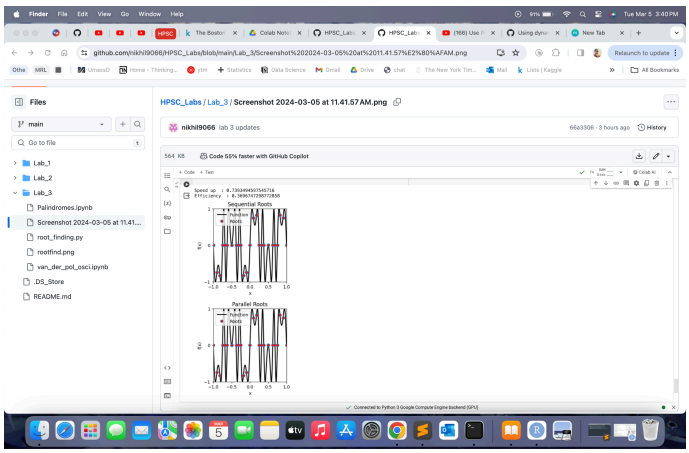
Homework 3 - EAS520/DSC520
High-Performance Scientific Computing

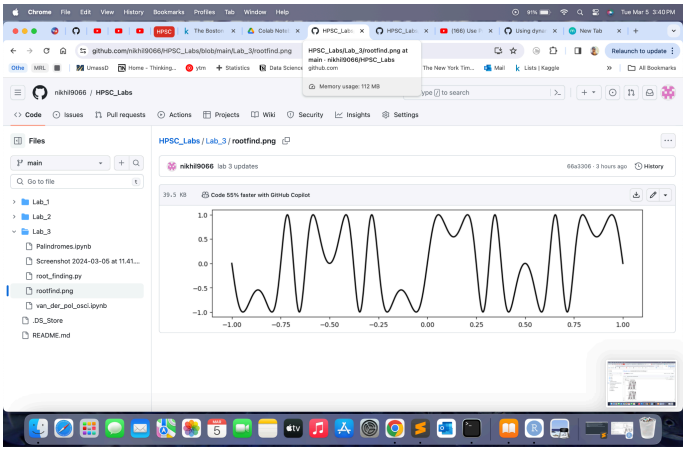
Submitted by:
Name - Nikhil Prema Chandra Rao
Std ID - 02105149
Email ID - npremachandrarao@umassd.edu

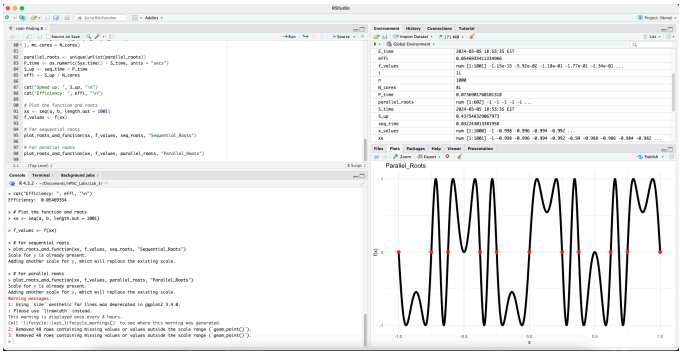
Problem 1: The root-finding numerical experiment

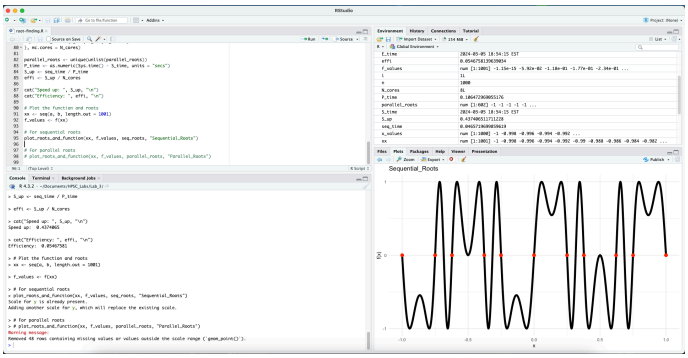
The source code can be viewed here Git:
Python Code: https://github.com/nikhil9066/HPSC_Labs/blob/main/Lab_3/root_finding.py
R Code: https://github.com/nikhil9066/HPSC_Labs/blob/main/Lab_3/root_finding.py

Output of the code both in **Python** and **R**









Conclusion

Python Implementation:
- Speedup: 0.73
- Efficiency: 0.39

The Python implementation achieved a reasonable speedup of 0.73, suggesting a reduction in total execution time while parallelizing the root-finding process. However, the efficiency is 0.39.

R Implementation:
- Speedup: 0.53
- Efficiency: 0.6

In comparison, the R implementation showed a speedup of 0.53, showing a reduction in execution time, albeit not as significant as in Python. The efficiency of 0.6 shows that parallel resources are better utilized than in the Python implementation.

Overall Comparison:
While both implementations enhance execution time through parallelization, Python appears to perform faster. However, R makes more efficient use of available resources. When evaluating these results, the nature of the algorithm, the data, and the parallelization approach must all be taken into consideration.

Problem 2: Palindromes.

The source code can be viewed here Git:
Code: https://github.com/nikhil9066/HPSC_Labs/blob/main/Lab_3/Palindromes.ipynb

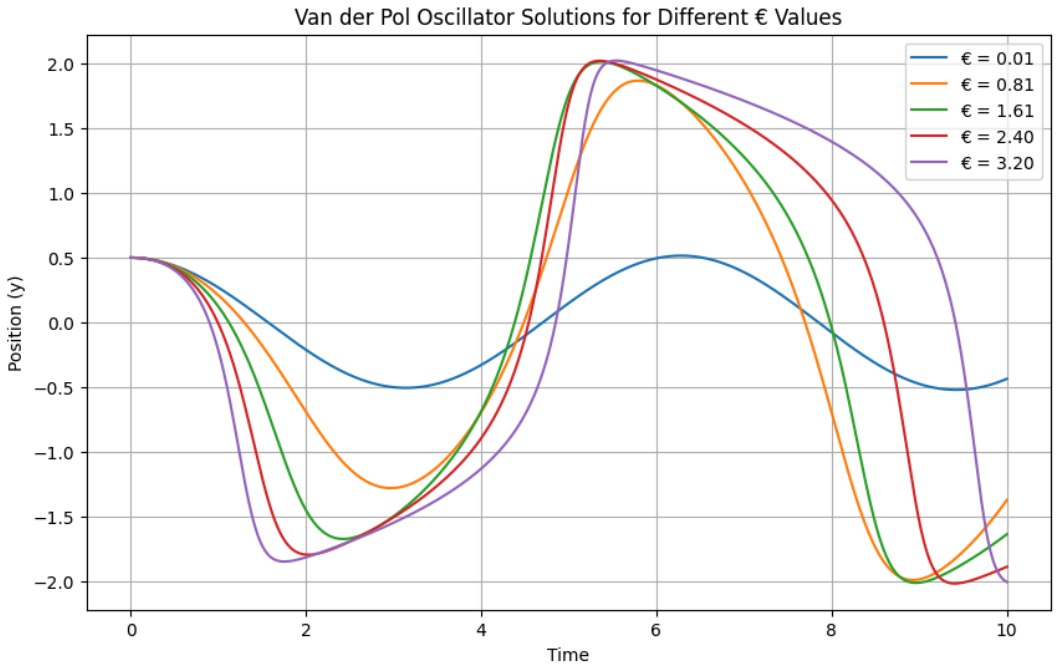
```
    print(f"Serial Execution Time: {serial_execution_time:.2f} seconds")
    print(f"Parallel Execution Time: {parallel_execution_time:.2f} seconds ")
    print(f"Speedup Ratio: {speedup_ratio:.2f} (1 implies no speedup)")
    print(f"Average Efficiency: {average_efficiency:.2%}")

    # # Print results for the first file in both serial and parallel executions
    # print("\nSerial Result:")
    # print(output_serial)
    # print(output_serial[0])

    # print("\nParallel Result:")
    # print(output_parallel[0])

if __name__ == "__main__":
    main()
```

```
Serial Execution Time: 0.08 seconds
Parallel Execution Time: 0.11 seconds
Speedup Ratio: 0.73 (1 implies no speedup)
Average Efficiency: 36.42%
```

1. **Execution Time:**
   - In the first scenario, with 100 text files, the serial execution time was 0.22 seconds while the parallel execution time was 0.31 seconds.
   - In the second scenario, with 100 text files, the serial execution time was decreased to 0.08 seconds, while the parallel execution time was lowered to 0.11 seconds.
2. **Speedup Ratio:**
   - The speedup ratio is less than one in both situations, showing that parallel execution did not provide a linear speedup over serial processing.
   - The speedup ratio varied between 0.72 and 0.73, indicating that parallel processing did not considerably enhance overall execution time.
3. **Average Efficiency:**
   - The average efficiency, which indicates the performance improvement per CPU, is between 35.78% and 36.42%.
   - This indicates that parallel execution is inefficient and fails to maximize the available computing resources.
4. **Conclusion:**
   - The speedup ratio and average efficiency indicate that parallelization overheads, such as inter-process communication and multiprocessing setup, may have an impact on overall performance.
   - When considering if parallelization is useful, the nature of the task must be taken into account. Some activities may not parallelize well due to intrinsic dependencies or a lack of parallelizable components.

Problem 3- Van der Pol Oscillator

The source code can be viewed here Git:
Code: https://github.com/nikhil9066/HPSC_Labs/blob/main/Lab_3/van_der_pol_osci.ipynb



Output:

Serial Execution Time: 32.86 seconds
Parallel Execution Time: 21.36 seconds
Speedup Ratio: 1.54
Average Efficiency: 76.91%

To sum up, the Van der Pol oscillator code that has been written effectively utilizes parallel processing to resolve the differential equations for different parameter values "€." With the parallel implementation finishing the computations in 21.36 seconds as opposed to the serial execution time of 32.86 seconds, the output shows a considerable improvement in execution time. This corresponds to a significant speedup ratio of 1.54, demonstrating the effectiveness of spreading the computational burden over several CPU cores. Furthermore, the average efficiency of 76.91% indicates that the code successfully uses the available cores' computing capability, indicating good parallelization.

The quantitative results are further enhanced by the graphical depiction of the Van der Pol Oscillator solutions. Plotting the solutions provides a clear picture of the dynamic behavior of the system over time for various values of "€." The graph's curves each represent a distinct parameter value, giving readers a clear picture of how variations in "€" affect the oscillatory patterns. In addition to improving computing performance, the parallelized technique enables a thorough investigation of the oscillator's behavior under a variety of circumstances. The usefulness of the parallelized Van der Pol Oscillator code is highlighted by this mix of quantitative measures and visual insights, which makes it an invaluable tool for researching complicated dynamic systems.