BDDC CIA 1 Assignment:

Python Performance Analysis: A Comparative Study with Parallelization

Name: Archi Mehta

Roll No.: 25

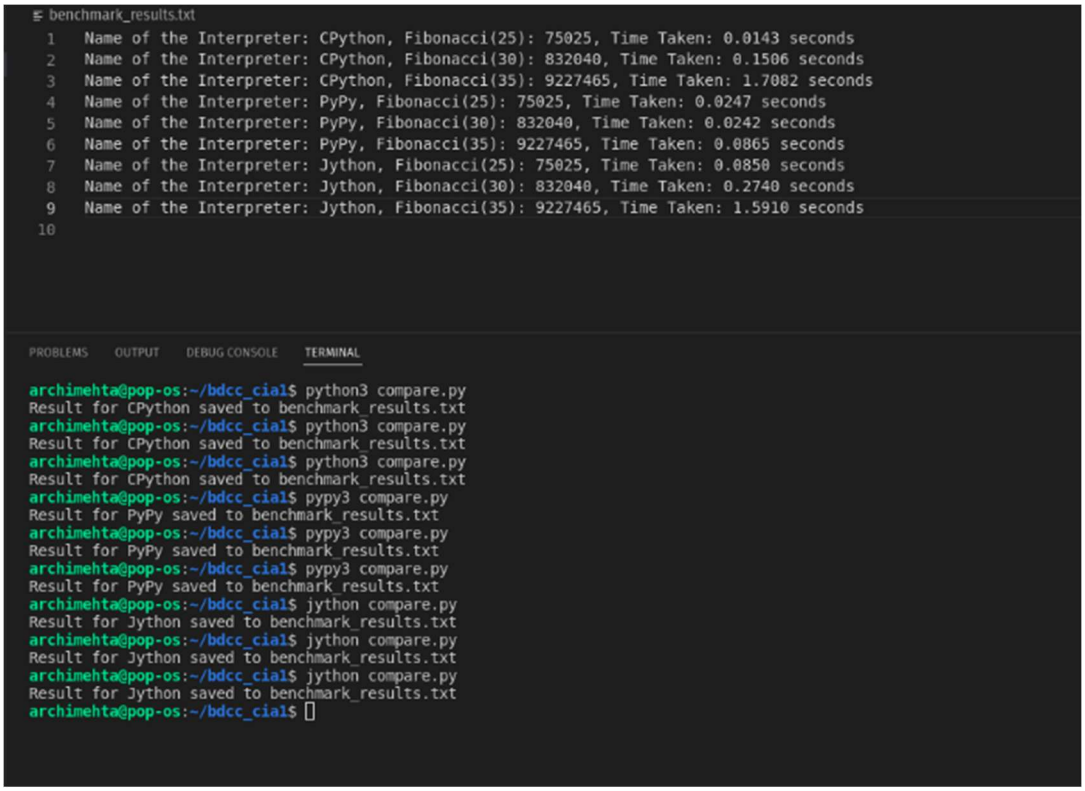
UID:225038

Task 1. Performance Comparison of Python Implementations

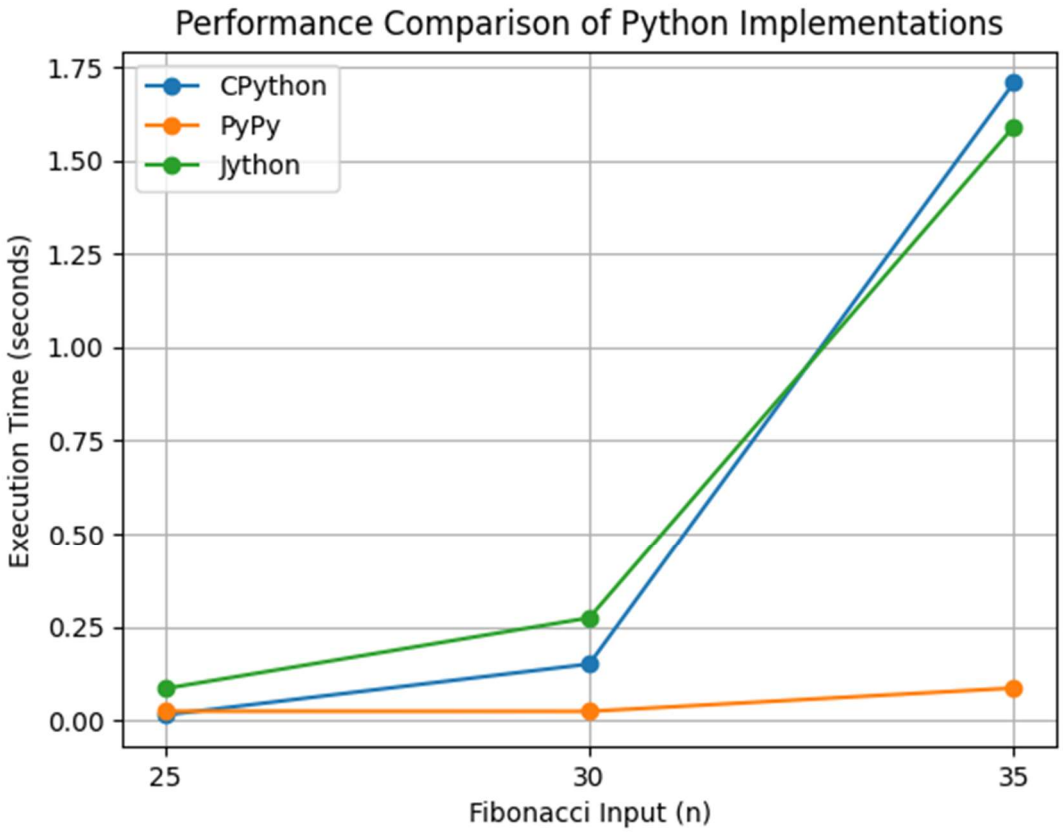
I used the following Pyhton implementations:

1. **CPython**: The reference implementation of Python, written in C, and the most widely used Python interpreter.
2. **PyPy**: An alternative Python interpreter with a focus on speed through Just-in-Time (JIT) compilation.
3. **Jython**: Python implementation for the Java platform, allowing integration with Java libraries and applications.

Output:



Performance Comparison Chart:



Analysis:

CPython:

1. Fibonacci(25): 0.0143 seconds
2. Fibonacci(30): 0.1506 seconds
3. Fibonacci(35): 1.7082 seconds
4. Trend: The time grows significantly with the increase in the input number (Fibonacci index). This is expected since CPython's execution speed is not optimized for large computations like Fibonacci numbers.

PyPy:

1. Fibonacci(25): 0.0247 seconds
2. Fibonacci(30): 0.0242 seconds
3. Fibonacci(35): 0.0865 seconds
4. Trend: PyPy shows a significant performance improvement over CPython, particularly for larger Fibonacci numbers. The use of Just-in-Time (JIT) compilation accelerates the execution, resulting in a much smaller increase in time with larger numbers.

Jython:

1. Fibonacci(25): 0.0850 seconds
2. Fibonacci(30): 0.2740 seconds
3. Fibonacci(35): 1.5910 seconds
4. Trend: Jython performs slower than both CPython and PyPy, and the time increases rapidly with larger Fibonacci numbers. This would signify that Jython is less optimized to handle such recursive computations than CPython and PyPy.

In conclusion:

1. CPython is the slowest among the three interpreters, with performance degrading significantly as the Fibonacci index increases.
2. PyPy demonstrates superior performance, especially for larger inputs, owing to JIT compilation.
3. Jython has the slowest overall performance, especially as the index grows, likely due to the overhead of running Python on the Java Virtual Machine (JVM).

Task 2. Algorithm Parallelization

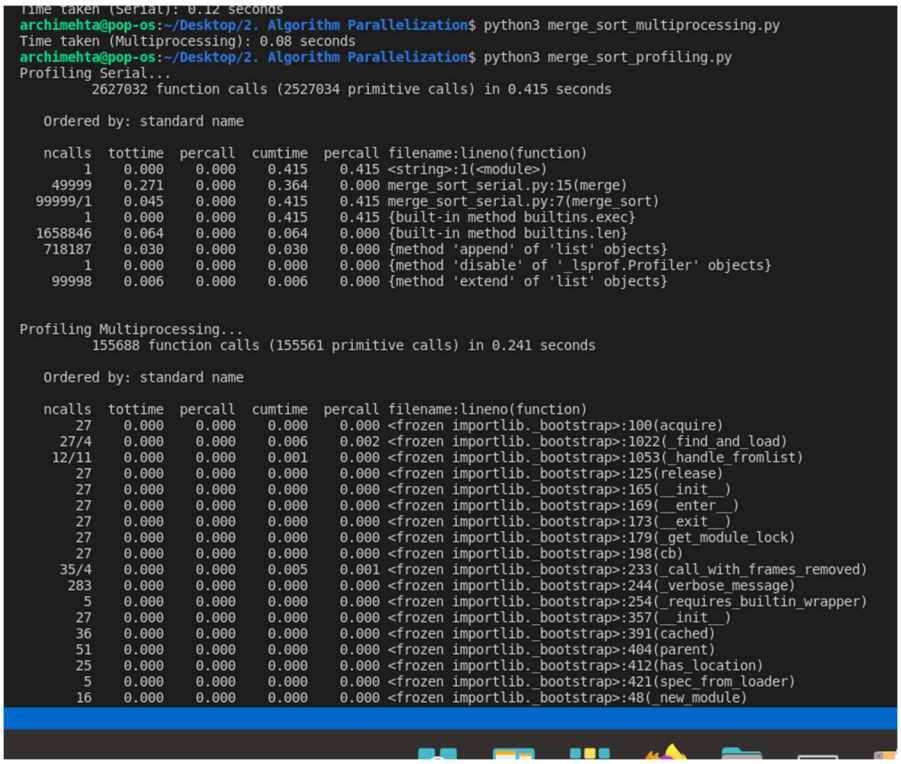
I used Merge Sort for this task.

Time Analysis:

1. Before parallelization:
    - **Time Taken**: 0.12 seconds consistently.
    - **Observation**: The serial version of the Merge Sort runs within a single process, therefore, it tackles the task of sorting sequentially. Its execution time is thus constant and predictable for several runs.

2. After parallelization (using multiprocessing):
    - **Time Taken**: 0.08 seconds consistently.
    - **Observation**: The multiprocessing version is faster than the serial one. Dividing the task of sorting across multiple CPU cores, or parallelism, reduces the total execution time to 0.08 seconds. This is a performance improvement because of the parallel processing capabilities, which enable multiple subarrays to be sorted simultaneously.
3. Comparison:
    - Speedup: Multiprocessing version is 1.5 times faster than the serial version (0.12 / 0.08).
    - Conclusion: Multiprocessing gives a performance boost, especially when the task can be effectively parallelized, such as sorting large datasets. The improvement depends on the size of the input and the number of available cores in the system. For smaller datasets, the speedup may not be dramatic, but for larger inputs, the parallel approach should scale better.

Profiling:



Serial Execution (before parallelization):

- **Total Time**: 0.415 seconds.
- **Function Calls**: 2,627,032 total (2,527,034 primitive).

Multiprocessing Execution (after parallelization):

- **Total Time**: 0.241 seconds.
- **Function Calls**: 155,688 total (155,561 primitive).

Scalability Analysis in terms of Big O Notation:

1. **Serial Execution:**
    - Time Complexity: O(n log n)
      Its time complexity is O(n log n) because Merge Sort uses a divide-and-conquer strategy in its serial form. It works by dividing the input list into two halves recursively and then sorting each half.

    - Space Complexity: O(n)
      Extra space for temporary arrays during the merging process makes Merge Sort have a space complexity of O(n).

- Efficiency:
  Merge Sort scales well with large datasets because of its logarithmic time complexity. However, as the input size increases, the runtime grows at a rate proportional to n log n. This makes it suitable for handling large datasets but not as efficient for extremely large-scale data compared to algorithms like QuickSort (on average).

2. **Multiprocessing Parallelized Execution**
   - Time Complexity: O(n log n)
     The time complexity remains the same as the serial Merge Sort since the underlying algorithm is not changed. Nonetheless, parallelization can greatly accelerate the merging process, significantly for large data sets.

   - Space complexity: O(n)
     Parallelization still does not modify the space needs, since for each process a space is required to contain parts of a list. Its space complexity remains O(n).

   - Efficiency:
     Speedwise improvement also comes from paralleling the algorithm, as breaking the task and distributing it in multiple processors/correspondingly cores improves parallelized merge sorting results in a diminution of running time for big inputs.