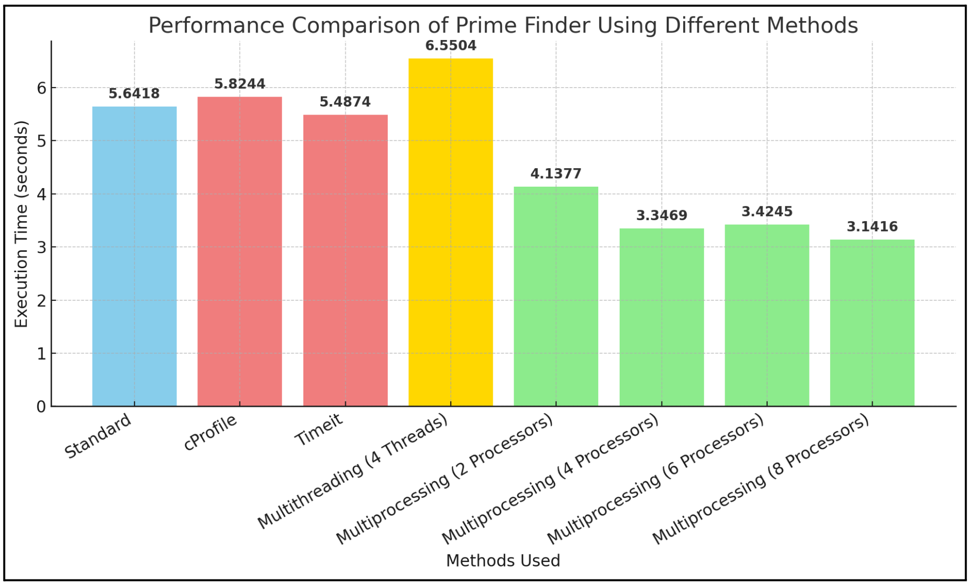
- File Name: multiprocessing_version.py
- Description:
  - This version divides the task among multiple processors using the multiprocessing module.
  - The program executes the task on 2, 4, 6, and 8 processors.
- Execution Time: 5.943514 seconds
- Output:

```
rahul@pop-os:~/Documents/bdcc_assignment/task_2$ python3 multiprocessing_version.py
Total number of primes in range(1, 1000000): 78498
Execution time: 5.943514 seconds
Processors: 2, Total Primes: 78498, Execution Time: 4.137661 seconds
Processors: 4, Total Primes: 78498, Execution Time: 3.346900 seconds
Processors: 6, Total Primes: 78498, Execution Time: 3.424486 seconds
Processors: 8, Total Primes: 78498, Execution Time: 3.141621 seconds
rahul@pop-os:~/Documents/bdcc_assignment/task_2$
```

Table: Execution Time of Parallel Prime Finder Using Different Methods

File Name	Program Name / Method Used	Number of Processors / Threads	Execution Time (seconds)
main.py	Standard / Basic version	N/A (Single Process)	5.641796
cprofile_version.py	cProfile Analysis	N/A (Single Process)	5.824426
timeit_version.py	Timeit Analysis	N/A (Single Process)	5.487442
multithreading_version.py	Multithreading	4 Threads	6.550390
multiprocessing_version.py	Multiprocessing	2 Processors	4.137661
multiprocessing_version.py	Multiprocessing	4 Processors	3.346900
multiprocessing_version.py	Multiprocessing	6 Processors	3.424486
multiprocessing_version.py	Multiprocessing	8 Processors	3.14.1621

Performance Comparison Graph



Analysis and Inference

- Fastest Method: Multiprocessing with 4 processors (3.34s).
- Slowest Method: Multithreading due to Python's GIL, resulting in a time of 6.55s.
- Best Method for Parallel Execution: Multiprocessing (more cores = faster execution, but optimal at 4).
- Reason for Differences:
  - cProfile: Added profiling overhead increases the execution time.
  - Timeit: Accurate measurement method, but no optimization applied.
  - Multithreading: Inefficient for CPU-bound tasks due to the GIL.
  - Multiprocessing: Best choice since processes run on separate cores.

Key Takeaways

- Profiling with cProfile identifies bottlenecks but increases execution time slightly.
- Timeit provides a clean, accurate measurement of execution time.
- Multithreading is ineffective for CPU-bound tasks due to the GIL (useful for I/O-bound tasks).
- Multiprocessing is the best choice for CPU-bound tasks, with 4 processors being the optimal choice.
- Optimal Strategy: Profile code with cProfile, time it with timeit, and execute it with multiprocessing (4 processors) for the best overall performance.