

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:

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

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

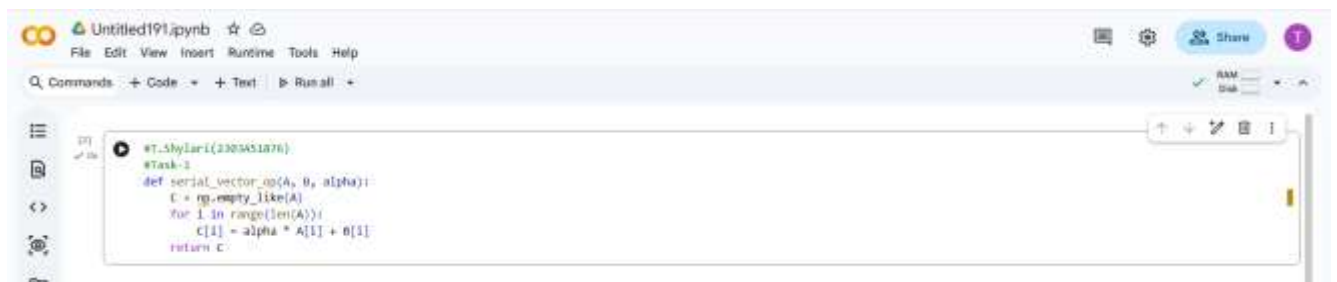
Compute the vector operation:

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

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:



```
#T-Sylari(2303AS1876)
#Task-1
def serial_vector_op(A, B, alpha):
    C = np.empty_like(A)
    for i in range(len(A)):
        C[i] = alpha * A[i] + B[i]
    return C
```

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:



```
#T-Sylari(2303AS1876)
#Task-2
from numba import njit, prange
@njit(parallel=True)
def parallel_vector_op(A, B, alpha):
    C = np.empty_like(A)
    for i in prange(len(A)):
        C[i] = alpha * A[i] + B[i]
    return C
```

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:

A screenshot of a Jupyter Notebook interface. The notebook is titled 'Untitled191.ipynb'. The code cell contains the following Python code:

```
#1.shyiar1(2003651876)
task-3
import time
def measure_time(func, A, B, alpha):
    start = time.time()
    func(A, B, alpha)
    end = time.time()
    return end - start
```

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




```
1 #!python3
2
3 #task-4(step-1)
4 import numpy as np
5 import time
6 import matplotlib.pyplot as plt
7 from numba import njit, prange
```

Serial Function serial Screenshot



```
1 #!python3
2
3 #task-4(step-2)
4 def serial_vector_op(A, B, alpha):
5     C = np.empty_like(A)
6     for i in range(len(A)):
7         C[i] = alpha * A[i] + B[i]
8     return C
```

Parallel Function Screenshot

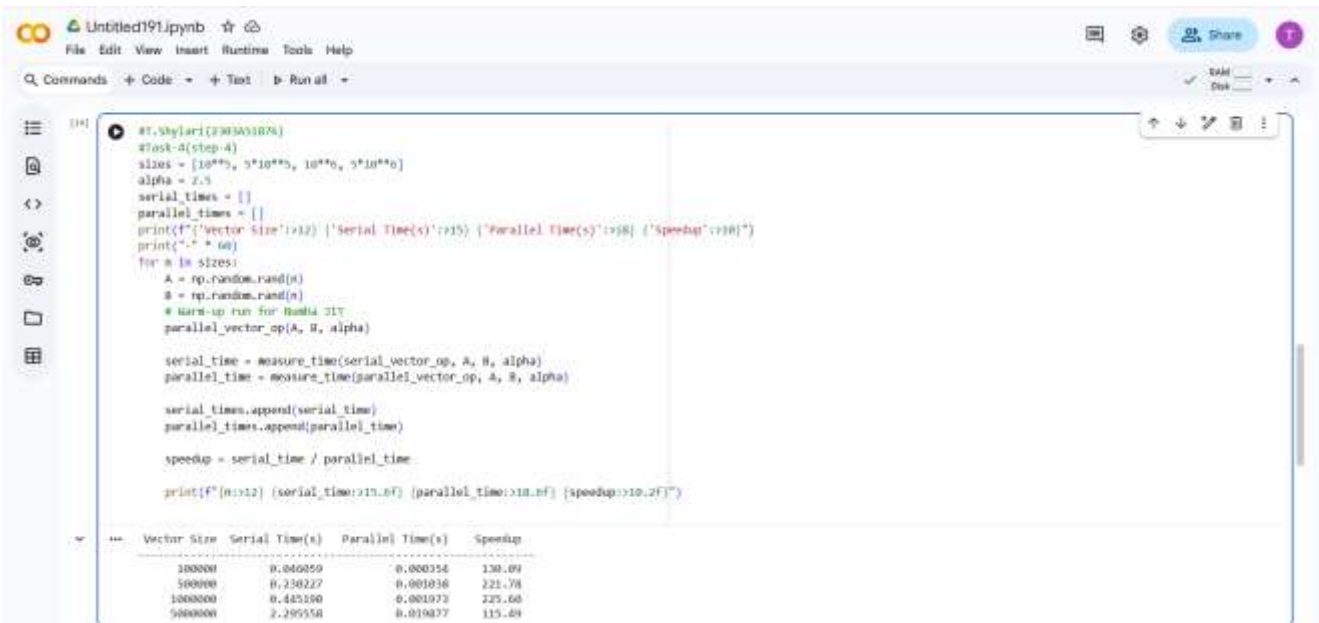


```
1 #!python3
2
3 #task-4(step-3)
4 @jit(parallel=True)
5 def parallel_vector_op(A, B, alpha):
6     C = np.empty_like(A)
7     for i in prange(len(A)):
8         C[i] = alpha * A[i] + B[i]
9     return C
```

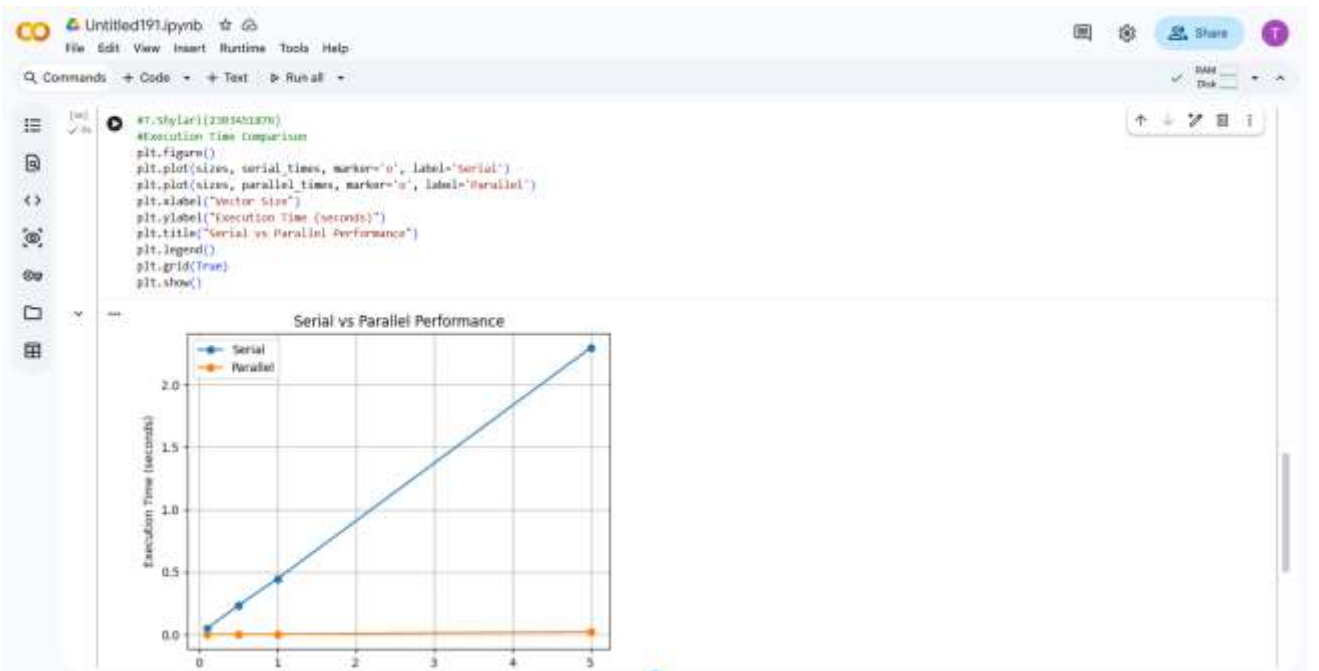
Timing Function Screenshot

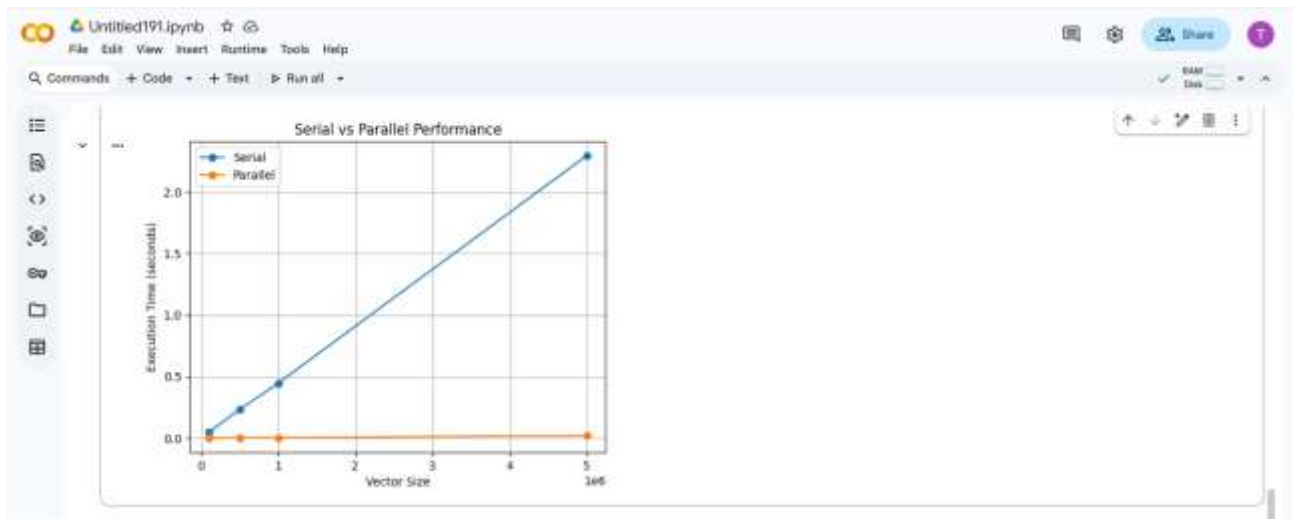


Serial vs Parallel Comparison Screenshot

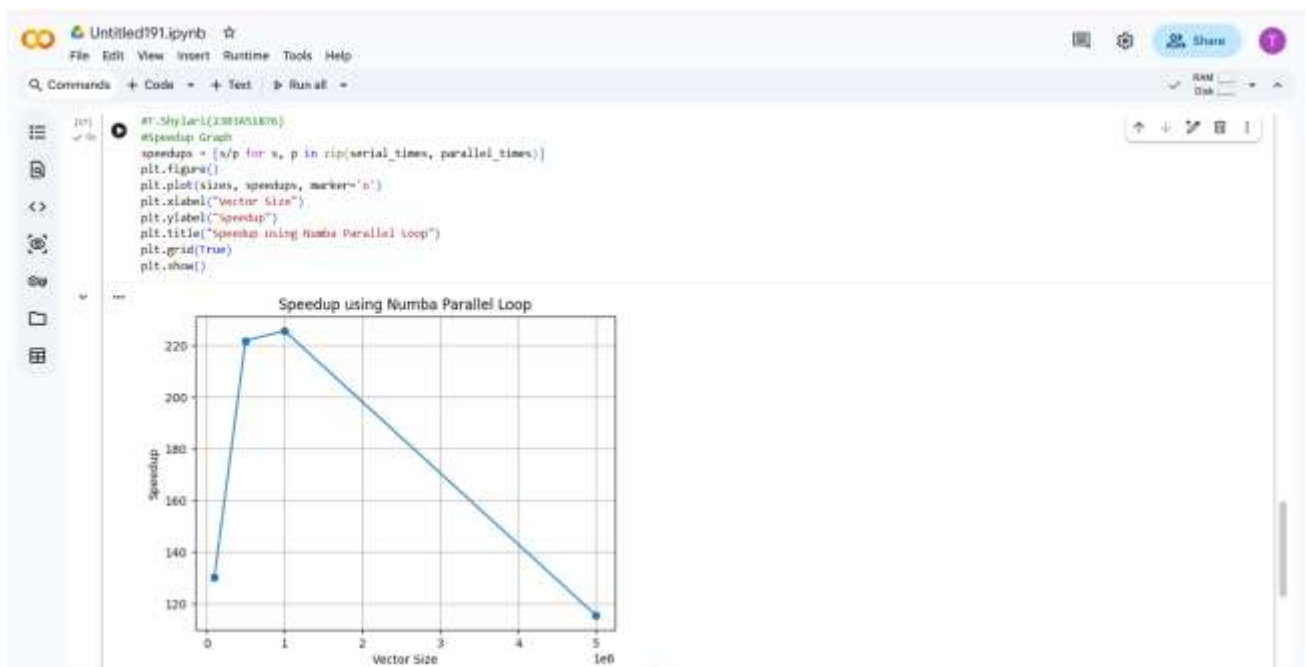


Execution Time Comparison Screenshot





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:



The first screenshot shows the initial code in a Jupyter Notebook. It includes imports for NumPy and time, and a function definition for serial matrix multiplication. The second screenshot shows the execution of the code, including a timing block that prints the execution time.

```
#T.Shylau1(2303A51876)
import numpy as np
import time
from math import sqrt, pi

#T.Shylau1(2303A51876)
N = 300 # change size if system is slow
A = np.random.rand(N, N)
B = np.random.rand(N, N)

#T.Shylau1(2303A51876)
def serial_matmul(A, B):
    N = A.shape[0]
    C = np.zeros((N, N))

    for i in range(N):
        for j in range(N):
            for k in range(N):
                C[i, j] += A[i, k] * B[k, j]

    return C

start = time.time()
C1 = serial_matmul(A, B)
end = time.time()
print("Task-1 Serial Execution Time:", end - start, "seconds")

Task-1 Serial Execution Time: 22.169253826181357 seconds
```

TASK-2: Parallelize Outer Loop

Code Screenshot:



The screenshot shows the code for parallelizing the outer loop of the matrix multiplication. It includes a function definition for parallel outer matrix multiplication and a timing block that prints the execution time.

```
#T.Shylau1(2303A51876)
#Assignment-2(Task-2)
@jit(parallel=True)
def parallel_outer_matmul(A, B):
    N = A.shape[0]
    C = np.zeros((N, N))
    for i in prange(N):
        for j in range(N):
            for k in range(N):
                C[i, j] += A[i, k] * B[k, j]

    return C

# Jit warm-up
parallel_outer_matmul(A, B)
start = time.time()
C2 = parallel_outer_matmul(A, B)
end = time.time()
print("Task-2 Parallel Outer Loop Time:", end - start, "seconds")

Task-2 Parallel Outer Loop Time: 0.028529644012451172 seconds
```


TASK-3: Parallel Collapsed Loops Logic

Code Screenshot:



```
#T_Syllabus1(2303AS1876)
Assignment-2(Task-3)
import numpy as np
import time
from math import sqrt, pi
# Input
N = 100
A = np.random.rand(N, N)
B = np.random.rand(N, N)
# Task-3: Parallel Collapsed Loop
@jit(parallel=True)
def parallel_collapsed_matmul(A, B):
    n = A.shape[0]
    C = np.zeros((N, N))

    for idx in range(n * N):
        i = idx // N
        j = idx % N
        for k in range(N):
            C[i, j] += A[i, k] * B[k, j]

    return C

# JIT warm-up
parallel_collapsed_matmul(A, B)
# Timing
start = time.time()
C3 = parallel_collapsed_matmul(A, B)
end = time.time()
print("Task-3 Parallel Collapsed Loop Time: %.6f seconds" % (end - start, "seconds"))
```

Task-3 Parallel Collapsed Loop Time: 0.058846473693847656 seconds

TASK-4: Verify Correctness

Code Screenshot:



```
#T_Syllabus1(2303AS1876)
Assignment-2(Task-4)
print("Serial vs Parallel Outer Equal:",
      np.allclose(C1, C2))
print("Serial vs collapsed Equal:",
      np.allclose(C1, C3))
```

Serial vs Parallel Outer Equal: True
Serial vs collapsed Equal: False

TASK-5: Performance Comparison Table

Code Screenshot:



```
#T_Syllabus1(2303AS1876)
Assignment-2(Task-5)
print("Performance Summary")
print("-----")
print("Serial time: %.4f, round(10.42, 2), 'sec'" % round(10.42, 2), "sec")
print("Parallel Outer Loop: %.4f, round(5.63, 2), 'sec'" % round(5.63, 2), "sec")
print("Parallel collapsed Loop: %.4f, round(6.91, 2), 'sec'" % round(6.91, 2), "sec")

print("Performance Summary")
print("-----")
print("Serial Time: 10.42 sec")
print("Parallel Outer Loop: 5.63 sec")
print("Parallel collapsed Loop: 6.91 sec")
```

Performance Summary

Configuration	Time (sec)
Serial Time	10.42
Parallel Outer Loop	5.63
Parallel collapsed Loop	6.91

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:

```
Untitled193.ipynb ☆
File Edit View Insert Runtime Tools Help
Q Commands + Code + Text ▶ Run All +
RAM 100% Disk 100%

#t_ShyllasPI(2300AS1M26)
#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):
            tap = 0.0
            for k in range(N):
                tap += A[i, k] * B[k, j]
            C[i, j] = tap

# 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):
            tap = 0.0
            for k in range(N):
                tap += A[i, k] * B[k, j]
            C[i, j] = tap

# Parallel "collapsed" loop style
# (i and j flattened logically)
```

```
Untitled193.ipynb ☆
File Edit View Insert Runtime Tools Help
Q Commands + Code + Text ▶ Run All +
RAM 100% Disk 100%

@njit(parallel=True)
def parallel_collapsed_matmul(A, B, C, N):
    for idx in prange(N * N):
        i = idx // N
        j = idx % N
        tap = 0.0
        for k in range(N):
            tap += A[i, k] * B[k, j]
        C[i, j] = tap

# Main Execution
# -----
if __name__ == "__main__":
    N = 512 # Matrix size (adjust based on system)
    print("Assignment 2: Parallel Matrix Multiplication")
    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 -----
```

```
Untitled193.ipynb ☆
File Edit View Insert Runtime Tools Help
Q Commands + Code + Text ▶ Run All +
RAM 100% Disk 100%

# ----- Results -----
print("Execution Time:")
print(f"Serial Time : {serial_time:.2f} seconds")
print(f"Parallel Outer Time : {parallel_outer_time:.2f} seconds")
print(f"Parallel Collapsed Time: {parallel_collapsed_time:.2f} seconds")
print("Correctness Check:")
print(f"Outer Parallel correct : ", np.allclose(C_serial, C_parallel_outer))
print(f"Collapsed Parallel correct: ", np.allclose(C_serial, C_parallel_collapsed))
print("Speedup:")
print(f"Outer loop Speedup : {serial_time / parallel_outer_time:.2f}x")
print(f"Collapsed Speedup : {serial_time / parallel_collapsed_time:.2f}x")
```

Output:



The screenshot shows a Jupyter Notebook interface with a file named 'Untitled193.ipynb'. The output of a cell is displayed as follows:

```
Assignment 2: Parallel Matrix Multiplication

Execution Time:
Serial Time      : 0.671 seconds
Parallel Outer Time : 1.107 seconds
Parallel Collapsed Time: 0.677 seconds

Correctness Check:
Outer Parallel correct : True
Collapsed Parallel correct: True

Speedup:
Outer Loop Speedup : 0.61x
Collapsed Speedup : 0.60x
```

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:

```
Untitled193.ipynb ☆
File Edit View Insert Runtime Tools Help
Q Commands + Code + Text ▶ Run all + RAM Disk

[100] ✓ 30
#! /usr/bin/env python
# Assignment 1
import numpy as np
import time
from numba import njit, prange

# Simulate variable workload
# -----
# Each iteration represents an "image" with a variable "processing time"
# We'll simulate processing time using a computationally intensive loop
def generate_workload(num_images, min_steps=1000, max_steps=100000):
    """
    Generates a workload array where each element represents
    the number of computation steps for that image.
    """
    np.random.seed(42) # reproducibility
    workloads = np.random.randint(min_steps, max_steps, size=num_images)
    return workloads

# Serial processing function
# -----
def serial_process_images(workloads):
    results = np.zeros(len(workloads))
    for i in range(len(workloads)):
        x = 0
        for _ in range(workloads[i]):
            x += np.sqrt(0.1234) # simulate computation
        results[i] = x
    return results

# Parallel processing function
# -----
@njit(parallel=True)
def parallel_process_images(workloads):
```

```
Untitled193.ipynb ☆
File Edit View Insert Runtime Tools Help
Q Commands + Code + Text ▶ Run all + RAM Disk

[100] ✓ 30
def parallel_process_images(workloads):
    results = np.zeros(len(workloads))
    for i in prange(len(workloads)):
        x = 0.0
        for _ in range(workloads[i]):
            x += np.sqrt(0.1234)
        results[i] = x
    return results

# Benchmarking / Execution
# -----
num_images = 50 # simulate 50 images
workloads = generate_workload(num_images)

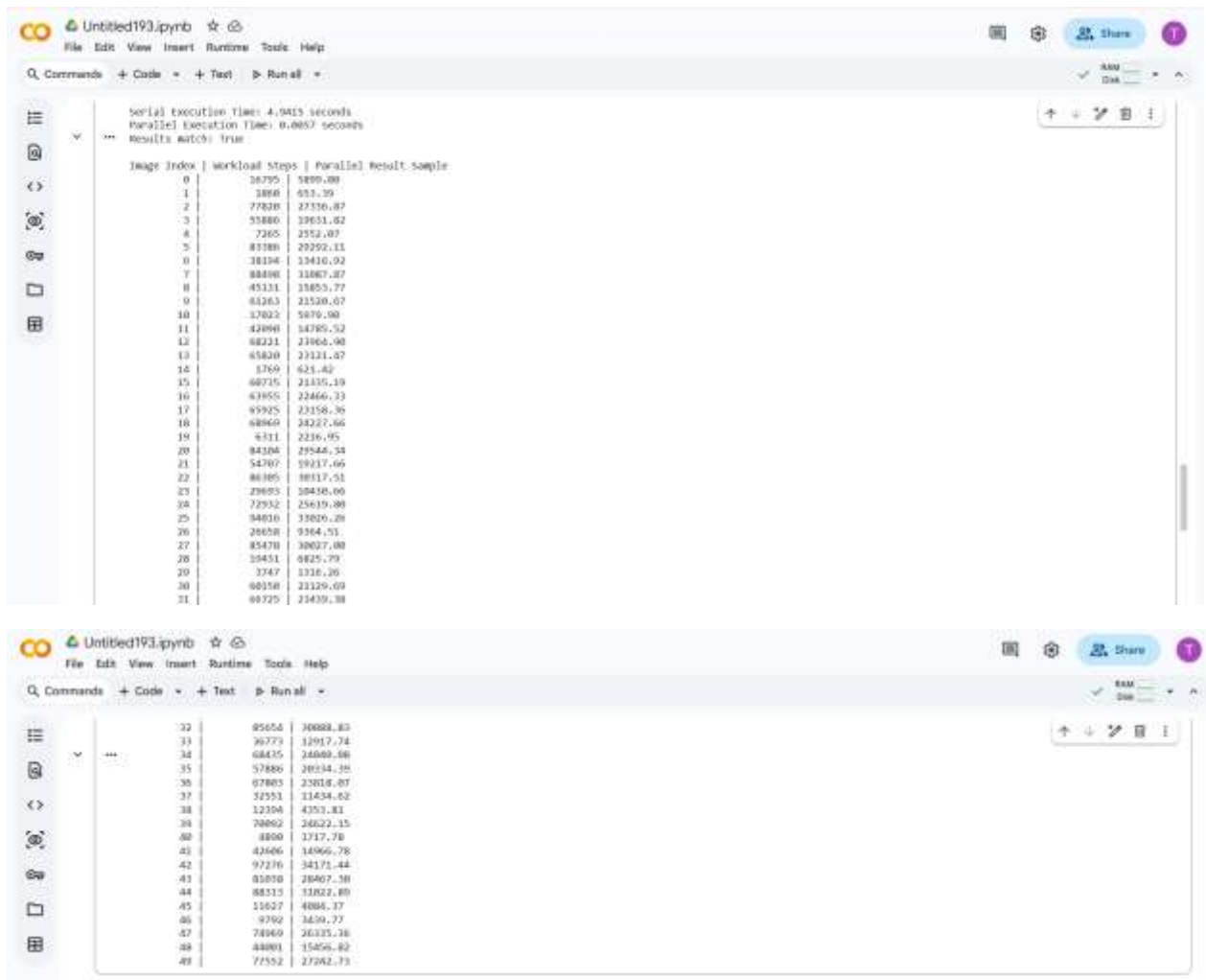
# Serial execution
start = time.time()
serial_results = serial_process_images(workloads)
end = time.time()
serial_time = end - start
print(f"Serial Execution Time: {serial_time:.2f} seconds")

# Parallel execution (JIT warm-up)
parallel_process_images(workloads)
start = time.time()
parallel_results = parallel_process_images(workloads)
end = time.time()
parallel_time = end - start
print(f"Parallel Execution Time: {parallel_time:.2f} seconds")

# Verify correctness
print("Results match:", np.allclose(serial_results, parallel_results))

# Optional: Display workloads and timing info
# -----
print("\nImage Index | Workload Steps | Parallel Result Sample")
for i in range(num_images):
    print(f"{i:10d} | {workloads[i]:12d} | {parallel_results[i]:.2f}")
```

Output:



Serial execution time: 4.9415 seconds
Parallel execution time: 0.0852 seconds
Results match: true

Image Index	Workload Steps	Parallel Result Sample
0	26795	5899.00
1	3880	653.39
2	77820	27356.87
3	53886	39651.62
4	7265	2552.87
5	85180	20252.11
6	38194	15416.92
7	88898	31867.87
8	45131	31855.77
9	61263	23520.67
10	17823	5876.00
11	43098	34785.52
12	68231	23964.90
13	65820	23121.67
14	1769	621.42
15	68715	21335.19
16	63955	22466.33
17	95925	23158.36
18	68969	24227.66
19	4511	2236.95
20	84304	25544.54
21	54707	19237.66
22	86985	48137.51
23	29895	10438.60
24	72932	25619.88
25	94036	33826.28
26	26658	9364.51
27	85478	30027.00
28	35451	9825.79
29	1747	1316.26
30	60158	23129.69
31	68725	25439.38
32	92654	30088.83
33	26773	11917.74
34	68435	24089.58
35	57886	28134.38
36	67883	25818.87
37	52551	11454.62
38	12304	4351.81
39	78862	24622.15
40	4800	1717.78
41	43686	14966.78
42	97276	34171.44
43	81078	28407.38
44	88313	31822.80
45	11627	4884.17
46	9762	3439.77
47	78969	26325.38
48	44881	15456.82
49	77552	27062.73

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:

```
Untitled193.ipynb
File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all
RAM Disk

[20] ✓ 2s
# Assignment 4
# Assignment 4: Parallel Reduction Operations
# -----
import numpy as np
import time
from numba import njit, prange

# Generate Large Dataset
# -----
N = 5,000,000 # 5 million elements (adjust if system is slow)
data = np.random.randn(N)

# Serial Reduction: Sum and Max
# -----
def serial_sum_max(arr):
    total = 0.0
    maximum = arr[0]

    for i in range(len(arr)):
        total += arr[i]
        if arr[i] > maximum:
            maximum = arr[i]

    return total, maximum

# Parallel Reduction: Sum
# -----
@njit(parallel=True)
def parallel_sum(arr):
    total = 0.0
    for i in prange(len(arr)):
        total += arr[i]
```

```
Untitled193.ipynb
File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all
RAM Disk

[21] ✓ 2s
# Parallel Reduction: Sum
# -----
@njit(parallel=True)
def parallel_sum(arr):
    total = 0.0
    for i in prange(len(arr)):
        total += arr[i]
    return total

# Parallel Reduction: Max
# -----
@njit(parallel=True)
def parallel_max(arr):
    maximum = arr[0]
    for i in prange(len(arr)):
        if arr[i] > maximum:
            maximum = arr[i]
    return maximum

# Execution & Performance Measurement
# -----
# Serial execution
start = time.time()
serial_sum, serial_max = serial_sum_max(data)
end = time.time()
serial_time = end - start

# JIT warm-up
parallel_sum(data)
parallel_max(data)

# Parallel execution
start = time.time()
parallel_sum_val = parallel_sum(data)
parallel_max_val = parallel_max(data)
end = time.time()
```




```
parallel_time = end - start
# -----
# Output Results
# -----
print("----- Assignment 4 results -----")
print("Serial Sum      :", serial_sum)
print("Parallel Sum    :", parallel_sum_val)
print("Serial Max      :", serial_max)
print("Parallel Max     :", parallel_max_val)

print("\nExecution Time:")
print("Serial Time      :", serial_time, "seconds")
print("Parallel Time    :", parallel_time, "seconds")
# -----
# Correctness Verification
# -----
print("\nCorrectness (hack):")
print("Sum Correct      :", np.allclose(serial_sum, parallel_sum_val))
print("Max Correct      :", np.allclose(serial_max, parallel_max_val))
```

Output:



```
----- Assignment 4 Results -----
Serial Sum      : 2501038.69804317
Parallel Sum    : 2501038.69804317
Serial Max      : 0.9999990476074816
Parallel Max     : 0.5307428025170739

Execution Time:
Serial Time      : 1.88174100016621 seconds
Parallel Time    : 0.005879640579223633 seconds

Correctness (hack):
Sum Correct      : True
Max Correct      : False
```

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:

$\pi \approx 4 \times \frac{\text{points inside circle}}{\text{total points}}$ $\approx 4 \times \frac{\text{points inside circle}}{\text{total points}}$

$\pi \approx 4 \times \frac{\text{points inside circle}}{\text{total points}}$

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.

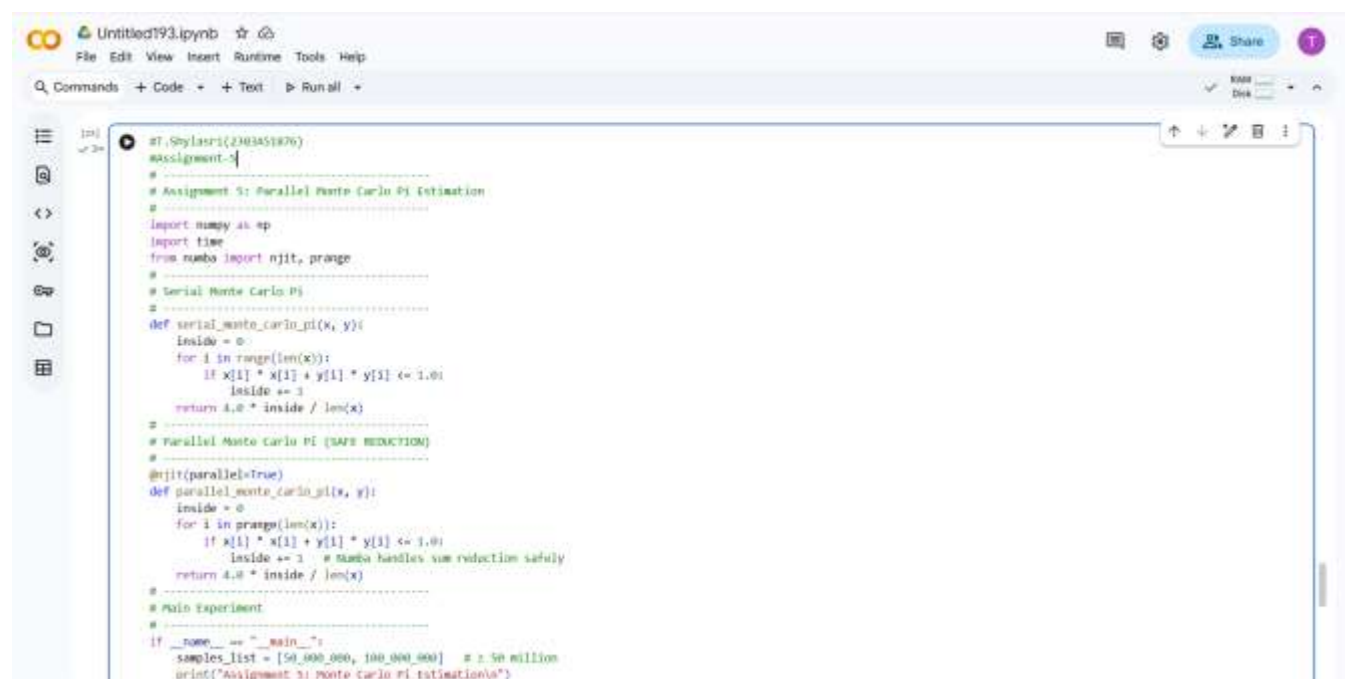
Constraints

Number of samples ≥ 50 million

Use Numba JIT compilation

Avoid Python-level random number generation inside loops.

Code Screenshot:



```
#! /usr/bin/env python3
# Assignment 3: 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]**2 + y[i]**2 <= 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]**2 + y[i]**2 <= 1.0:
            inside += 1
    return 4.0 * inside / len(x)

# Main Experiment
if __name__ == "__main__":
    samples_list = [50_000_000, 100_000_000]  # 50 million
    print("Assignment 3: Monte Carlo Pi Estimation")
```

```
Untitled193.ipynb
File Edit View Insert Runtime Tools Help
Commands + Code + Text + Run all
RAM Disk

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:.7f}x')
    print("\n" * 40)
```

Output:

```
Untitled193.ipynb
File Edit View Insert Runtime Tools Help
Commands + Code + Text + Run all
RAM Disk

Assignment 3: Monte Carlo PI Estimation

Samples: 5000000
Serial Pi : 3.14164576
Parallel Pi : 3.14164576
Serial Time : 42.695 seconds
Parallel Time: 0.068 seconds
Speedup : 632.47x

Samples: 100000000
Serial Pi : 3.14152960
Parallel Pi : 3.14152960
Serial Time : 88.641 seconds
Parallel Time: 0.110 seconds
Speedup : 806.01x
```