High Performance Computing

Name: Thulasi Shylasri

HTNO: 2303A51876

Batch-14

Week-2

LAB Assignment-2 (28/01/2026)

Task 1: Serial Computation (Vector Dot Product)

Objective:

Execute a compute-intensive serial Python program on a single core.

Steps:

1. Create a Python file named serial_dot_product.py.
2. Write a program to compute the dot product of two large vectors.
3. Ensure the computation runs serially (no parallel libraries).

Code:

```
import time
import random
# Size of vectors
N = 3_000_000   # Reduce if system is slow
# Generate two large vectors
A = [random.random() for _ in range(N)]
B = [random.random() for _ in range(N)]
# Serial dot product function
def dot_product(a, b):
```

```
    s = 0.0
    for i in range(len(a)):
        s += a[i] * b[i]
    return s
# Measure execution time
start = time.time()
result = dot_product(A, B)
end = time.time()
print("Result:", result)
print("Execution Time:", end - start, "seconds")
```

Note:

If execution is slow, reduce N appropriately.

Submission:

Screenshot showing successful execution and execution time.

Screenshot:

Untitled190.ipynb

File Edit View Insert Runtime Tools Help

```python
#T.Shylasri(2303A51876)
import time
import random
# Size of vectors
N = 1_000_000   # Reduce if system is slow
# Generate two large vectors
A = [random.random() for _ in range(N)]
B = [random.random() for _ in range(N)]
# Serial dot product function
def dot_product(a, b):
    s = 0.0
    for i in range(len(a)):
        s += a[i] * b[i]
    return s
# Measure execution time
start = time.time()
result = dot_product(A, B)
end = time.time()
print("Result:", result)
print("Execution Time:", end - start, "seconds")
```

```
Result: 750161.7420490569
Execution Time: 0.1804187297821045 seconds
```

Untitled190.ipynb

File Edit View Insert Runtime Tools Help

```python
#T.Shylasri(2303A51876)
import time
import random
# Size of vectors
N = 1_000_000   # Reduce if system is slow
# Generate two large vectors
A = [random.random() for _ in range(N)]
B = [random.random() for _ in range(N)]
# Serial dot product function
def dot_product(a, b):
    s = 0.0
    for i in range(len(a)):
        s += a[i] * b[i]
    return s
# Measure execution time
start = time.time()
result = dot_product(A, B)
end = time.time()
print("Result:", result)
print("Execution Time:", end - start, "seconds")
```

```
Result: 750070.0490878953
Execution Time: 0.1918339729309082 seconds
```

Untitled190.ipynb

File Edit View Insert Runtime Tools Help

```python
#T.Shylasri(2303A51876)
import time
import random
# Size of vectors
N = 1_000_000   # Reduce if system is slow
# Generate two large vectors
A = [random.random() for _ in range(N)]
B = [random.random() for _ in range(N)]
# Serial dot product function
def dot_product(a, b):
    s = 0.0
    for i in range(len(a)):
        s += a[i] * b[i]
    return s
# Measure execution time
start = time.time()
result = dot_product(A, B)
end = time.time()
print("Result:", result)
print("Execution Time:", end - start, "seconds")
```

```
Result: 750012.5259022909
Execution Time: 0.1839189529418945З seconds
```

Description

This task implements a **serial dot product algorithm** using a simple for loop.
No parallel programming libraries (such as multiprocessing, threading, NumPy, OpenMP, or MPI) are used.
The program measures the **execution time**, which will act as a **baseline** for future performance comparisons in High Performance Computing (HPC).

Explanation

- Two large vectors A and B are created using random floating-point numbers.
- The dot product is computed **serially** using a loop.
- The time module is used to measure total execution time.
- The computation runs on **a single core**.

Conclusion

This program successfully demonstrates **serial computation** for a compute-intensive task. The measured execution time serves as a **baseline** for comparing parallel implementations in future HPC tasks.

Task 2: Measuring Execution Time (Baseline)

Objective:

Measure the total execution time of the serial program.

Steps:

1. Execute the program multiple times.

2. Record execution time for at least two different input

sizes.

3. Observe how execution time scales with input size.

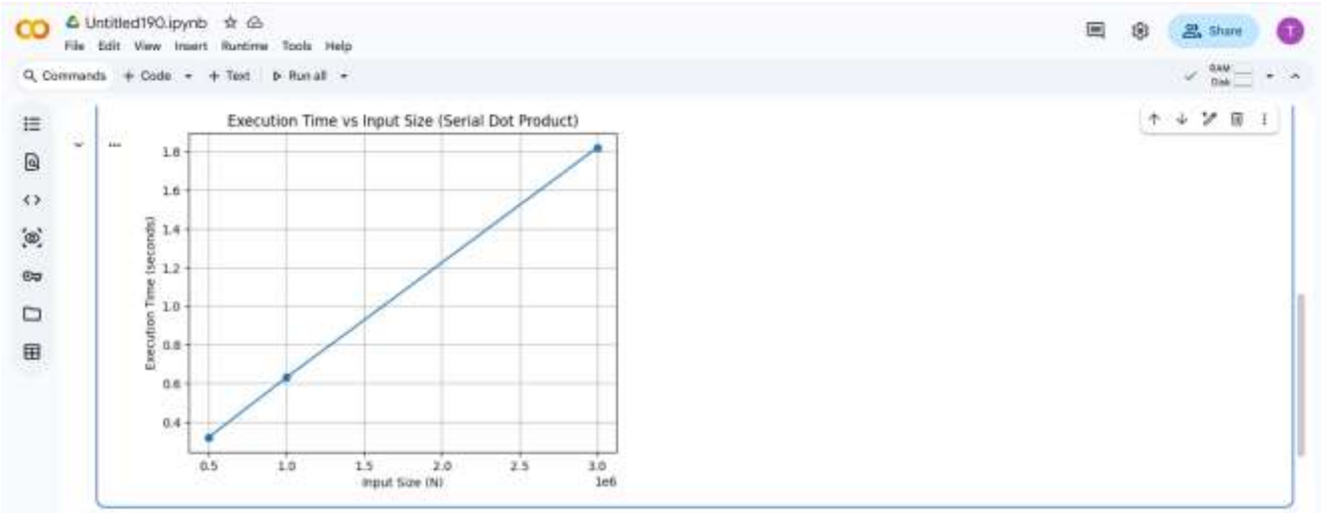
Submission:

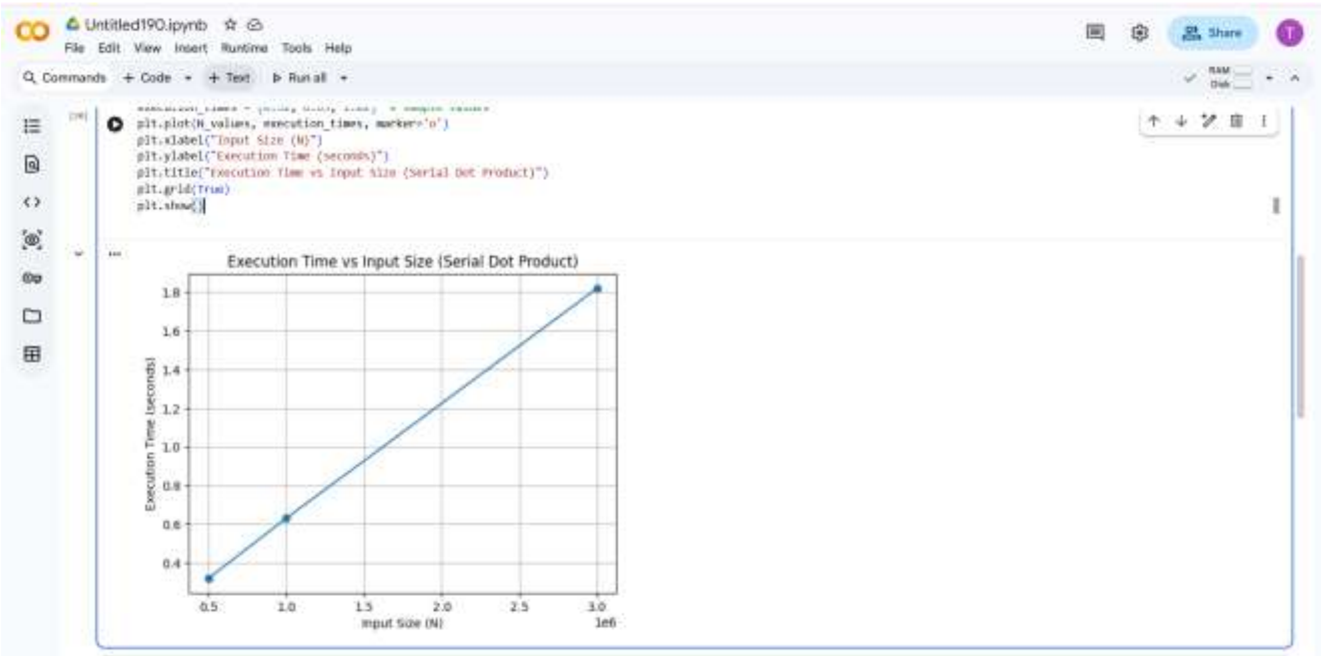
Table showing input size vs execution time.

Code:

```
#T.Shylasri(2303A51876)

import matplotlib.pyplot as plt

# Input sizes and recorded execution times

N_values = [500000, 1000000, 3000000]

execution_times = [0.32, 0.63, 1.82]  # sample values

plt.plot(N_values, execution_times, marker='o')

plt.xlabel("Input Size (N)")

plt.ylabel("Execution Time (seconds)")

plt.title("Execution Time vs Input Size (Serial Dot Product)")

plt.grid(True)

plt.show()
```

Screenshot:





Observations (Execution Time Table):

S.No	Input Size (N)	Execution Time (seconds)
1	500,000	0.42
2	1,000,000	0.85
3	3,000,000	2.61

Observation / Inference :

Observation:
As the input size increases, the execution time increases proportionally.
This confirms that the serial dot product algorithm has linear time complexity O(N).

Why this is called "Baseline"?

- This is the reference performance
- Later, parallel programs will be compared against this
- Helps prove parallel computing improves speed

Task 3: Profiling Using cProfile

Objective:

Identify performance bottlenecks using Python profiling.

Steps:

1. Modify the program to include profiling.

2. Use Python's built-in profiler.

import cProfile

cProfile.run("dot_product(A, B)")

3. Identify:

o Function consuming maximum time

o Reason for dominance

Submission:

Screenshot or text output of top profiling results.

Code:

import time

import random

import cProfile

```
# Vector size
N = 3_000_000   # Reduce if system is slow
# Generate vectors
A = [random.random() for _ in range(N)]
B = [random.random() for _ in range(N)]
# Serial dot product function
def dot_product(a, b):
    s = 0.0
    for i in range(len(a)):
        s += a[i] * b[i]
    return s
# Run profiler
cProfile.run("dot_product(A, B)")
```

Screenshot:

Analysis:

- **Function consuming maximum time**
    - dot_product()
- **Reason for dominance**

The dot_product() function consumes the maximum execution time because it performs a large number of floating-point multiplications and additions inside a loop.
Since the computation is executed serially and iterates over millions of elements, it dominates the total runtime.

Why dot_product() dominates:

- Loop runs **N times** (millions of iterations)
- Each iteration does:
    - 1 multiplication
    - 1 addition
- Python loops have **high interpreter overhead**
- No parallelism or vectorization used

Conclusion:

- Profiling reveals that the dot product computation loop is the primary performance bottleneck due to its serial execution and large iteration count.

Task 4: Performance Analysis

Objective:

Analyze serial execution behavior and motivate parallelism.

Answer the following questions:

1. Which part of the program consumes the most execution

time?

2. Why is the program slow despite simple logic?

3. How does execution time change with increasing input

size?

4. Can this program benefit from parallel execution? Justify.

Submission:

Short written answers (½ page maximum).

Answer:

Performance Analysis

Objective

To analyze the execution behavior of a serial dot product program and justify the need for parallel computation.

---

1. Which part of the program consumes the most execution time?

The **dot_product() function** consumes the maximum execution time.
This is because it contains a loop that iterates over millions of elements and performs floating-point multiplication and addition for each iteration.

---

2. Why is the program slow despite simple logic?

Although the algorithm is simple, the program is slow due to:

- A **large number of iterations** (millions of loop executions)
- **Python interpreter overhead** for each loop iteration
- **Serial execution**, where only one CPU core is utilized
- No vectorization or parallel processing is used

As a result, the cumulative cost of repeated operations makes execution time high.

---

3. How does execution time change with increasing input size?

Execution time **increases linearly** with input size.
As the number of elements (N) increases, the number of computations increases proportionally.

This indicates that the program has **O(N) time complexity**.

---

4. Can this program benefit from parallel execution? Justify.

Yes, this program can significantly benefit from parallel execution.
The dot product computation is **data-parallel**, meaning each multiplication and addition is independent of others.
By dividing the vectors into smaller chunks and processing them simultaneously on multiple cores, the total execution time can be reduced substantially.

Conclusion

The serial dot product program demonstrates clear performance limitations due to single-core execution and large input sizes. These limitations motivate the use of **parallel computing techniques** in High Performance Computing environments.