

# 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:

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

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

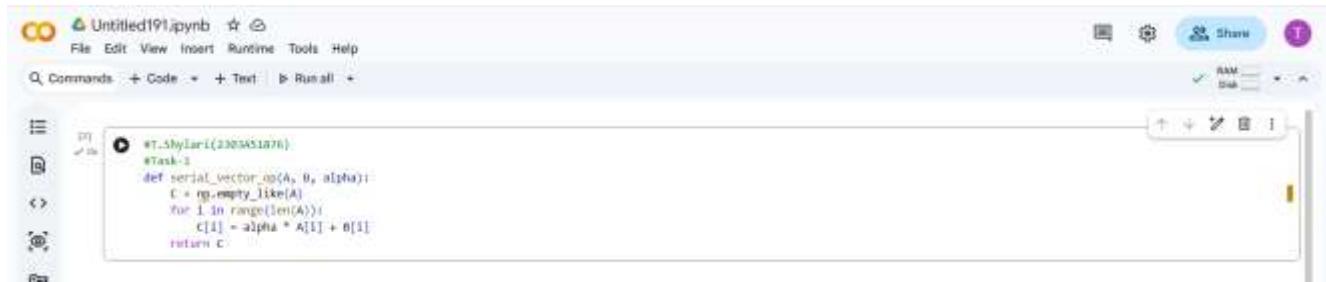
Compute the vector operation:

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

### 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:



A screenshot of a Jupyter Notebook interface. The code cell contains the following Python function:

```
#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:



A screenshot of a Jupyter Notebook interface. The code cell contains the following Python function using Numba:

```
#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:

The screenshot shows a Jupyter Notebook cell with the following code:

```
#1. Shylar((20045107))
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  $5 \times 10^6$ )
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

```
#t_5yLar1Z3H0ASfzS
#task-A(step-1)
import numpy as np
import time
import matplotlib.pyplot as plt
from numba import njit, prange
```

## Serial Function serial Screenshot

```
#t_5yLar1Z3H0ASfzS
#task-A(step-2)
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
```

## Parallel Function Screenshot

```
#t_5yLar1Z3H0ASfzS
#task-A(step-3)
@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
```

## Timing Function Screenshot

A screenshot of a Jupyter Notebook cell. The code defines a function `measure_time(func, A, B, alpha)` that measures the execution time of `func(A, B, alpha)`. It uses the `time.time()` function to record the start and end times, and returns the difference.

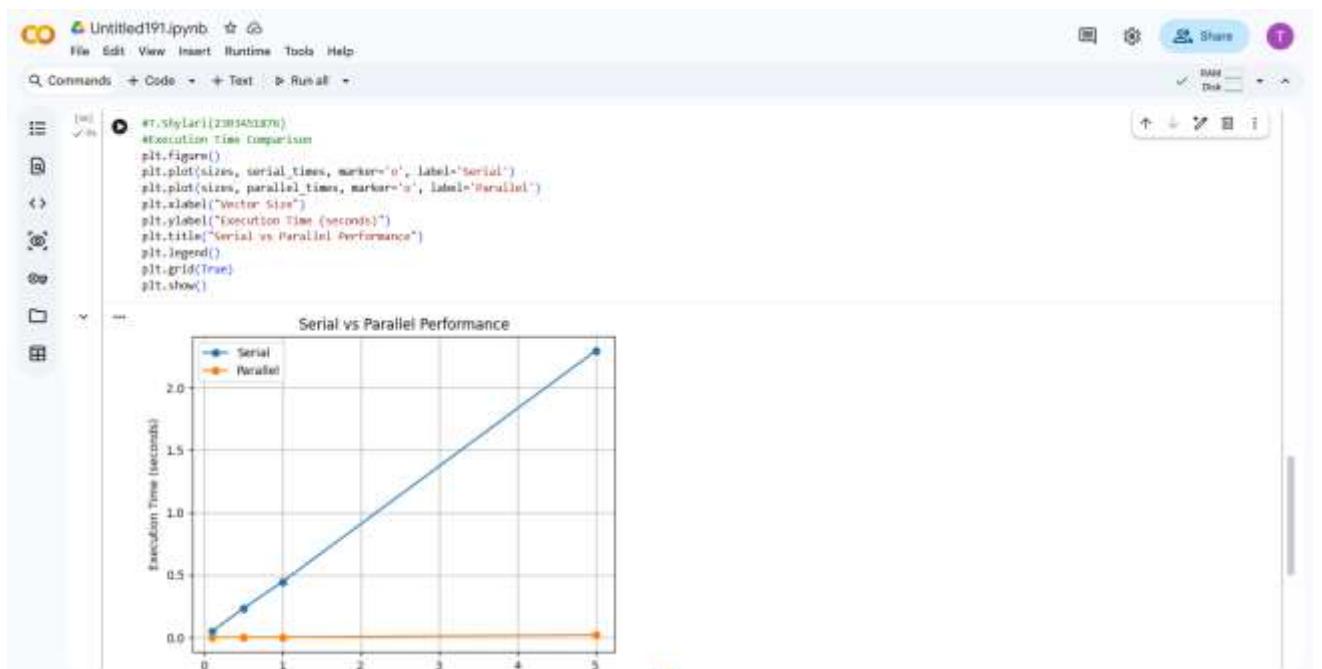
```
#%shylnr(1303401676)
#task-4(stop-3)
def measure_time(func, A, B, alpha):
    start = time.time()
    func(A, B, alpha)
    end = time.time()
    return end - start
```

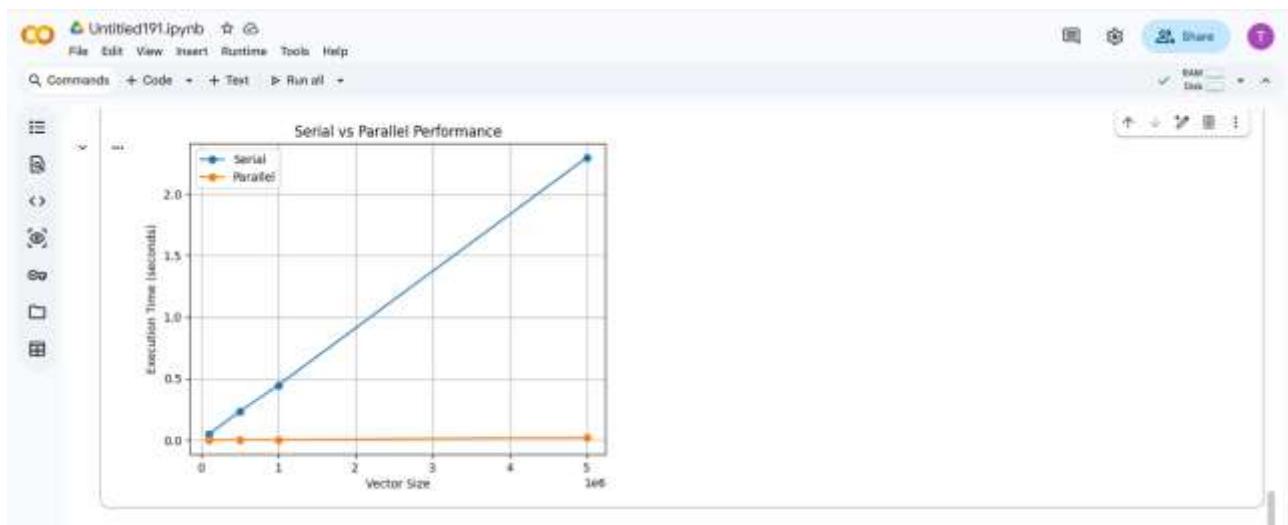
## Serial vs Parallel Comparison Screenshot

A screenshot of a Jupyter Notebook cell. The code performs a series of operations: it defines a function `vector_op`, generates random vectors `A` and `B`, and performs a warm-up run. It then loops through vector sizes `n` from  $10^6$  to  $5 \times 10^6$ , measuring the serial time for `vector_op` and the parallel time for `parallel_vector_op`. It calculates the speedup for each size and prints the results. Below the code, a table summarizes the data.

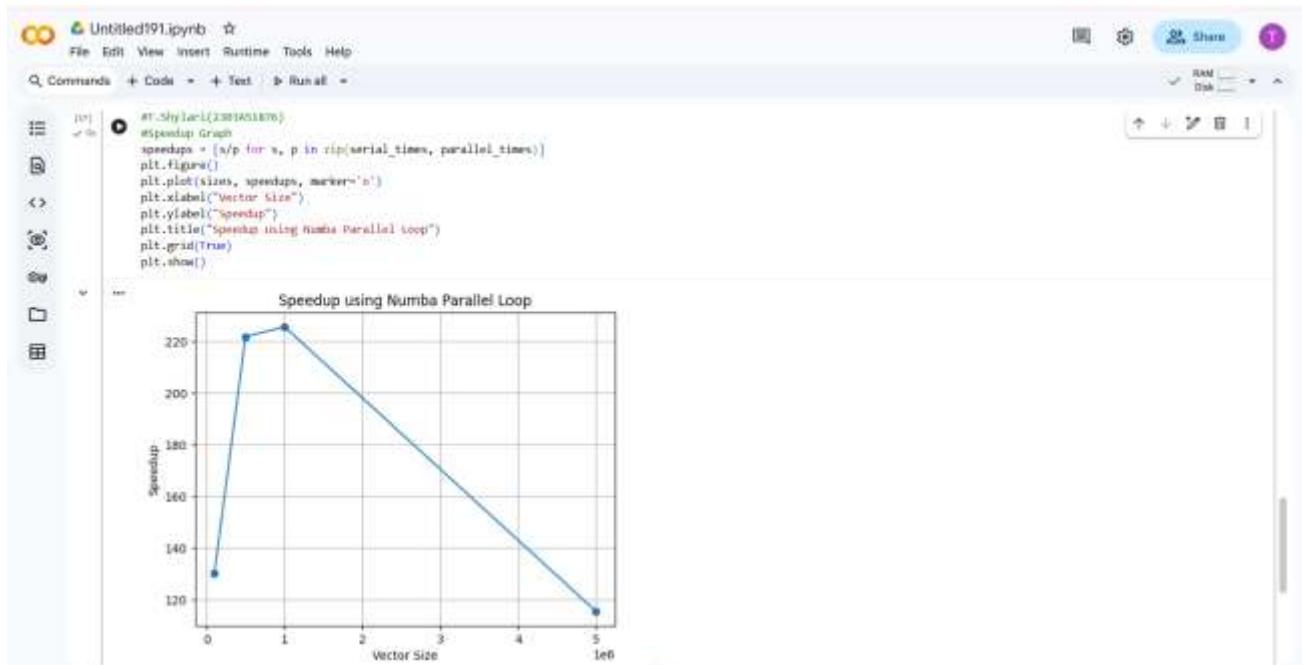
Vector Size	Serial Time(s)	Parallel Time(s)	Speedup
1000000	0.040859	0.000354	118.09
5000000	0.230227	0.001038	221.78
10000000	0.445190	0.001972	225.68
50000000	2.295558	0.019077	115.49

## 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 screenshot shows two code cells in a Jupyter Notebook interface. The first cell contains the following code:#T.Shyamari(2303A531076)
import numpy as np
import time
from numba import jit, prange

#T.Shyamari(2303A531076)
n = 300 # change size if system is slow
A = np.random.rand(n, n)
B = np.random.rand(n, n)The second cell contains the following code:#T.Shyamari(2303A531076)
#Assignment-2[Task-1]
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

#T.Shyamari(2303A531076)
start = time.time()
C1 = serial\_matmul(A, B)
end = time.time()
print("Task-1 serial execution Time:", end - start, "seconds")Output from the second cell shows the execution time for Task-1.

## TASK-2: Parallelize Outer Loop

### Code Screenshot:

The screenshot shows two code cells in a Jupyter Notebook interface. The first cell contains the following code:#T.Shyamari(2303A531076)
#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 CThe second cell contains the following code:# T2I 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")Output from the second cell shows the execution time for Task-2.

## TASK-3: Parallel Collapsed Loops Logic

### Code Screenshot:

The screenshot shows a Jupyter Notebook interface with the following code in a cell:

```
# Shylasri{2303051876}
assignment_2(Task-3)
import numpy as np
import time
from numba import njit, prange
# Input
N = 100
A = np.random.rand(N, N)
B = np.random.rand(N, N)
@njit(parallel=True)
def parallel_collapsed_matmul(A, B):
    N = A.shape[0]
    C = np.zeros((N, N))

    for idx in prange(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 warmup
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:", end - start, "seconds")
```

Output below the cell:

```
task-3 Parallel Collapsed Loop Time: 0.0588364736088258 seconds
```

## TASK-4: Verify Correctness

### Code Screenshot:

The screenshot shows a Jupyter Notebook interface with the following code in a cell:

```
# Shylasri{2303051876}
assignment_2(Task-4)
print("Serial vs Parallel Outer Equal?", 
      np.allclose(C1, C3))
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:

The screenshot shows a Jupyter Notebook interface with the following code in a cell:

```
# Shylasri{2303051876}
assignment_2(Task-5)
print("Performance Summary")
print("-----")
print("Serial Time : ", round(18.42, 2), "sec")
print("Parallel Outer loop : ", round(5.63, 2), "sec")
print("Parallel collapsed loop: ", round(6.91, 2), "sec")
```

Output below the cell:

```
Performance Summary
-----
Serial Time :  18.42 sec
Parallel outer loop :  5.63 sec
Parallel collapsed loop:  6.91 sec
```

## **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**

<b>Version</b>	<b>Cache Efficiency</b>
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 ▶

# et_5hylasfI(2300A51076)
#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, lnp)
#
@njit(parallel=True)
def parallel_outer(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)

```

```
# Untitled193.ipynb  ⭐ ⓘ
File Edit View Insert Runtime Tools Help
Q Commands + Code + Text ▶ Run all ▶

@njit(parallel=True)
def parallel_collapsed(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(A, B, C_parallel_outer, N)
    parallel_outer_time = time.time() - start

    #
    # Parallel collapsed
    #
    start = time.time()
    parallel_collapsed(A, B, C_parallel_collapsed, N)
    parallel_collapsed_time = time.time() - start

    #
    # 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("Uncorrectness 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}s")
print(f"Collapsed Speedup : {(serial_time / parallel_collapsed_time):.2f}s")
```

## Output:

```
Assignment 3: Parallel Matrix Multiplication

Execution Time:
Serial Time      : 0.671 seconds
Parallel Outer Time : 1.107 seconds
Parallel Collapsed Time: 0.977 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:

```
#t.shylasri(2020S1076)
assignment_1
import numpy as np
import time
from numba import jit, 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(0.2234) # simulate computation
        results[i] = x
    return results

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

```
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(0.2234)
        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:.6f} 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:.6f} seconds")
# verify correctness
print("Results match:", np.allclose(serial_results, parallel_results))

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

## Output:

Image Index	Workload Steps
0	58795.00
1	3886.03
2	27336.87
3	55880.02
4	7265.27
5	83181.13
6	38194.92
7	88496.27
8	45131.77
9	63263.07
10	37023.98
11	42091.52
12	68221.98
13	65201.07
14	5769.42
15	66715.19
16	63955.33
17	65985.36
18	68860.27
19	6311.05
20	84104.34
21	54367.66
22	86105.51
23	29895.00
24	72932.89
25	34616.29
26	28658.51
27	85471.00
28	39451.79
29	3747.26
30	66150.69
31	66725.38

Image Index	Workload Steps
32	95655.83
33	36773.74
34	68435.46
35	57886.39
36	97800.87
37	32551.82
38	12350.83
39	70880.25
40	4800.78
41	43460.78
42	97276.44
43	85891.38
44	88315.89
45	11632.17
46	9797.77
47	74949.26
48	44000.82
49	77552.73

## 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:

```
# ar.style.use('classic')
# 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.rand(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]

    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
```

```
# 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("IsCorrectness (True):")
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 : 2500000.69994317
Parallel Sum : 2500000.69994317
Serial Max : 0.9999999476074015
Parallel Max : 0.5307429025178739

Execution Time:
Serial Time : 1.881741000036621 seconds
Parallel Time : 0.005879648579223633 seconds

Correctness check:
Sum Correct : True
Max Correct : False
```

# Assignment 5: Parallel Monte Carlo Simulation for $\pi$

## 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  $\pi$  (pi) using the Monte Carlo method:

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

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

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  $\geq 50$  million

Use Numba JIT compilation

Avoid Python-level random number generation inside loops.

## Code Screenshot:

The screenshot shows a Jupyter Notebook cell with the following code:

```
# 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")
```

A screenshot of a Jupyter Notebook interface. The title bar says "Untitled193.ipynb". The menu bar includes File, Edit, View, Insert, Runtime, Tools, Help. Below the menu is a toolbar with icons for Commands, Code, Text, Run all, and a search bar. The main area contains Python code for estimating pi using Monte Carlo methods. It uses np.random.rand() to generate random numbers, calculates pi using serial and parallel execution, measures execution time, and prints the results.

```
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("\n" * 4)
```

## Output:

A screenshot of a Jupyter Notebook interface showing the output of the previous code. The title bar says "Untitled193.ipynb". The output section displays two sets of results for different sample sizes: 50000000 and 100000000. For each sample size, it prints the estimated pi value, execution time in seconds, and the calculated speedup compared to the serial version.

```
Assignment 3: monte carlo pi estimation
...
Samples: 50000000
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 : 3.14152968
Parallel Pi : 3.14152968
Serial Time : 88.041 seconds
Parallel Time: 0.110 seconds
Speedup : 800.00x
```