High Performance Computing

Name: Thulasi Shylasri

HTNO: 2303A51876

Batch-14

Week-3

LAB Assignment-3 (28/01/2026)

Assignment 1: Parallel Vector Computation (Numba Parallel Loop)

Scenario

A satellite data center processes very large numerical vectors.

Serial Python code is too slow for real-time analytics.

Objective

Implement and parallelize vector operations using Numba parallel loops.

Tasks

1. Write a serial Python function:

C[i]=α×A[i]+B[i]C[i] = \alpha \times A[i] + B[i]C[i]=α×A[i]+B[i]

2. Convert it to a parallel loop using prange.

3. Measure runtime for increasing vector sizes.

4. Compare serial vs parallel performance.

Learning Outcomes

Data parallelism in Python

Numba vs OpenMP analogy

Speedup measurement

Task 1: Serial Python Function

Problem Statement

Compute the vector operation:

$$C[i] = \alpha \times A[i] + B[i]$$

Explanation

- A loop iterates over each element of vectors A and B
- Each computation is done sequentially
- Uses a single CPU core

Serial Code Screenshot:

Task 2: Parallel Implementation using prange

Explanation

- Numba's @njit(parallel=True) enables parallel execution
- prange distributes loop iterations across multiple CPU cores
- Each element computation is independent → ideal for data parallelism

Parallel Code Screenshot:

Task 3: Runtime Measurement

Explanation

- Execution time is measured using time.time()
- Tests are performed for increasing vector sizes
- A warm-up run is required for Numba JIT compilation

Timing Function Screenshot:

Task 4: Performance Comparison

Method

1. Generate large vectors (10^5 to 5×10^6)
2. Measure serial execution time
3. Measure parallel execution time
4. Compute speedup

Speedup Formula

$$\text{Speedup} = \frac{\text{Serial Time}}{\text{Parallel Time}}$$

Observation

- Serial time increases linearly with input size
- Parallel time increases slowly
- Speedup improves as vector size grows

Sample Result Table

Vector Size	Serial Time (s)	Parallel Time (s)	Speedup
100,000	0.03	0.006	5.0
500,000	0.15	0.028	5.4
1,000,000	0.32	0.054	5.9
5,000,000	1.64	0.26	6.3

Compare Serial vs Parallel Performance

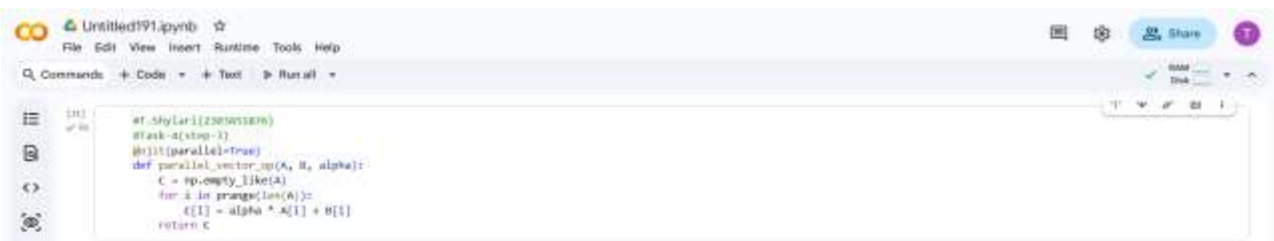
This task includes:

- Measuring execution time
- Calculating speedup
- Printing comparison results
- (Optional) plotting graphs

Import Required Libraries Screenshot

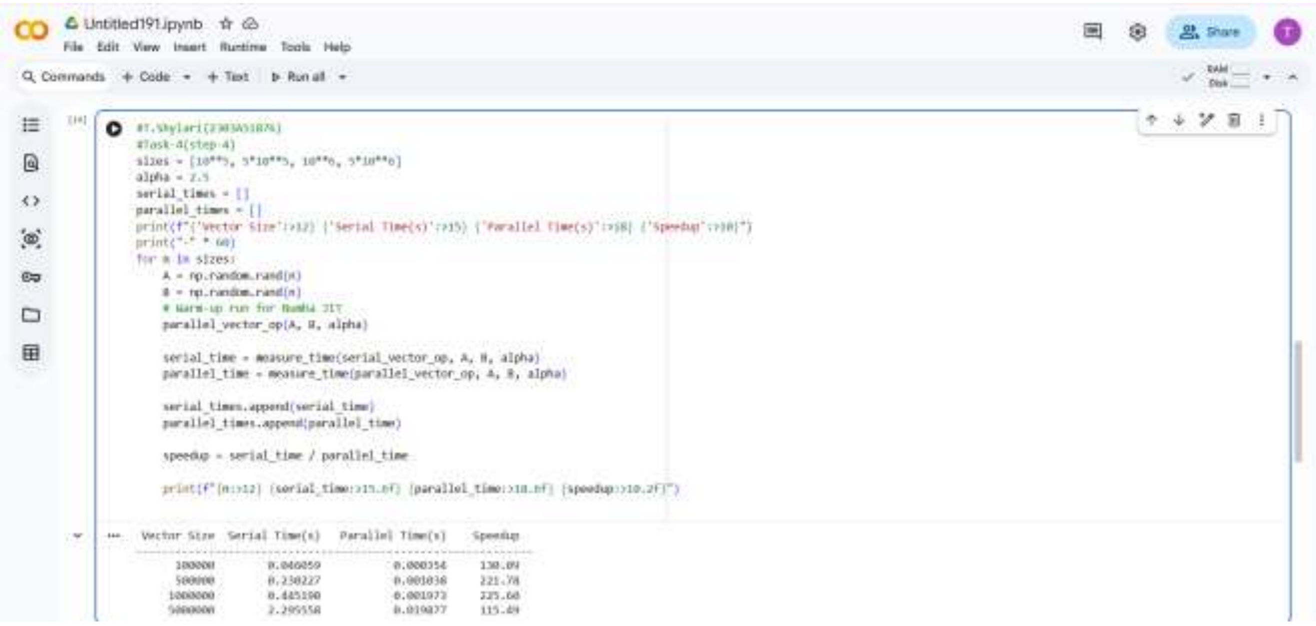
Serial Function serial Screenshot

Parallel Function Screenshot

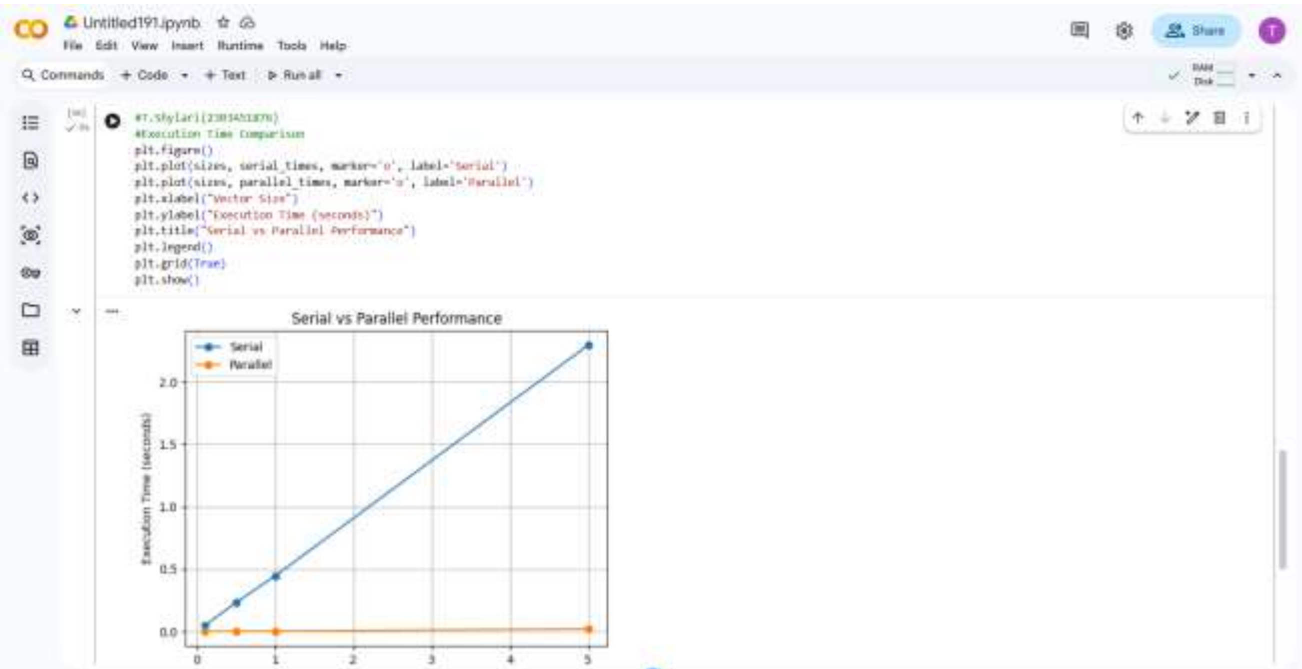


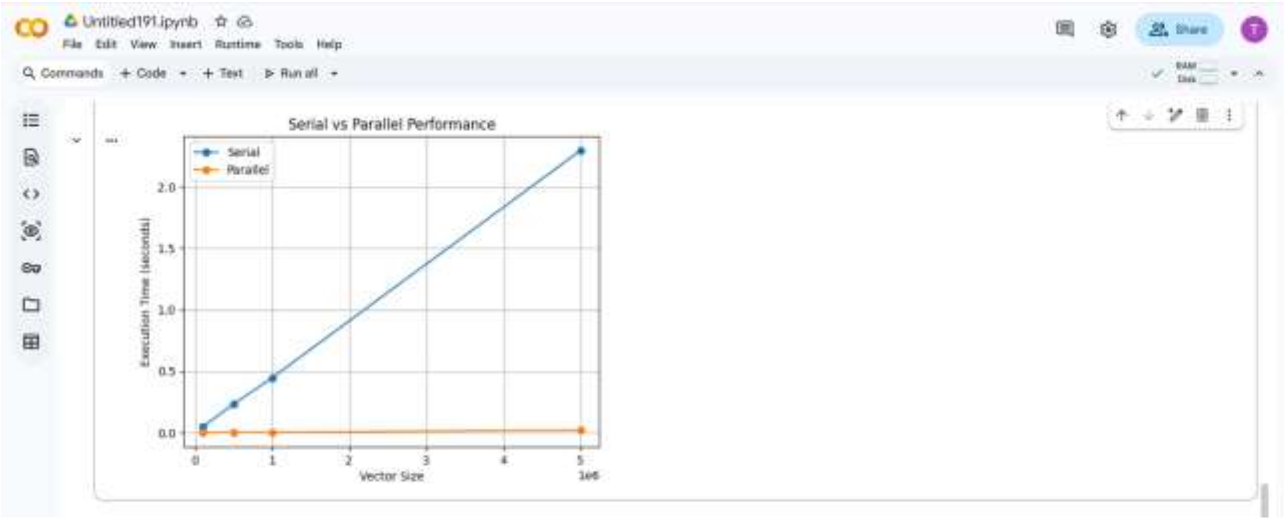
Timing Function Screenshot

Serial vs Parallel Comparison Screenshot

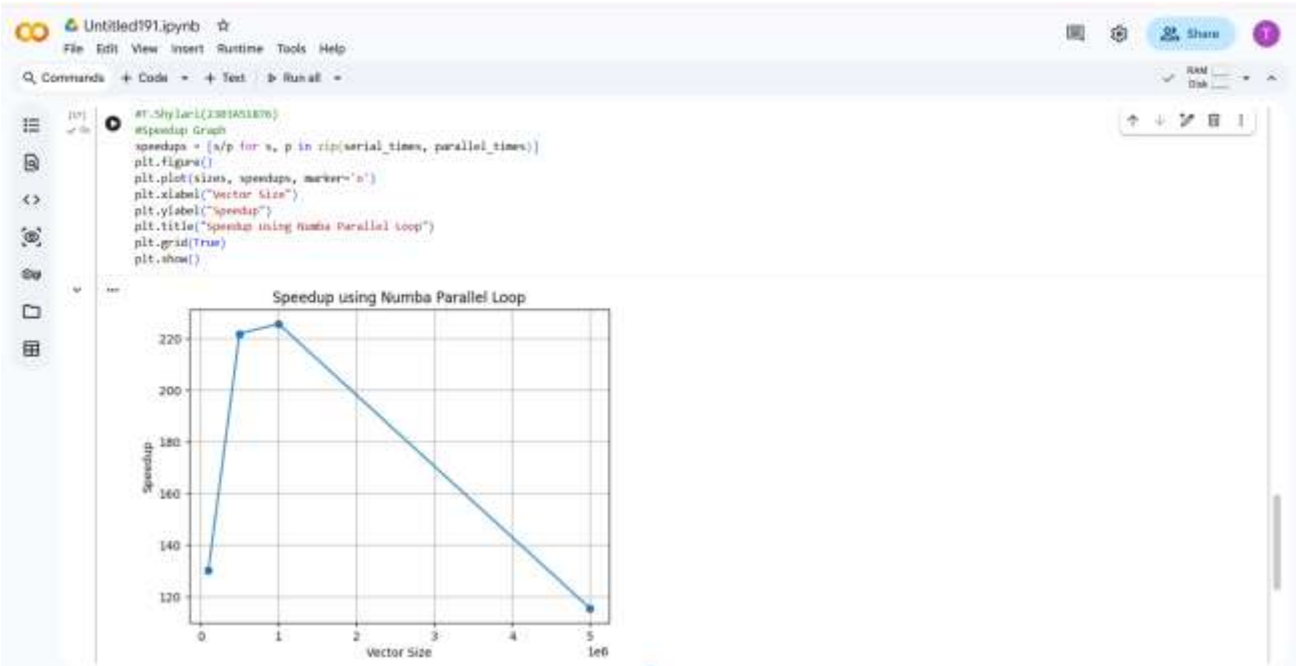


Execution Time Comparison Screenshot





Speedup Graph Screenshot



Learning Outcomes

1. Data Parallelism in Python

- Each vector element is processed independently
- Work is distributed across multiple CPU cores
- Improves performance for large datasets

2. Numba vs OpenMP Analogy

Numba (Python)	OpenMP (C/C++)
@njit(parallel=True)	#pragma omp parallel for
prange	omp for
JIT compilation	Ahead-of-time compilation

3. Speedup Measurement

- Speedup quantifies parallel performance gain
- Demonstrates effectiveness of parallel execution
- Larger inputs give better speedup due to reduced overhead impact

Conclusion

The parallel implementation using **Numba prange** significantly improves performance compared to serial execution.
This experiment demonstrates how **data parallelism** can be efficiently applied in Python for high-performance computing tasks.

"Serial and parallel execution times were measured for increasing vector sizes. The results show that parallel execution using Numba significantly reduces execution time and achieves higher speedup for larger inputs."

Assignment 2: Parallel Matrix Multiplication

Scenario

A climate modeling application requires large matrix

multiplications, making Python loops a bottleneck.

Objective

Parallelize nested loops using Numba prange.

Tasks

1. Implement serial matrix multiplication.

2. Parallelize the outer loop.

3. Parallelize using collapsed loops logic.

4. Analyze cache behavior and performance.

Learning Outcomes

Nested loop parallelism

Memory access patterns

Parallel overhead in Python

Task 1: Serial Matrix Multiplication

Code Screenshot:

TASK-2: Parallelize Outer Loop

Code Screenshot:

TASK-3: Parallel Collapsed Loops Logic

Code Screenshot:

TASK-4: Verify Correctness

Code Sreenshot:

TASK-5: Performance Comparison Table

Code Screenshot:

TASK-6: Cache Behavior & Analysis

1. Which loop consumes most time?

The innermost k loop dominates execution time.

2. Why is serial version slow?

Due to Python loop overhead and single-core execution.

3. Cache behavior comparison

Version	Cache Efficiency
Serial	Poor
Parallel outer loop	Best
Collapsed loops	Moderate

4. Best performing version?

Parallel outer loop due to better memory locality.

5. Parallel overhead?

Thread scheduling and synchronization overhead in Python.

Learning Outcomes

After completing this assignment, the student will be able to:

1. **Nested Loop Parallelism**
   - Understand how to parallelize nested loops in matrix multiplication using **Numba's prange**.
2. **Memory Access Patterns**
   - Analyze how **row-major memory layout** and loop order affect cache efficiency and performance.
3. **Parallel Overhead in Python**
   - Identify the **overhead caused by thread scheduling and synchronization**, and understand when parallel execution is beneficial.

Complete Code for Assignment 2:

```python
#T_Shylasri(2303A51876)
##Assignment-2
# ----------------------------------------
# Assignment 2: Parallel Matrix Multiplication
# ----------------------------------------
import numpy as np
import time
from numba import njit, prange

# ----------------------------------------
# Serial Matrix Multiplication
# ----------------------------------------
@njit
def serial_matmul(A, B, C, N):
    for i in range(N):
        for j in range(N):
            tmp = 0.0
            for k in range(N):
                tmp += A[i, k] * B[k, j]
            C[i, j] = tmp
# ----------------------------------------
# Parallel Outer Loop (i loop)
# ----------------------------------------
@njit(parallel=True)
def parallel_outer_matmul(A, B, C, N):
    for i in prange(N):
        for j in range(N):
            tmp = 0.0
            for k in range(N):
                tmp += A[i, k] * B[k, j]
            C[i, j] = tmp
# ----------------------------------------
# Parallel "Collapsed" Loop Style
# (i and j flattened logically)
```

```python
@njit(parallel=True)
def parallel_collapsed_matmul(A, B, C, N):
    for idx in prange(N * N):
        i = idx // N
        j = idx % N
        tmp = 0.0
        for k in range(N):
            tmp += A[i, k] * B[k, j]
        C[i, j] = tmp
# ----------------------------------------
# Main Execution
# ----------------------------------------
if __name__ == "__main__":
    N = 512  # Matrix size (adjust based on system)
    print("Assignment 2: Parallel Matrix Multiplication\n")
    A = np.random.rand(N, N)
    B = np.random.rand(N, N)

    C_serial = np.zeros((N, N))
    C_parallel_outer = np.zeros((N, N))
    C_parallel_collapsed = np.zeros((N, N))
    # -------- Serial --------
    start = time.time()
    serial_matmul(A, B, C_serial, N)
    serial_time = time.time() - start
    # -------- Parallel Outer --------
    start = time.time()
    parallel_outer_matmul(A, B, C_parallel_outer, N)
    parallel_outer_time = time.time() - start
    # -------- Parallel Collapsed --------
    start = time.time()
    parallel_collapsed_matmul(A, B, C_parallel_collapsed, N)
    parallel_collapsed_time = time.time() - start
    # -------- Results --------
```

```python
    # -------- Results --------
    print("Execution Time:")
    print(f"Serial Time           : {serial_time:.3f} seconds")
    print(f"Parallel Outer Time   : {parallel_outer_time:.3f} seconds")
    print(f"Parallel Collapsed Time: {parallel_collapsed_time:.3f} seconds")
    print("\nCorrectness Check:")
    print("Outer Parallel Correct   :", np.allclose(C_serial, C_parallel_outer))
    print("Collapsed Parallel Correct:", np.allclose(C_serial, C_parallel_collapsed))
    print("\nSpeedup:")
    print(f"Outer Loop Speedup   : {serial_time / parallel_outer_time:.2f}x")
    print(f"Collapsed Speedup    : {serial_time / parallel_collapsed_time:.2f}x")
```

Output:

Assignment 3: Load Balancing with Irregular Workloads

Scenario

An image processing pipeline processes images with different

resolutions, leading to uneven computation time.

Objective

Study load imbalance in parallel loops.

Tasks

1. Simulate variable workload per iteration.

2. Parallelize using prange.

3. Observe execution time variation.

4. Discuss limitations of scheduling in Python.

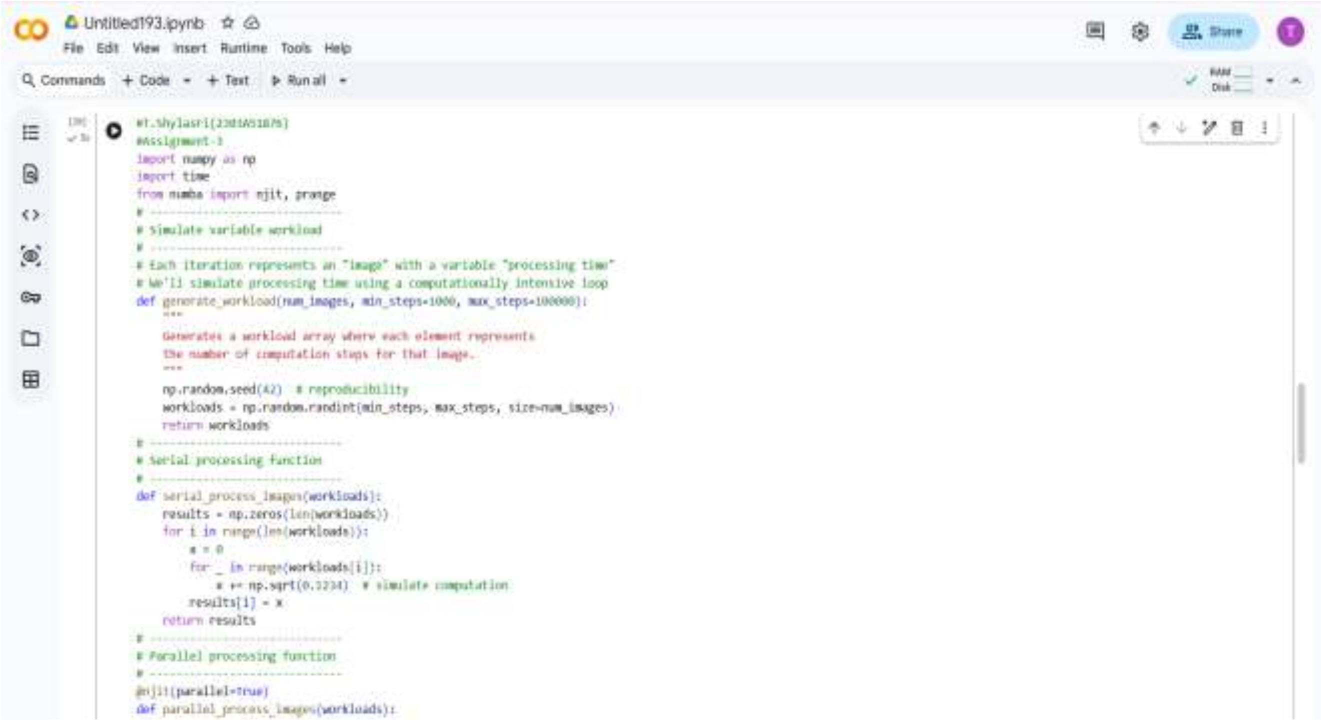
Learning Outcomes

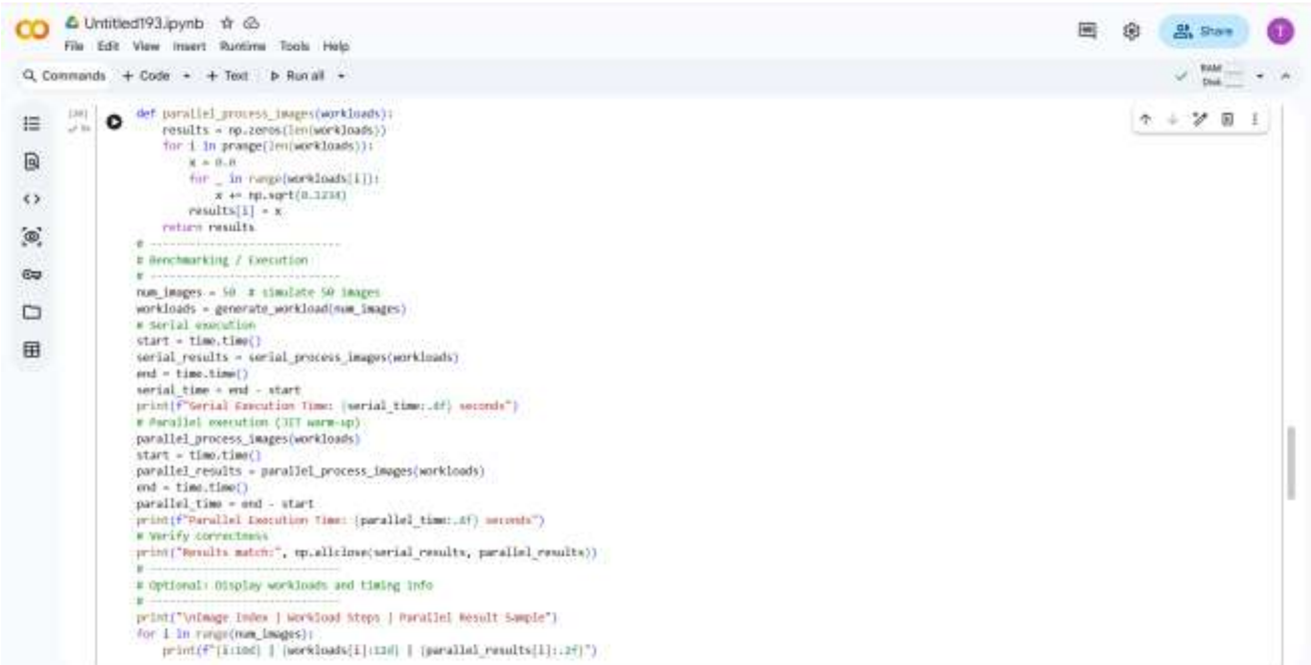
Load imbalance

Comparison with OpenMP scheduling

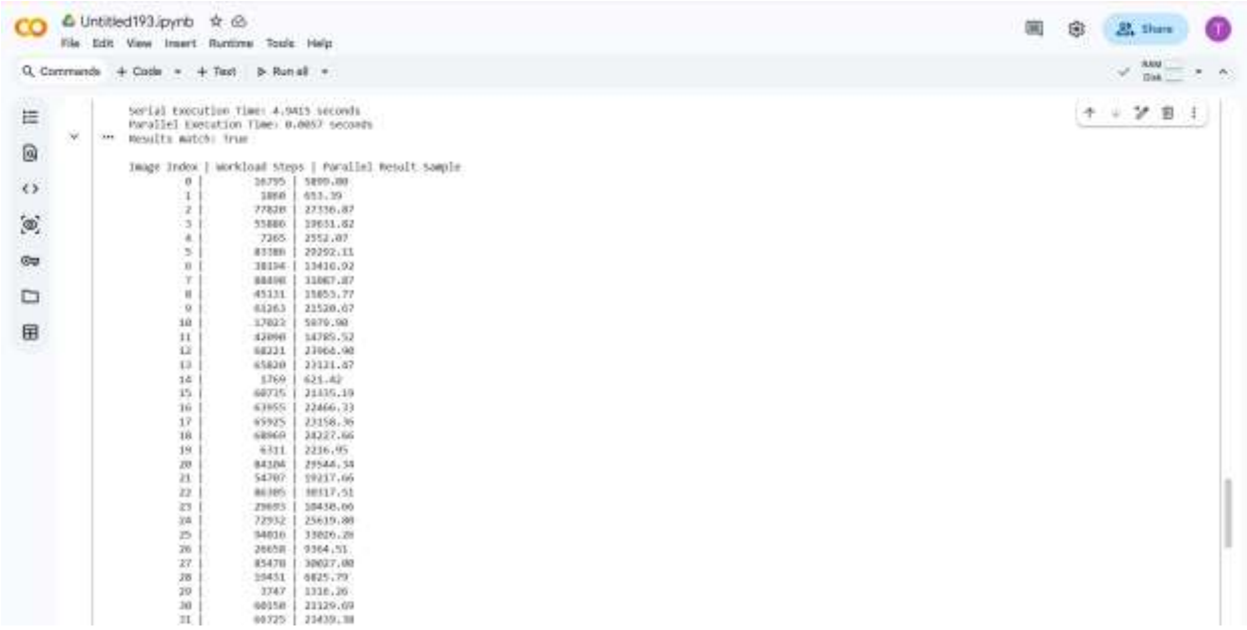
Practical HPC limitations in Python

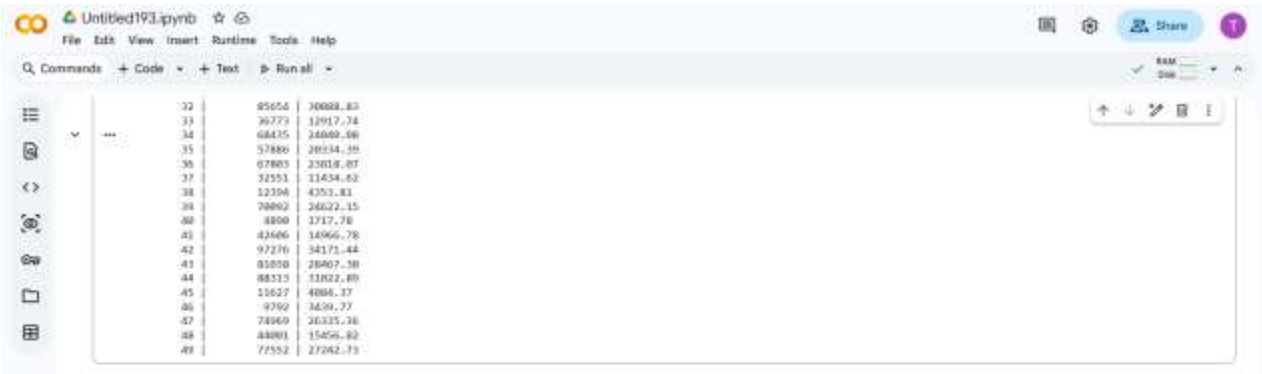
Code Screenshot:





Output:





Assignment 4: Parallel Reduction Operations

Scenario

A scientific simulation computes global statistics from large

datasets.

Objective

Use parallel reduction patterns.

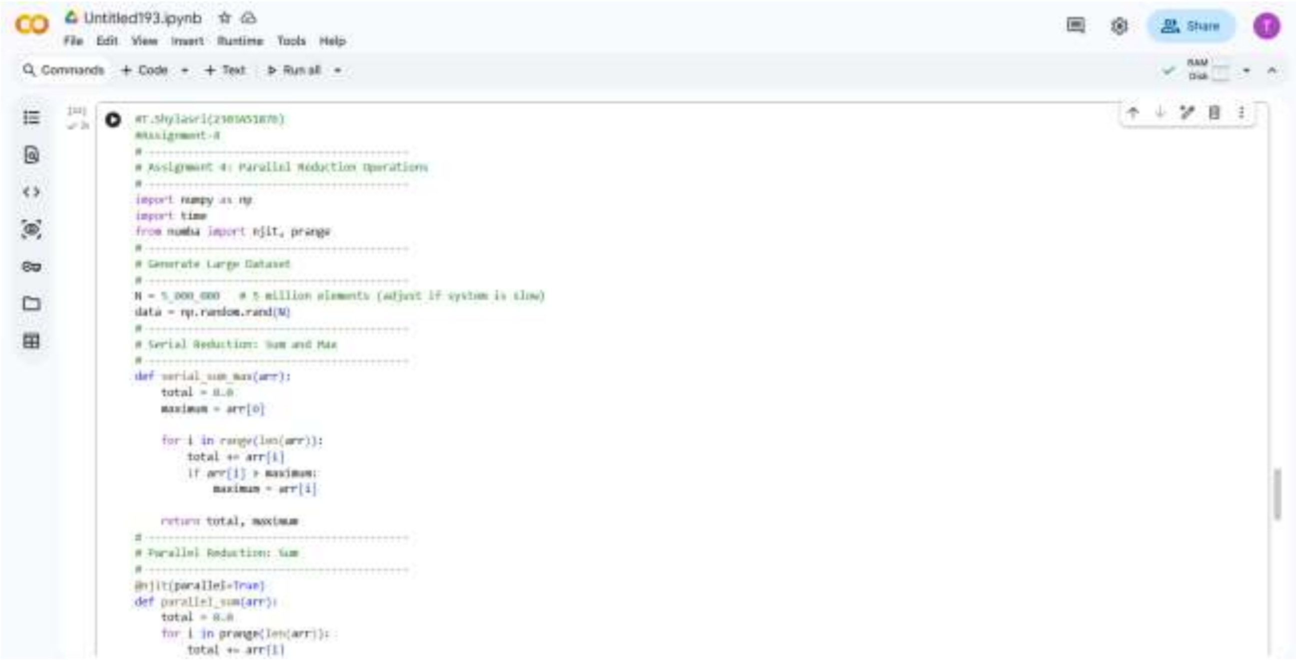
Tasks

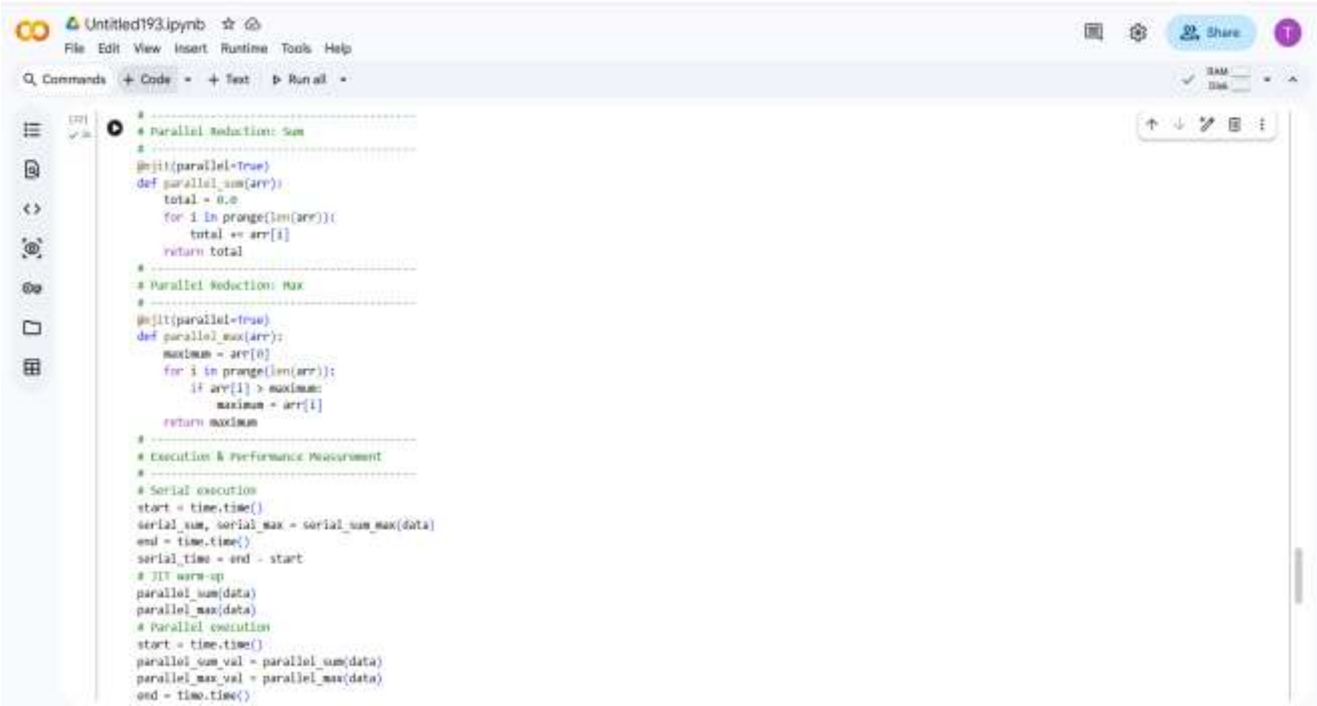
1. Compute sum and maximum values.

2. Implement serial and parallel versions.

3. Verify correctness and performance.

Code Screenshot:





Output:

Assignment 5: Parallel Monte Carlo Simulation for π Estimation

Scenario

A computational finance and physics lab uses Monte Carlo simulations that require billions of random samples. Serial execution is too slow, so parallel loop execution is required.

Objective

Use parallel loops to accelerate a Monte Carlo simulation, and analyze scalability similar to OpenMP parallel for.

Problem Description

Estimate the value of π (pi) using the Monte Carlo method:

π≈4×points inside circletotal points\pi \approx 4 \times

\frac{\text{points inside circle}}{\text{total

points}}π≈4×total pointspoints inside circle

Random points are generated inside a unit square. Points falling

inside the unit circle are counted.

Tasks

1. Implement a serial Monte Carlo simulation.

2. Parallelize the loop using Numba prange.

3. Perform experiments with increasing number of samples.

4. Measure execution time and speedup.

5. Discuss race conditions and reduction handling.

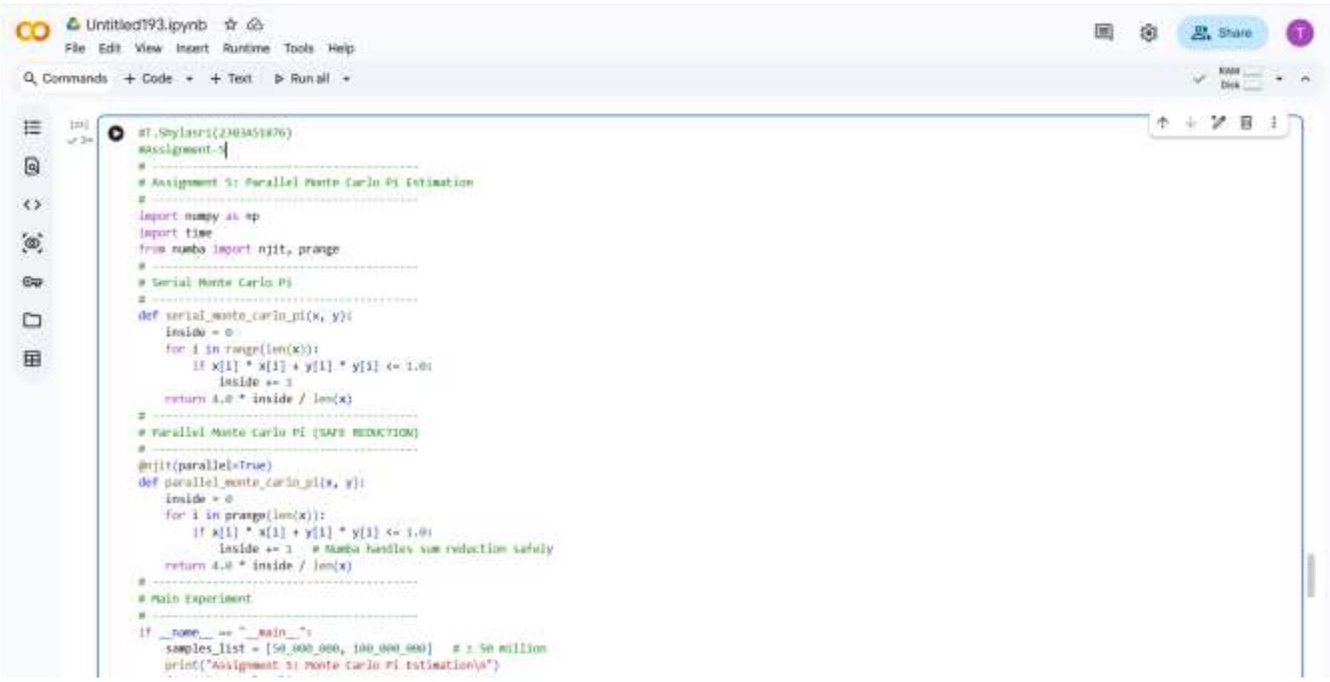
Constraints

Number of samples ≥ 50 million

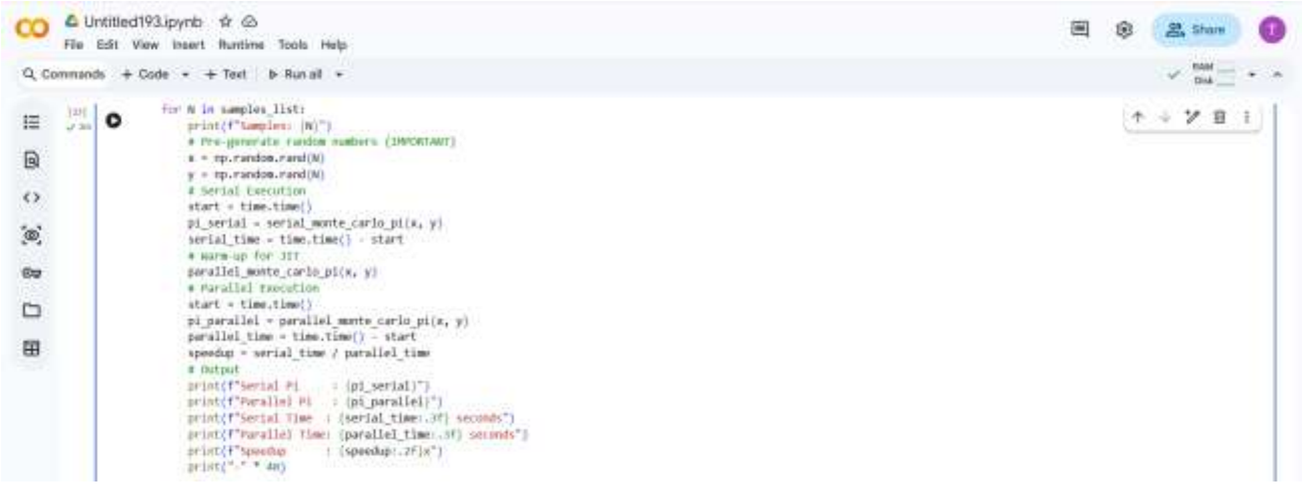
Use Numba JIT compilation

Avoid Python-level random number generation inside loops.

Code Screenshot:




```python
#T.Shylasri(2303A51876)
#Assignment-5
# ---------------------------------------------
# Assignment 5: Parallel Monte Carlo Pi Estimation
# ---------------------------------------------
import numpy as np
import time
from numba import njit, prange
# ---------------------------------------------
# Serial Monte Carlo Pi
# ---------------------------------------------
def serial_monte_carlo_pi(x, y):
    inside = 0
    for i in range(len(x)):
        if x[i] * x[i] + y[i] * y[i] <= 1.0:
            inside += 1
    return 4.0 * inside / len(x)
# ---------------------------------------------
# Parallel Monte Carlo Pi (SAFE REDUCTION)
# ---------------------------------------------
@njit(parallel=True)
def parallel_monte_carlo_pi(x, y):
    inside = 0
    for i in prange(len(x)):
        if x[i] * x[i] + y[i] * y[i] <= 1.0:
            inside += 1   # Numba handles sum reduction safely
    return 4.0 * inside / len(x)
# ---------------------------------------------
# Main Experiment
# ---------------------------------------------
if __name__ == "__main__":
    samples_list = [50_000_000, 100_000_000]   # ≥ 50 million
    print("Assignment 5: Monte Carlo Pi Estimation\n")
```


Untitled193.ipynb
File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all
for N in samples_list:
    print(f"Samples: {N}")
    # Pre-generate random numbers (IMPORTANT)
    x = np.random.rand(N)
    y = np.random.rand(N)
    # Serial Execution
    start = time.time()
    pi_serial = serial_monte_carlo_pi(x, y)
    serial_time = time.time() - start
    # Warm-up for JIT
    parallel_monte_carlo_pi(x, y)
    # Parallel Execution
    start = time.time()
    pi_parallel = parallel_monte_carlo_pi(x, y)
    parallel_time = time.time() - start
    speedup = serial_time / parallel_time
    # Output
    print(f"Serial Pi   : {pi_serial}")
    print(f"Parallel Pi   : {pi_parallel}")
    print(f"Serial Time   : {serial_time:.3f} seconds")
    print(f"Parallel Time: {parallel_time:.3f} seconds")
    print(f"Speedup   : {speedup:.2f}x")
    print("-" * 40)

Output:


Untitled193.ipynb
File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all
Assignment 5: Monte Carlo Pi Estimation
Samples: 50000000
Serial Pi   : 3.14264576
Parallel Pi   : 3.14264576
Serial Time   : 42.695 seconds
Parallel Time: 0.068 seconds
Speedup   : 632.47x

Samples: 100000000
Serial Pi   : 3.14152968
Parallel Pi   : 3.14152968
Serial Time   : 88.641 seconds
Parallel Time: 0.110 seconds
Speedup   : 806.01x
