**Class:** Final Year (Computer Science and Engineering)

**Year:** 2024-25 **Semester:** 1

**Course:** High Performance Computing Lab

**Practical No. 5**

**Exam Seat No: 22510047**

**Name: Aadarsh Nandedkar**

**Batch: B-8**

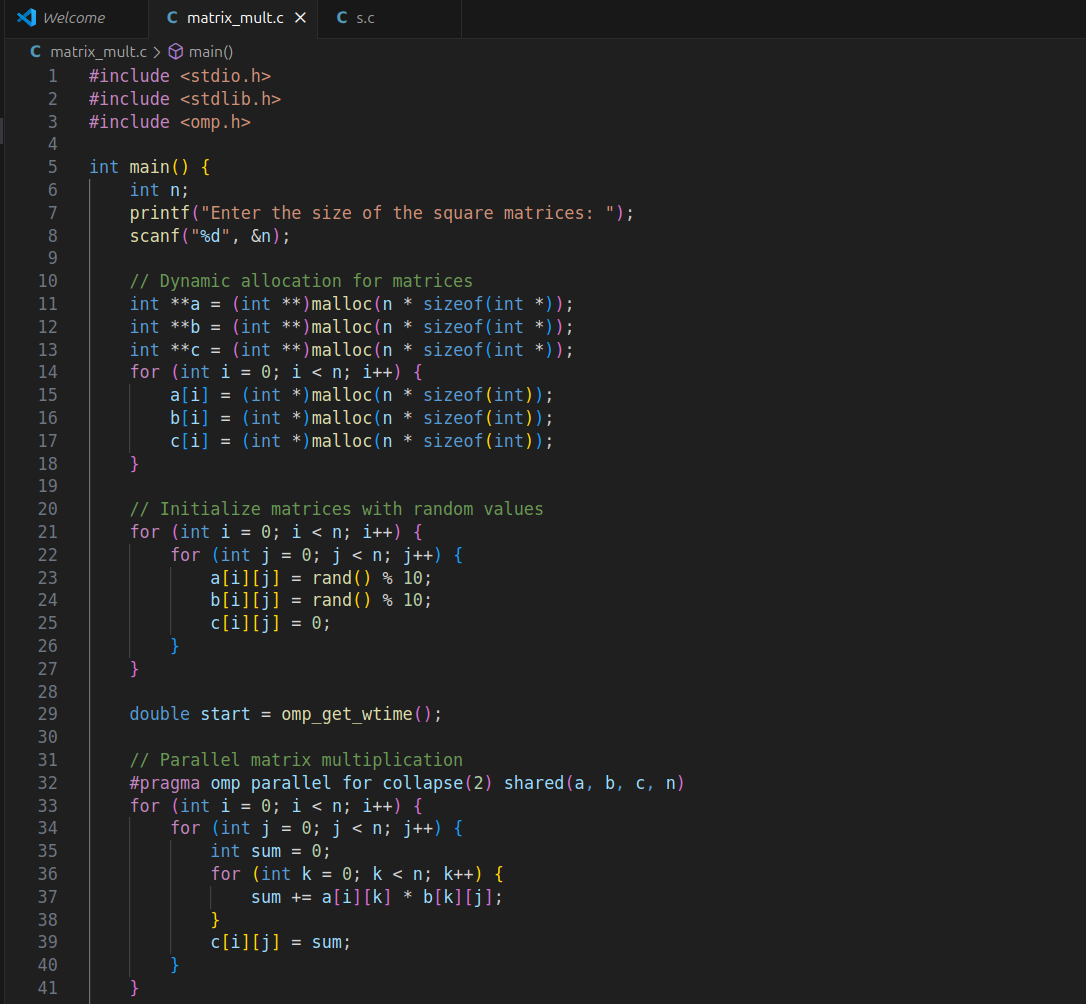
**Title of practical: Implementation of OpenMP programs.**

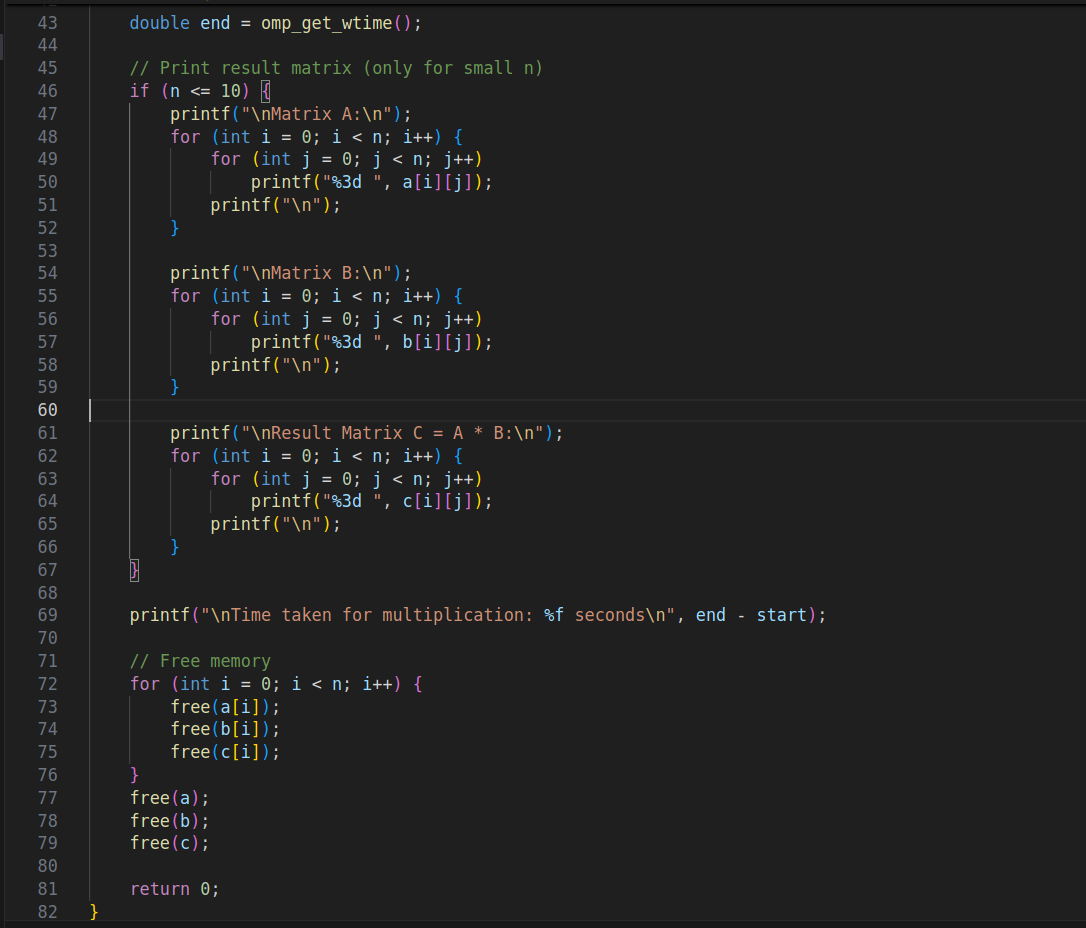
Implement following Programs using OpenMP with C:

1. Implementation of Matrix-Matrix Multiplication.
2. Implementation of Matrix-scalar Multiplication.
3. Implementation of Matrix-Vector Multiplication.
4. Implementation of Prefix sum.

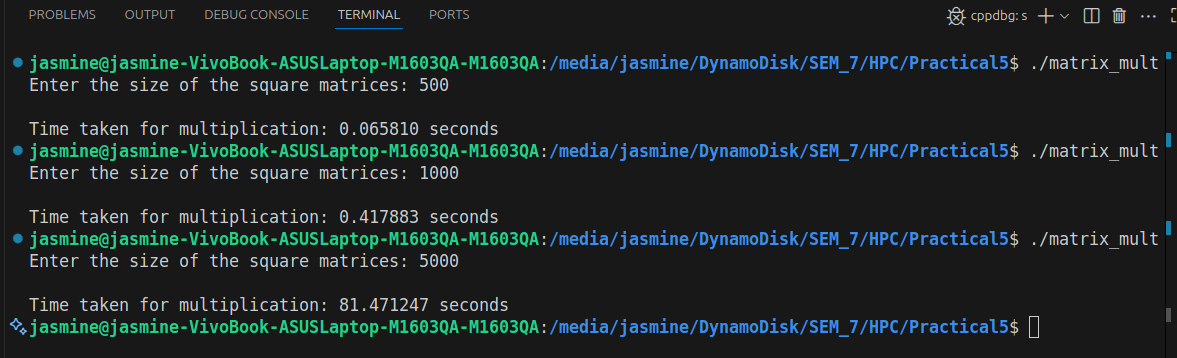
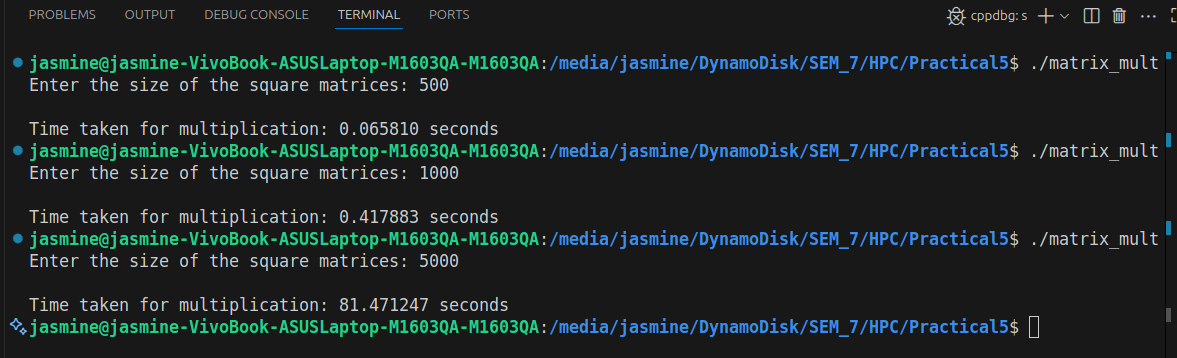
**Problem Statement 1:** Implementation of Matrix-Matrix Multiplication.

**Screenshots:**

****

****

**Output:**

****

**Information:**

* **collapse(2)**

Normally, OpenMP can only parallelize the outermost loop directly.  
But here, we have two nested loops (i and j)

So instead of parallelizing only over i, OpenMP parallelizes over both (i, j) pairs.

With collapse(2): both i and j are parallelized → threads get chunks of cells of the matrix, leading to better load balancing.

* **shared(a, b, c, n)**

Specifies which variables are shared among all threads.  
Here:  
a, b, c → matrices, must be visible to all threads.  
n → matrix size, same for all threads.  
important because OpenMP by default makes loop indices (i, j, k) private, but arrays must stay shared.

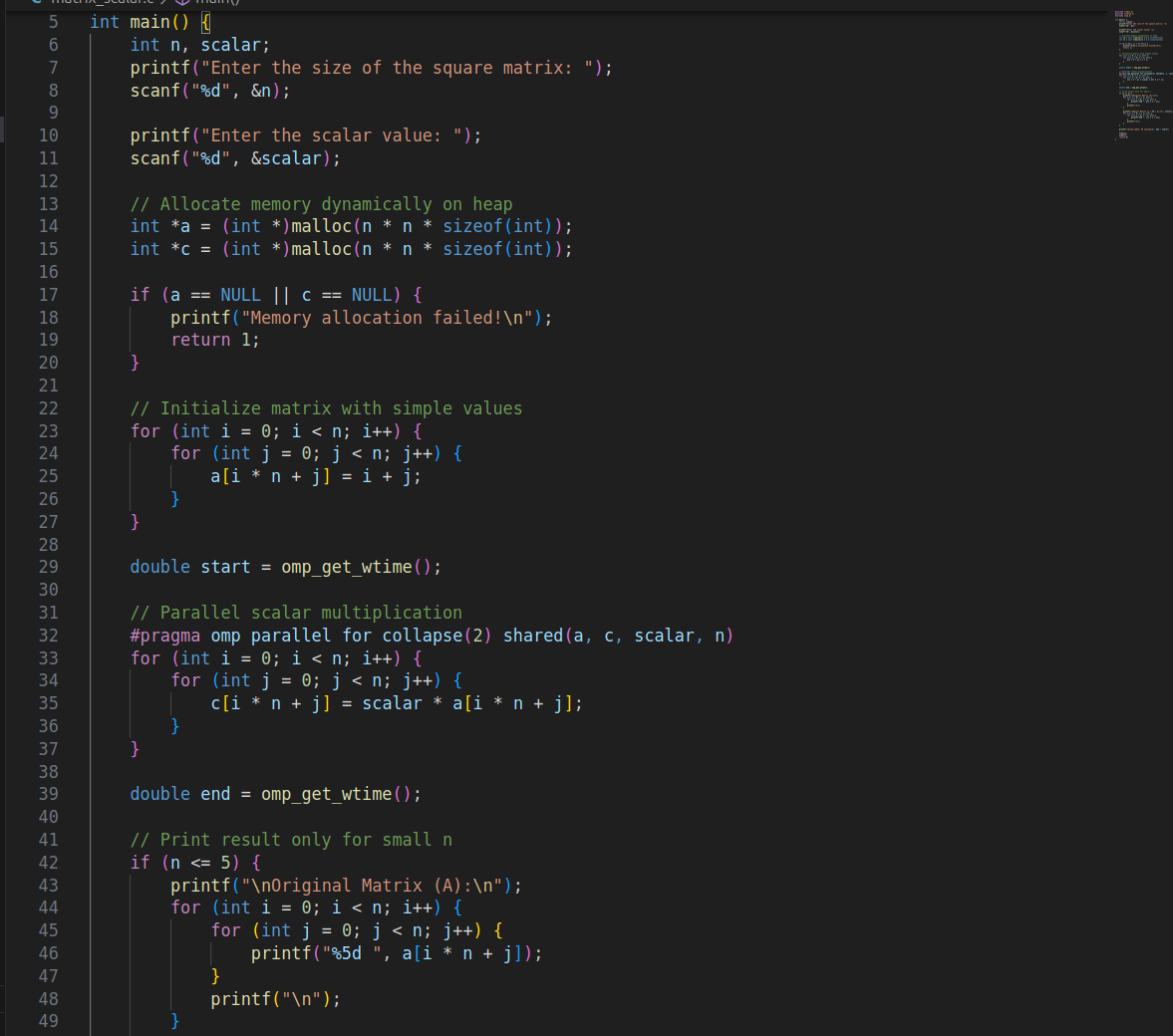
**Analysis:**

For small n (e.g., 3, 5, 10):  
Sequential is faster → OpenMP overhead (thread creation/synchronization) dominates.

For large n (e.g., 100, 500, 1000):  
OpenMP is faster → workload is heavy enough to benefit from parallelism.

**Problem Statement 2:** Implementation of Matrix-scalar Multiplication.

**Screenshots:**

****

****

**Information:**

* **collapse(2)**

Normally, OpenMP can only parallelize the outermost loop directly.  
But here, we have two nested loops (i and j)

So instead of parallelizing only over i, OpenMP parallelizes over both (i, j) pairs.

With collapse(2): both i and j are parallelized → threads get chunks of cells of the matrix, leading to better load balancing.

* **shared(a, b, c, n)**

Specifies which variables are shared among all threads.  
Here:  
a, b, c → matrices, must be visible to all threads.  
n → matrix size, same for all threads.  
important because OpenMP by default makes loop indices (i, j, k) private, but arrays must stay shared.

**Analysis:**

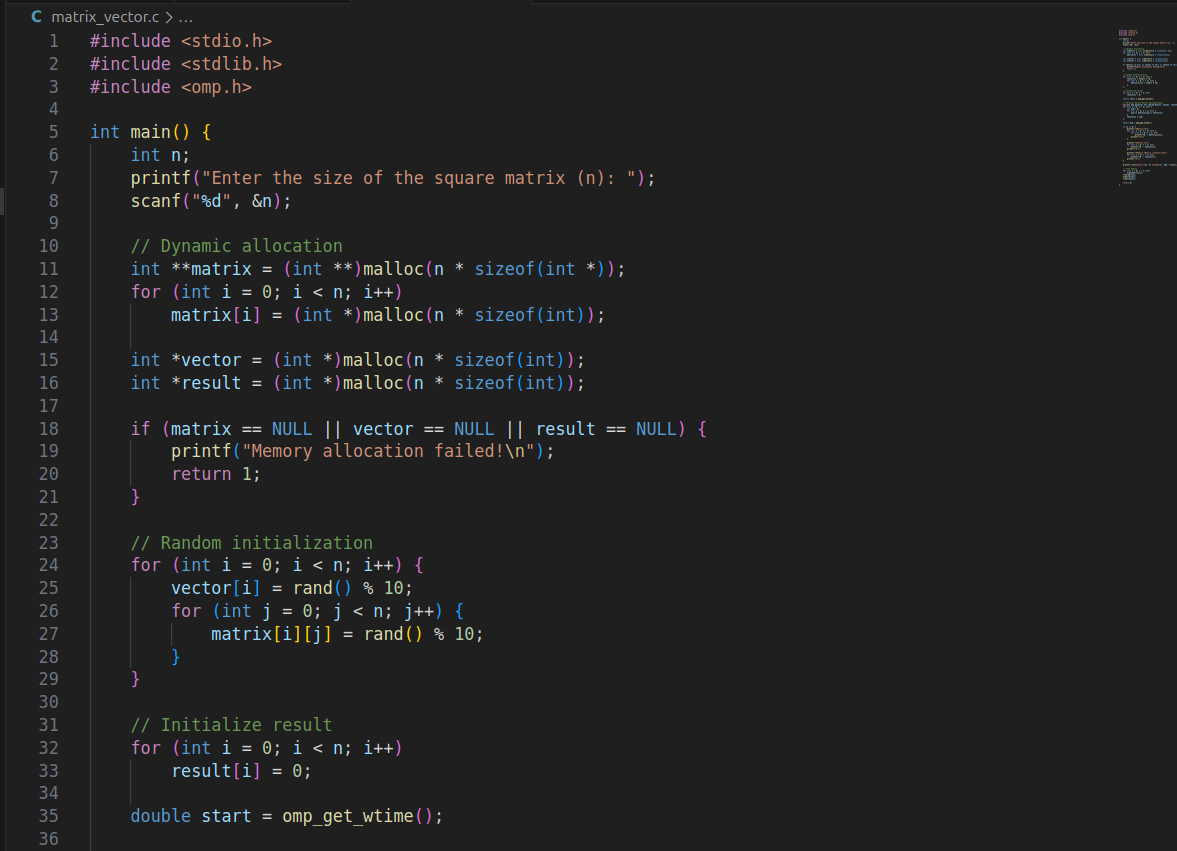
For small n, sequential execution can be faster (OpenMP overhead).

For large n (like 5000), OpenMP shows clear benefits due to parallel workload.

No inter-thread dependency (each element is independent), so parallel efficiency is high.

**Problem Statement 3:** Implementation of Matrix-Vector Multiplication.

**Screenshots:**

****

****

****

****

****

**Information:**

Multiplies an n × n matrix with an n × 1 vector.

Both matrix and vector are randomly initialized.

The result is stored in an n × 1 vector.

Execution time measured with omp\_get\_wtime().

Time complexity: O(n²).

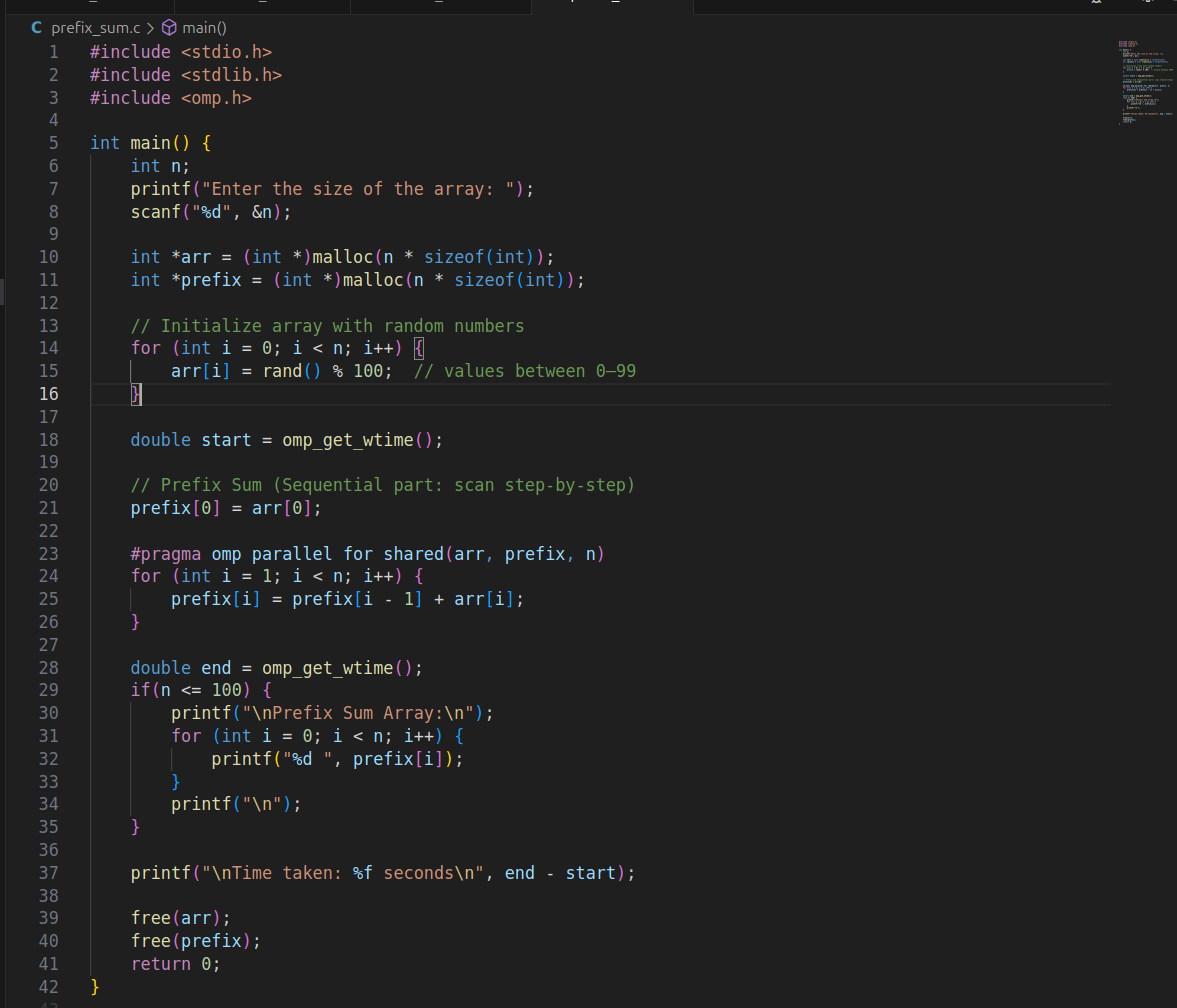
**Analysis:**

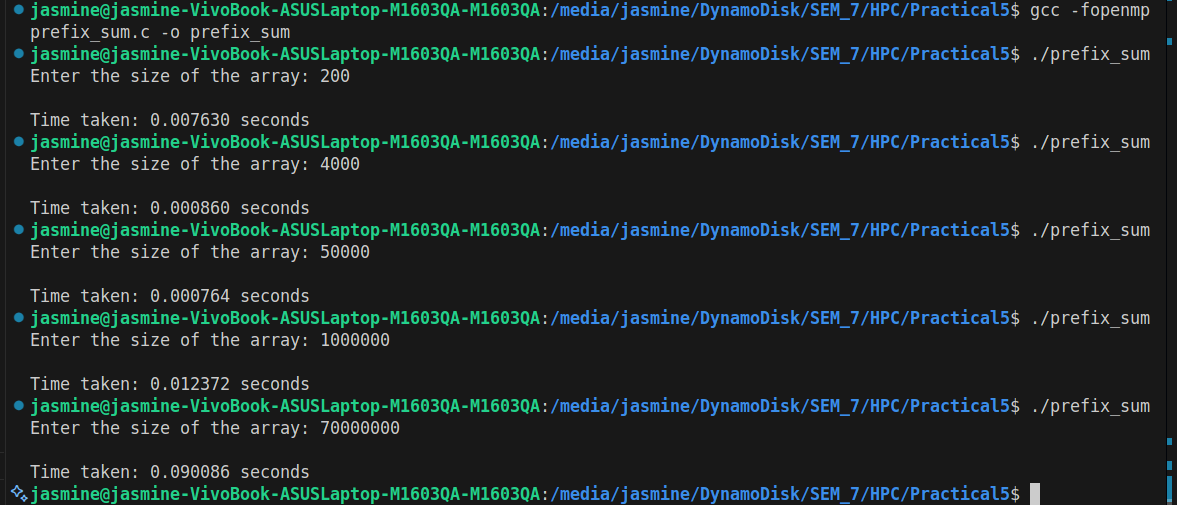
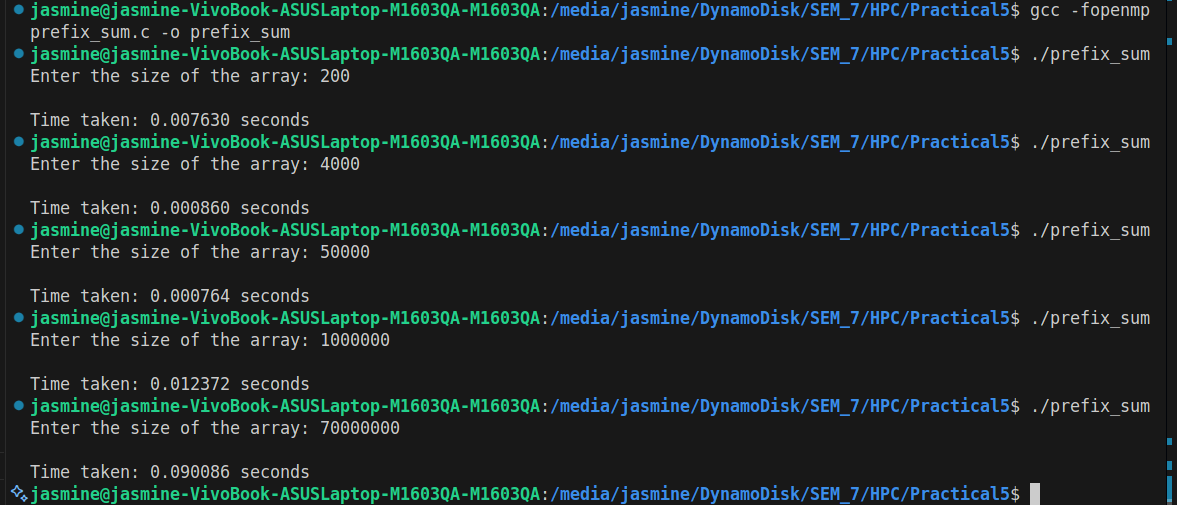
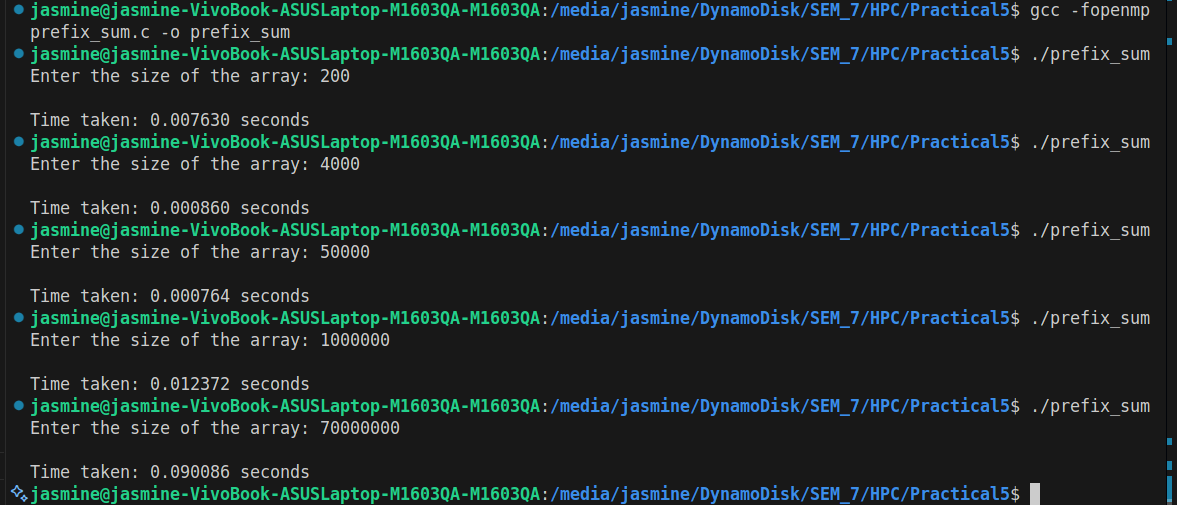
Each row’s dot product is independent → ideal for parallelism.

Parallel matrix-vector multiplication is beneficial for large sizes, but not for very small inputs.

**Problem Statement 4:** Implementation of Prefix sum.

**Screenshots:**

****

****

**Information:**

Prefix sum (also called scan) computes the cumulative sum of elements in an array.

Example: For input [a0, a1, a2, a3], prefix sum = [a0, a0+a1, a0+a1+a2, a0+a1+a2+a3].

**Analysis:**

For small n → sequential is faster.

For large n (like 1e7 elements) → parallel scan algorithms can achieve significant speedup.

This example demonstrates OpenMP usage, but real performance gain comes only with advanced prefix sum algorithms.

**Github Link:**

<https://github.com/av-nandedkar/HPC>