Name : Johanna Rodrigues
Roll no. : 7

UID : 225011
Course : TYIT

CIA 1 Assignment: Python Performance Analysis: A Comparative Study with Parallelization

GitHub Repository Link :
https://github.com/Jojo-Rod/Python-Performance-Analysis-A_Comparative_Study_with_Parallelization

1. Performance Comparison of Python Implementations or Flavours (CPython, PyPy, Jython) using various benchmarks:

COMPARATIVE CHART ANALYSIS :

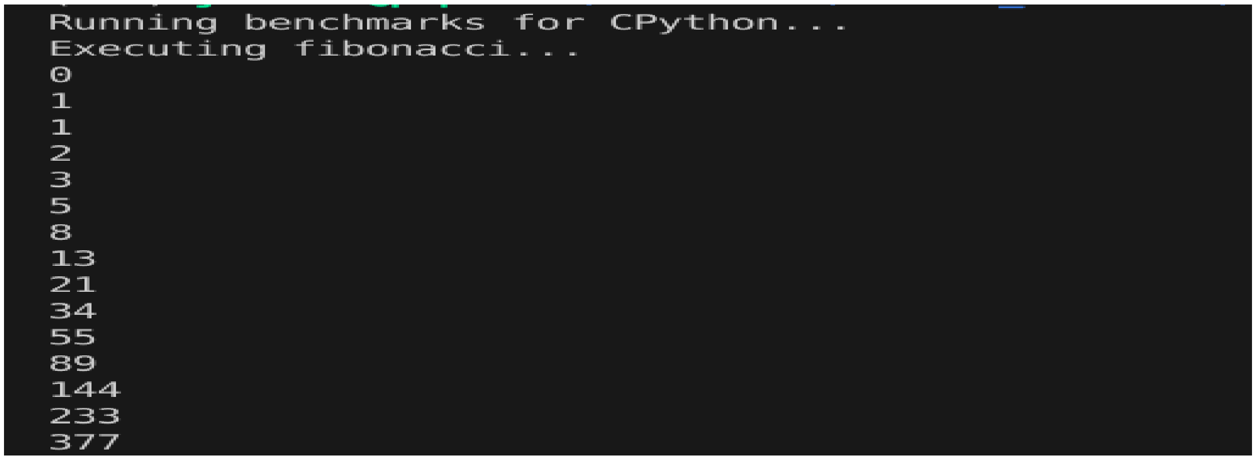
Benchmarks	CPython (seconds)	PyPy (seconds)	Jython (seconds)
fibonacci	0.01	0.02	0.68
string_reverse	0.01	0.03	0.74
file_write	0.02	0.04	0.82
list_sorting	0.6	0.19	2.19

OBSERVATIONS :

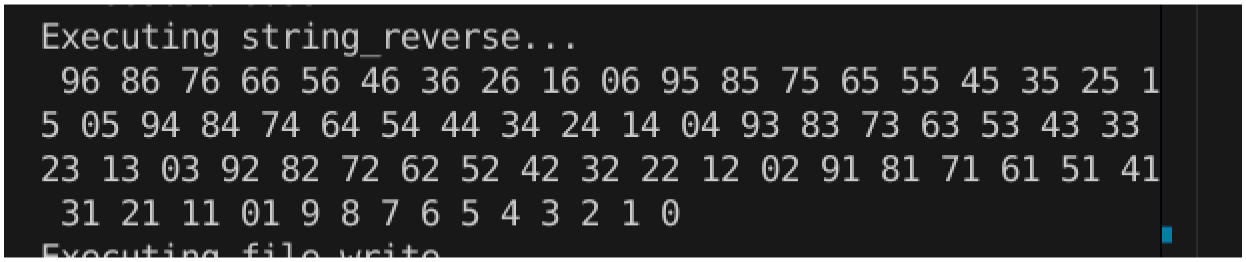
Benchmark	CPython (seconds)	PyPy (seconds)	Jython (seconds)
fibonacci	Fast for small iterative tasks. However, the Fibonacci sequence involves simple arithmetic and a while loop, making it straightforward for CPython to execute quickly.	Faster due to JIT optimization. However, optimization gains are moderate due to the simplicity of the arithmetic operations.	Jython runs on the Java Virtual Machine (JVM), which isn't optimized for Python-style dynamic typing or iterative calculations. This results in much slower execution compared to CPython and PyPy.
string_reverse	String manipulation (e.g., concatenation and slicing) in CPython is relatively fast but not the most optimized due to the creation of new strings in each iteration of the loop.	PyPy's JIT can optimize repeated string concatenations better than CPython, reducing execution time.	Jython's performance suffers significantly here because string handling in JVM is not as fast or efficient as in CPython or PyPy. Java strings are immutable, and creating new strings during concatenation can become computationally expensive.
file_write	File I/O operations in CPython are generally efficient but constrained by Python's GIL (Global Interpreter Lock) and the OS-level disk I/O limitations.	PyPy offers faster file I/O in this case because it optimizes the repeated writes, benefiting from its faster overall execution speed.	File operations in Jython are slower because they are implemented using Java's file I/O classes, which introduce additional overhead. Moreover, Python-style operations may not map directly to JVM's strengths.
list_sorting	CPython's sorted() function is	PyPy excels here because its	Sorting large lists in Jython is slow due to

	implemented using the Timsort algorithm, which is efficient for typical datasets. The performance is reasonable but limited by CPython's lack of JIT optimization.	**JIT compilation can optimize the iteration and sorting process, resulting in significantly better performance than CPython.**	**the overhead of mapping Python list operations to Java's data structures and the lack of JIT optimizations tailored for such tasks.**

<u>OUTPUTS :</u> (CPython)

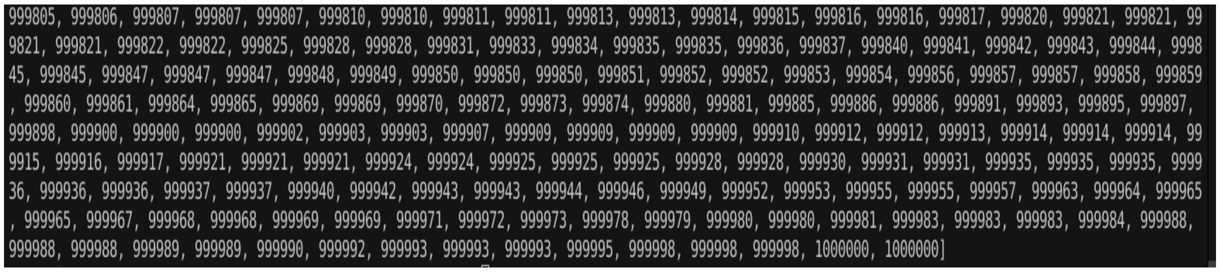


```
Running benchmarks for CPython...
Executing fibonacci...
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
```

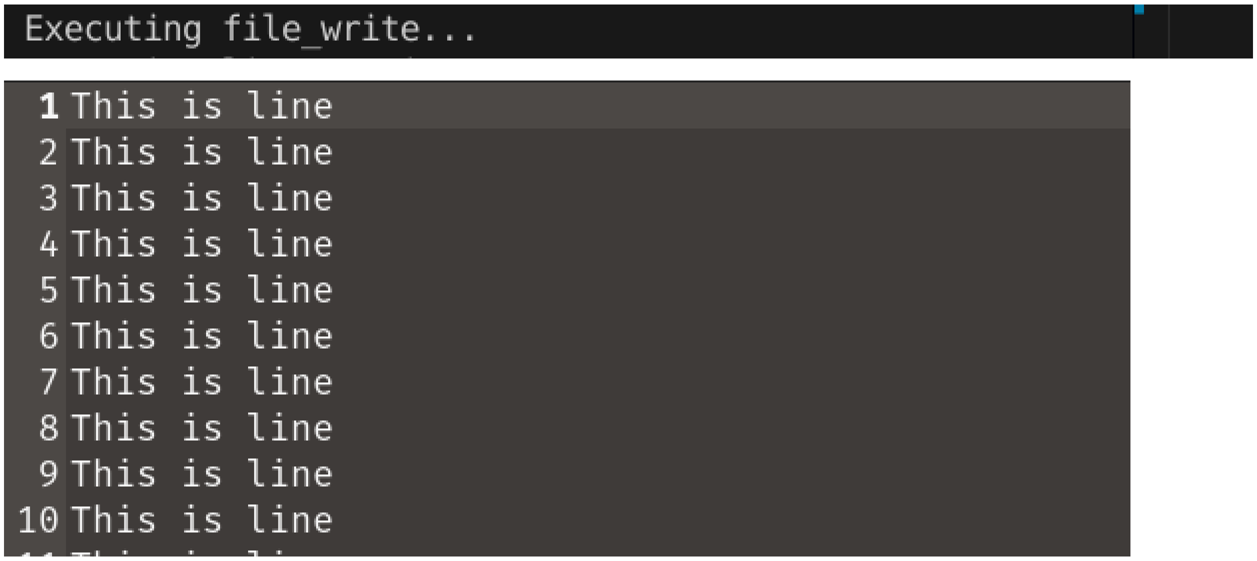


```
Executing string_reverse...
 96 86 76 66 56 46 36 26 16 06 95 85 75 65 55 45 35 25 1
5 05 94 84 74 64 54 44 34 24 14 04 93 83 73 63 53 43 33
23 13 03 92 82 72 62 52 42 32 22 12 02 91 81 71 61 51 41
 31 21 11 01 9 8 7 6 5 4 3 2 1 0
Executing file write
```

List Sorting output :



```
999805, 999806, 999807, 999807, 999807, 999810, 999810, 999811, 999811, 999813, 999813, 999814, 999815, 999816, 999816, 999817, 999820, 999821, 999821, 99
9821, 999821, 999822, 999822, 999825, 999828, 999828, 999831, 999833, 999834, 999835, 999835, 999836, 999837, 999840, 999841, 999842, 999843, 999844, 9998
45, 999845, 999847, 999847, 999847, 999848, 999849, 999850, 999850, 999850, 999851, 999852, 999852, 999853, 999854, 999856, 999857, 999857, 999858, 999859
, 999860, 999861, 999864, 999865, 999869, 999869, 999870, 999872, 999873, 999874, 999880, 999881, 999885, 999886, 999886, 999891, 999893, 999895, 999897,
999898, 999900, 999900, 999900, 999902, 999903, 999903, 999907, 999909, 999909, 999909, 999909, 999910, 999912, 999912, 999913, 999914, 999914, 999914, 99
9915, 999916, 999917, 999921, 999921, 999921, 999924, 999924, 999925, 999925, 999925, 999928, 999928, 999930, 999931, 999931, 999935, 999935, 999935, 9999
36, 999936, 999936, 999937, 999937, 999940, 999942, 999943, 999943, 999944, 999946, 999949, 999952, 999953, 999955, 999955, 999957, 999963, 999964, 999965
, 999965, 999967, 999968, 999968, 999969, 999969, 999971, 999972, 999973, 999978, 999979, 999980, 999980, 999981, 999983, 999983, 999983, 999984, 999988,
999988, 999988, 999989, 999989, 999990, 999992, 999993, 999993, 999993, 999995, 999998, 999998, 999998, 1000000, 1000000]
```

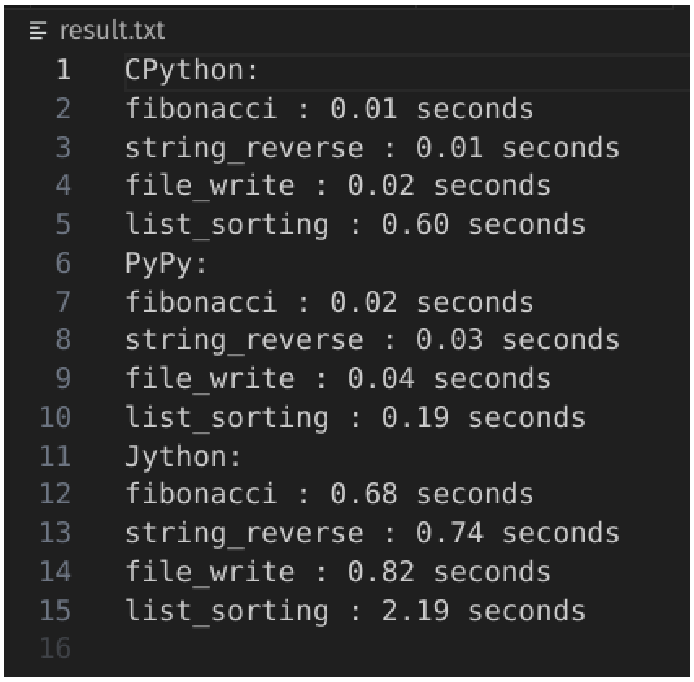


```
Executing file_write...
```

```
This is line
This is line
This is line
This is line
This is line
This is line
This is line
This is line
This is line
This is line
```

Similarly, the above mentioned 4 benchmarks are executed for the other two Python flavours as well namely, PyPy and Jython.

File storing benchmark execution time results for each of the selected Python Implementations :



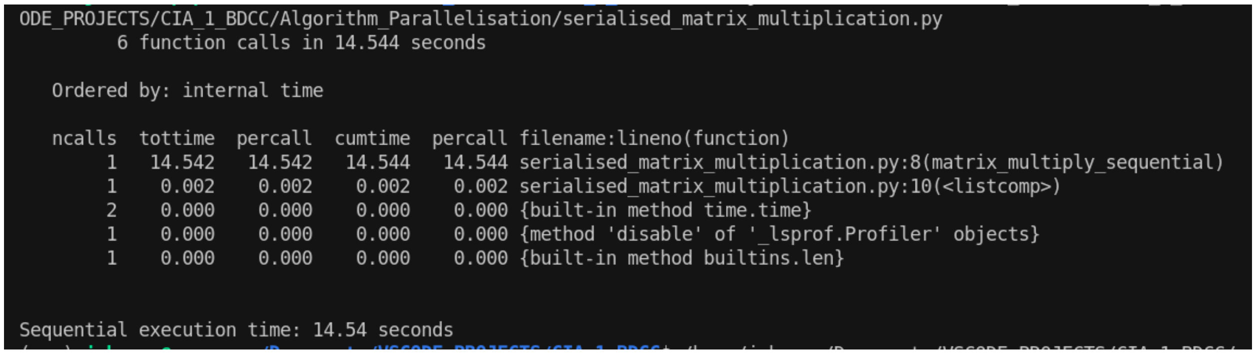
result.txt

```
CPython:
fibonacci : 0.01 seconds
string_reverse : 0.01 seconds
file_write : 0.02 seconds
list_sorting : 0.60 seconds
PyPy:
fibonacci : 0.02 seconds
string_reverse : 0.03 seconds
file_write : 0.04 seconds
list_sorting : 0.19 seconds
Jython:
fibonacci : 0.68 seconds
string_reverse : 0.74 seconds
file_write : 0.82 seconds
list_sorting : 2.19 seconds
```

2. Algorithm Parallelization

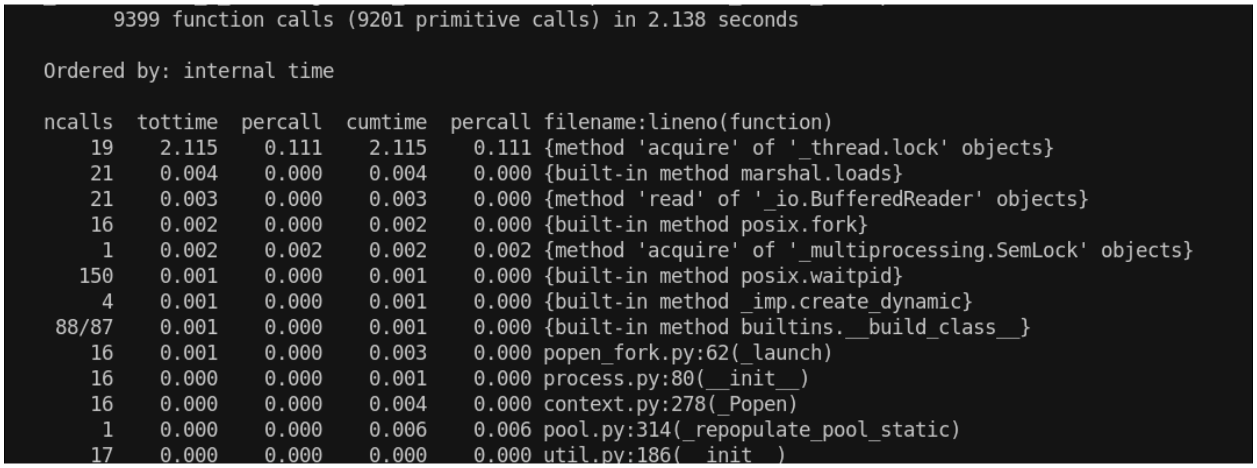
Algorithm selected : Matrix Multiplication
Complexity : O(n^3)

Serialised Program Output using ‘cProfile’ tool :

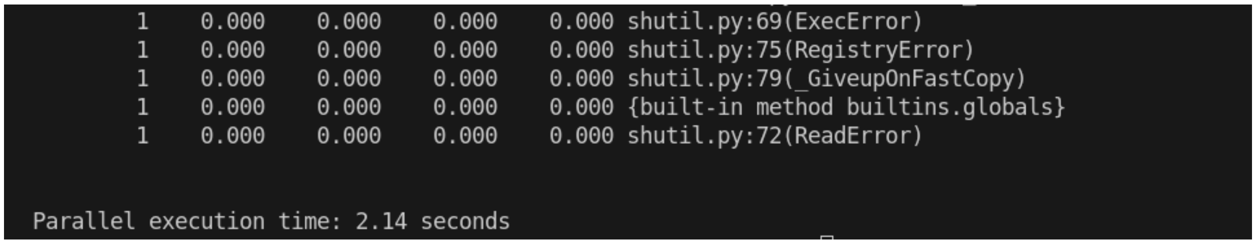


```
ODE_PROJECTS/CIA_1_BDCC/Algorithm_Parallelisation/serialised_matrix_multiplication.py
         6 function calls in 14.544 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1   14.542   14.542   14.544   14.544 serialised_matrix_multiplication.py:8(matrix_multiply_sequential)
        1    0.002    0.002    0.002    0.002 serialised_matrix_multiplication.py:10(<listcomp>)
        2    0.000    0.000    0.000    0.000 {built-in method time.time}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
        1    0.000    0.000    0.000    0.000 {built-in method builtins.len}


Sequential execution time: 14.54 seconds
```

Parallelised Program Output using ‘cProfile’ tool :

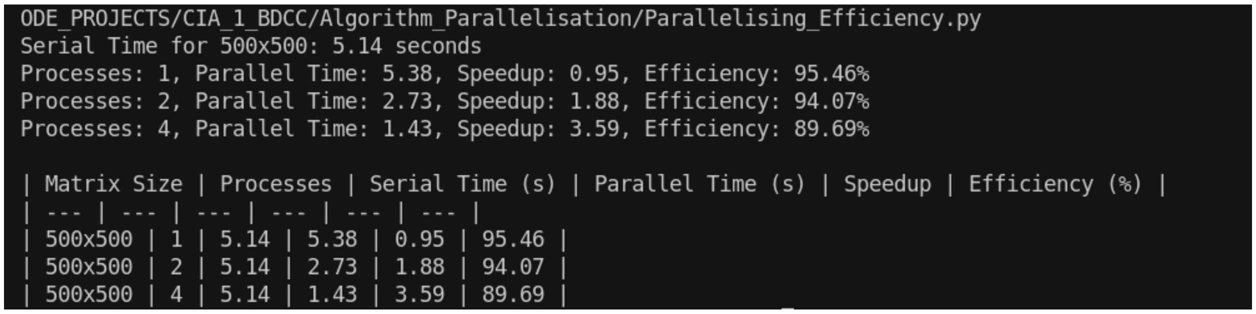


```
         9399 function calls (9201 primitive calls) in 2.138 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
       19    2.115    0.111    2.115    0.111 {method 'acquire' of '_thread.lock' objects}
       21    0.004    0.000    0.004    0.000 {built-in method marshal.loads}
       21    0.003    0.000    0.003    0.000 {method 'read' of '_io.BufferedReader' objects}
       16    0.002    0.000    0.002    0.000 {built-in method posix.fork}
        1    0.002    0.002    0.002    0.002 {method 'acquire' of '_multiprocessing.SemLock' objects}
      150    0.001    0.000    0.001    0.000 {built-in method posix.waitpid}
        4    0.001    0.000    0.001    0.000 {built-in method _imp.create_dynamic}
    88/87    0.001    0.000    0.001    0.000 {built-in method builtins.__build_class__}
       16    0.001    0.000    0.003    0.000 popen_fork.py:62(_launch)
       16    0.000    0.000    0.001    0.000 process.py:80(__init__)
       16    0.000    0.000    0.004    0.000 context.py:278(_Popen)
        1    0.000    0.000    0.006    0.006 pool.py:314(_repopulate_pool_static)
       17    0.000    0.000    0.000    0.000 util.py:186(__init__)
```



```
        1    0.000    0.000    0.000    0.000 shutil.py:69(ExecError)
        1    0.000    0.000    0.000    0.000 shutil.py:75(RegistryError)
        1    0.000    0.000    0.000    0.000 shutil.py:79(_GiveupOnFastCopy)
        1    0.000    0.000    0.000    0.000 {built-in method builtins.globals}
        1    0.000    0.000    0.000    0.000 shutil.py:72(ReadError)


Parallel execution time: 2.14 seconds
```

Checking Serialisation & Parallelization outputs for Execution Time, Speedup, Efficiency using 'Pool' and 'Time' libraries :



```
ODE_PROJECTS/CIA_1_BDCC/Algorithm_Parallelisation/Parallelising_Efficiency.py
Serial Time for 500x500: 5.14 seconds
Processes: 1, Parallel Time: 5.38, Speedup: 0.95, Efficiency: 95.46%
Processes: 2, Parallel Time: 2.73, Speedup: 1.88, Efficiency: 94.07%
Processes: 4, Parallel Time: 1.43, Speedup: 3.59, Efficiency: 89.69%

| Matrix Size | Processes | Serial Time (s) | Parallel Time (s) | Speedup | Efficiency (%) |
| --- | --- | --- | --- | --- | --- |
| 500x500 | 1 | 5.14 | 5.38 | 0.95 | 95.46 |
| 500x500 | 2 | 5.14 | 2.73 | 1.88 | 94.07 |
| 500x500 | 4 | 5.14 | 1.43 | 3.59 | 89.69 |
```

Matrix Size	Processes	Serial Time (s)	Parallel Time (s)	Speedup	Efficiency (%)
500x500	**1**	**5.14**	**5.38**	**0.95**	**95.46**
500x500	**2**	**5.14**	**2.73**	**1.88**	**94.07**
500x500	**4**	**5.14**	**1.43**	**3.59**	**89.69**

Scalability Analysis

1. **Execution Time:**
    - Parallel execution time decreases with an increasing number of processes:
        i. 1 process: 5.38 seconds.
        ii. 2 processes: 2.73 seconds.
        iii. 4 processes: 1.43 seconds.
    - Conclusion: Execution time improves significantly with parallelization, though the improvement is less pronounced as the number of processes increases.
2. **Big O Notation**:
    - Serial: O(n^3) (iterating through all rows, columns, and elements).
    - Parallel: Still O(n^3) overall but with reduced wall-clock time due to process parallelization.
3. **Efficiency** :
    - Parallelization is effective for 2 processes, with high efficiency.
    - Beyond 2 processes, the efficiency starts to decrease due to communication and task management overhead, which is typical in parallel systems.

4. **Speedup:**

   Speedup measures the ratio of serial time to parallel time and indicates how much faster the task runs when parallelized.

   - From the table:
     1. Speedup is 0.95 with 1 process, showing slight inefficiency even in serial mode (possibly due to system overhead).
     2. Speedup is 1.88 with 2 processes, indicating a nearly linear improvement.
     3. Speedup is 3.59 with 4 processes, which is less than the ideal 4 due to parallelization overhead.
   - Conclusion: Speedup is good but suboptimal, showing diminishing returns as more processes are used.