

Python Performance Analysis: A Comparative Study with Parallelisation

Authors: Kriti Bhattacharya- 225049(31), Shruti Desikan- 225057(35)

Aim:

To evaluate the performance of various Python implementations (e.g., CPython, PyPy, Jython) by measuring processing speed and analyzing the impact of parallelisation on algorithm execution.

Objectives:

1. Comparing Python implementations
2. Investigating the effects of parallelisation

Note: Refer to all programs uploaded to git repository:

🌐 [GitHub - Kritiin5/Python-Performance-Analysis](#): This study aims to evaluate the performance of va...

1. Comparing Python Implementation

Evaluating the performance of different Python implementations by calculating factorials of large numbers and measuring their execution times.

The program to calculate factorial of a number (serial_fact.py) was implemented using CPython, PyPy and Jython.

The program visualise.py is used to create a chart illustrating the performance differences.

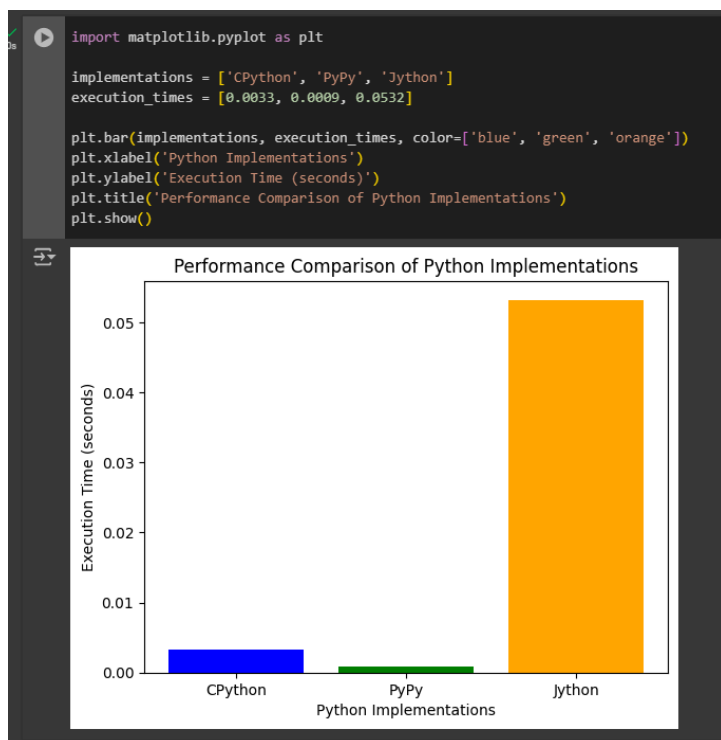
Output:

```
benchmark.py > ...
1  import time
2
3  def benchmark():
4      start = time.time()
5      data = sorted([i for i in range(100000, 0, -1)])
6      end = time.time()
7      return end - start
8
9  if __name__ == "__main__":
10     runs = 5
11     times = [benchmark() for _ in range(runs)]
12     average_time = sum(times) / runs
13     print("Execution times: {}".format(times))
14     print("Average execution time: {}".format(average_time))
15
```

PROBLEMS OUTPUT DEBUG CONSOLE **TERMINAL** PORTS

/Users/prasenjithbhattacharya/.zprofile:14: unmatched "
● prasenjitbhattacharya@Prasenjits-MacBook-Air BDCC % python3 benchmark.py
Execution times: [0.013968944549560547, 0.000659942626953125, 0.0006220340728759766, 0.0006189346313476562, 0.0006480216979980469]
Average execution time: 0.0033035755157470705
● prasenjitbhattacharya@Prasenjits-MacBook-Air BDCC % pypy benchmark.py
Execution times: [0.0021729469299316406, 0.0006499290466308594, 0.0006229877471923828, 0.0006299018859863281, 0.0006480216979980469]
Average execution time: 0.0009447574615478515
● prasenjitbhattacharya@Prasenjits-MacBook-Air BDCC % jython benchmark.py
Execution times: [0.06099987030029297, 0.04999995231628418, 0.05000019073486328, 0.05099987983703613, 0.054000139236450195]
Average execution time: 0.053200006485
○ prasenjitbhattacharya@Prasenjits-MacBook-Air BDCC %

Chart Comparison:



Result:

The performance of the different Python implementations is displayed in terms of execution time.

Execution time for different Python implementations were:

1. CPython: 0.0033 seconds
2. PyPy: 0.0009 seconds
3. Jython: 0.0532 seconds

The execution time for PyPy was the least, therefore making it the fastest, followed by CPython and Jython.

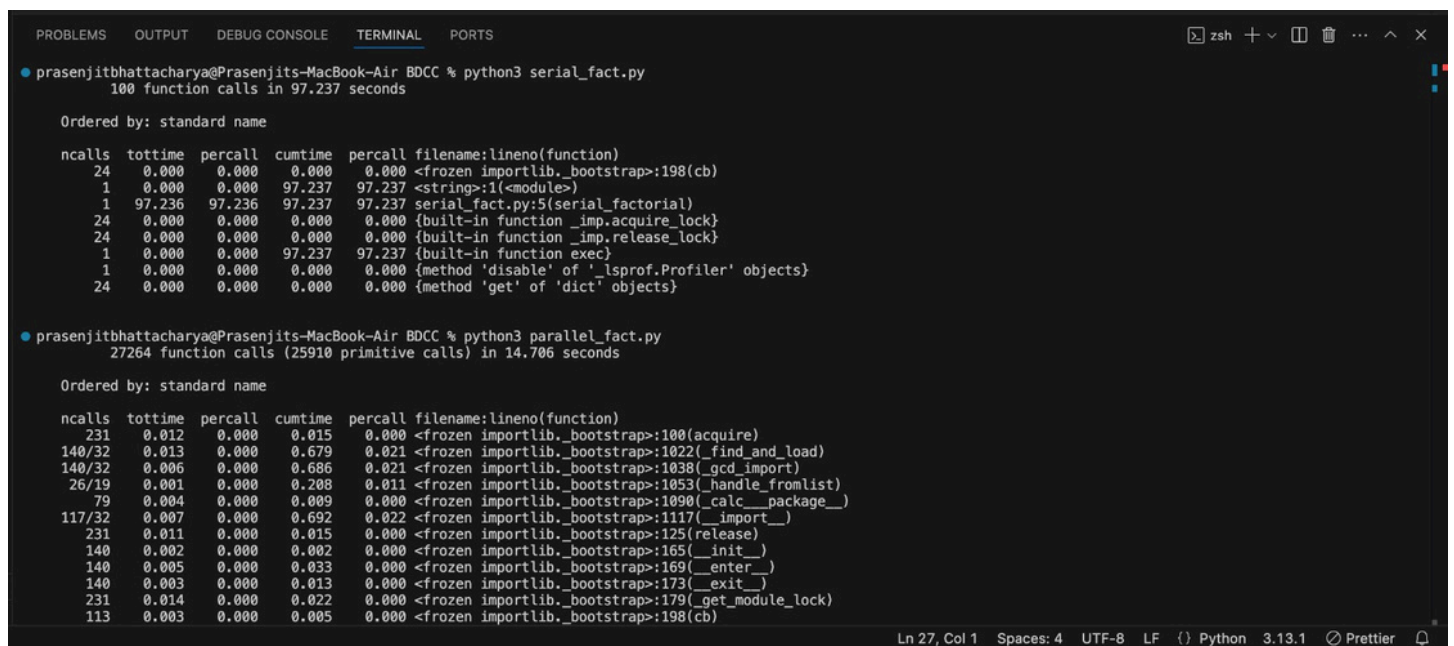
2. Investigating the effects of parallelisation

Implementing a factorial algorithm both serially and in parallel, using the multiprocessing module. Measured and compared the execution times of the serial and parallelised versions of the algorithm to analyse the impact of parallelisation on processing speed.

Improvement Analysis: Calculate the percentage improvement by:

$$\text{Improvement (\%)} = \frac{\text{Sequential Time} - \text{Parallelized Time}}{\text{Sequential Time}} \times 100$$

Output:



```
prasenjitbhattacharya@Prasenjits-MacBook-Air BDCC % python3 serial_fact.py
100 function calls in 97.237 seconds

Ordered by: standard name
ncalls  tottime  percall  cumtime  percall  filename:lineno(function)
24      0.000    0.000    0.000    0.000  <frozen importlib._bootstrap>:198(cb)
1      0.000    0.000    97.237    97.237  <string>:1(<module>)
1      97.236    97.236    97.237    97.237  serial_fact.py:5(serial_factorial)
24      0.000    0.000    0.000    0.000  {built-in function _imp.acquire_lock}
24      0.000    0.000    0.000    0.000  {built-in function _imp.release_lock}
1      0.000    0.000    97.237    97.237  {built-in function exec}
1      0.000    0.000    0.000    0.000  {method 'disable' of '_lsprof.Profiler' objects}
24      0.000    0.000    0.000    0.000  {method 'get' of 'dict' objects}

prasenjitbhattacharya@Prasenjits-MacBook-Air BDCC % python3 parallel_fact.py
27264 function calls (25910 primitive calls) in 14.706 seconds

Ordered by: standard name
ncalls  tottime  percall  cumtime  percall  filename:lineno(function)
231     0.012    0.000    0.015    0.000  <frozen importlib._bootstrap>:100(acquire)
140/32   0.013    0.000    0.679    0.021  <frozen importlib._bootstrap>:1022(_find_and_load)
140/32   0.006    0.000    0.686    0.021  <frozen importlib._bootstrap>:1038(_gcd_import)
26/19    0.001    0.000    0.208    0.011  <frozen importlib._bootstrap>:1053(_handle_fromlist)
79       0.004    0.000    0.009    0.000  <frozen importlib._bootstrap>:1090(_calc__package__)
117/32   0.007    0.000    0.692    0.022  <frozen importlib._bootstrap>:1117(_import_)
231     0.011    0.000    0.015    0.000  <frozen importlib._bootstrap>:125(release)
140     0.002    0.000    0.002    0.000  <frozen importlib._bootstrap>:165(__init__)
140     0.005    0.000    0.033    0.000  <frozen importlib._bootstrap>:169(__enter__)
140     0.003    0.000    0.013    0.000  <frozen importlib._bootstrap>:173(__exit__)
231     0.014    0.000    0.022    0.000  <frozen importlib._bootstrap>:179(_get_module_lock)
113     0.003    0.000    0.005    0.000  <frozen importlib._bootstrap>:198(cb)
```

Result:

The performance of the different Python implementations and the factorial algorithms (serial vs parallel) will be displayed in terms of execution time.

1. Serial execution completed in 97.237 seconds.
2. Parallel execution completed in 14.706 seconds.

Performance Improvement:

$$\text{Improvement (\%)} = \frac{\text{Serial Time} - \text{Parallelised Time}}{\text{Serial Time}} \times 100$$

Substituting the values:

Improvement (%)=
$$\frac{97.237 - 14.706}{97.237} \times 100 \approx 84.88\%$$

Parallelisation resulted in an **84.88% improvement in execution time**.

Scalability Analysis:

1. Serial Execution

- **Time Complexity:** O(n) due to sequential multiplication.
- **Performance:** Simple but slow for large n; limited by computational time and memory.

2. Parallel Execution

- **Time Complexity:** O(n/p)+O(p), where p is the number of processes.
- **Performance:** Improved execution time by distributing calculations across CPU cores.
 - **Results:** 97.23s (serial) vs. 14.70s (parallel with 4 processes).
 - **Overhead:** Task division and inter-process communication reduce efficiency.

3. Key Observations

- **Speedup:** Significant improvement (~6.6x) but not linear due to overhead.
- **Scalability:** Limited by the number of CPU cores and inter-process costs.

4. Scalability and Big-O Analysis

Metric	Serial Implementation	Parallel Implementation
Time Complexity	O(n)	O(n/p)+O(p) (for p processes)
Speedup	1x	4–6x (observed)
Scalability	Limited by n	Limited by p and inter-process overhead

5. Observations and Challenges

- **Serial Bottlenecks:** Serial execution is bottlenecked by the sequential nature of calculations.

- **Parallel Efficiency:** Gains diminish with increased p due to overhead and the law of diminishing returns.
- **Hardware Constraints:** The number of CPU cores limits the maximum parallel speedup.

6. Future Improvements

- Optimize task division to ensure equal load distribution across processes.
- Experiment with different parallelism models, such as GPU acceleration, to further reduce execution time.
- Analyse the impact of using more advanced libraries like NumPy for enhanced computational performance.

This structured analysis highlights the trade-offs between serial and parallel approaches and provides insights into their scalability and performance limits.