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

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

**Course:** High Performance Computing Lab

**Practical No. 5**

**Exam Seat No:**

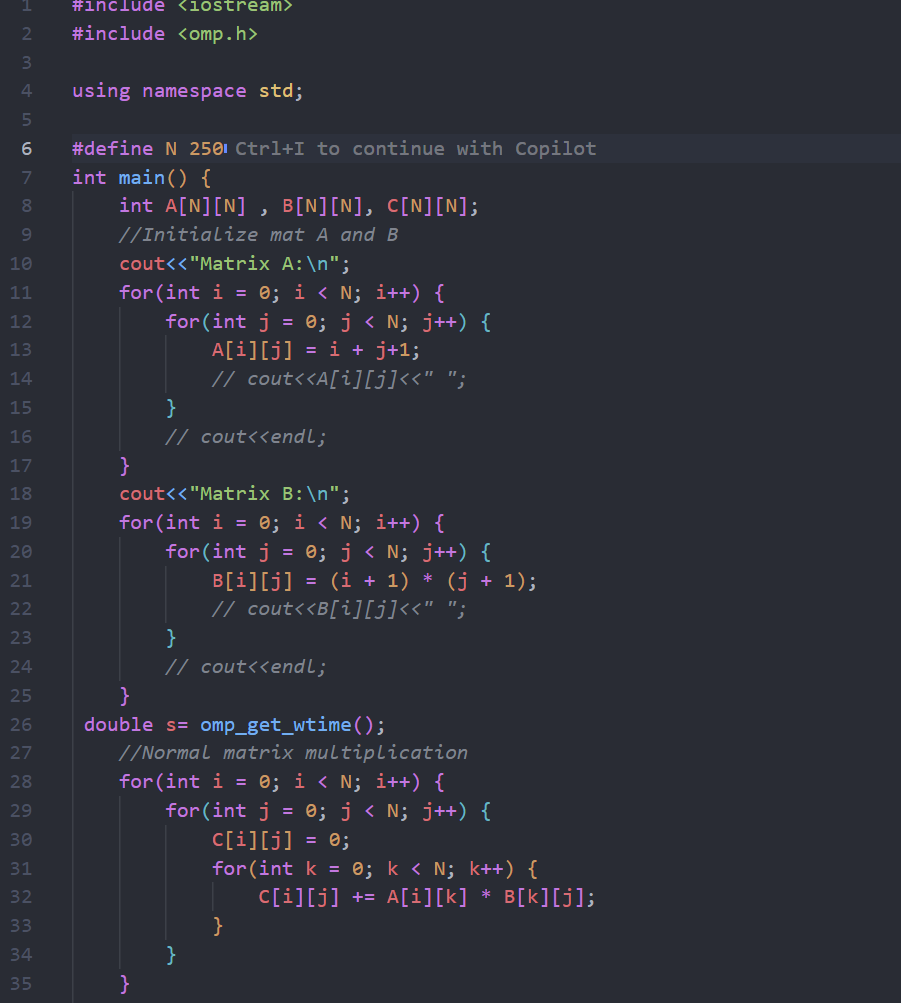
**Title of practical: Implementation of OpenMP programs.**

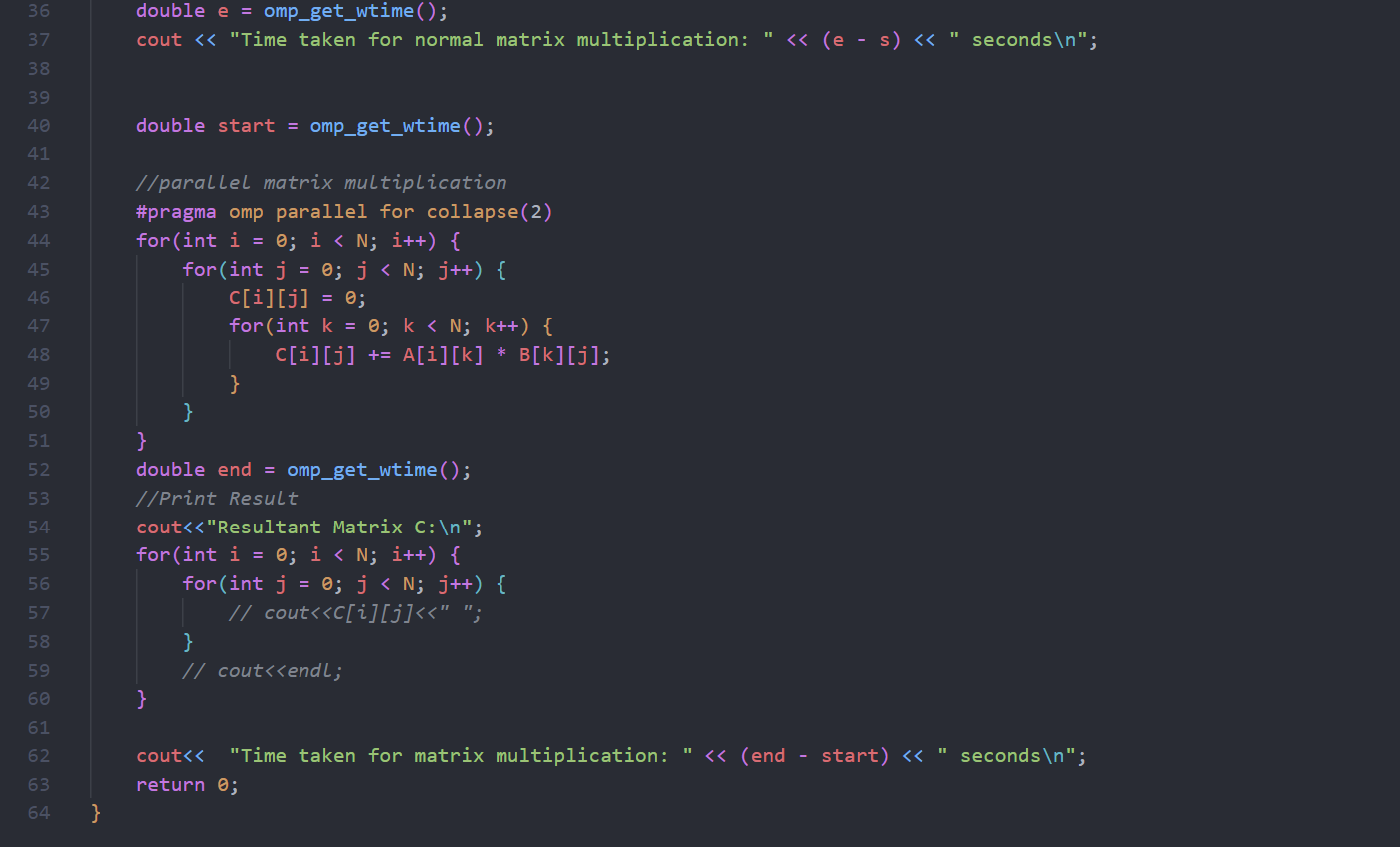
Implement following Programs using OpenMP with C:

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

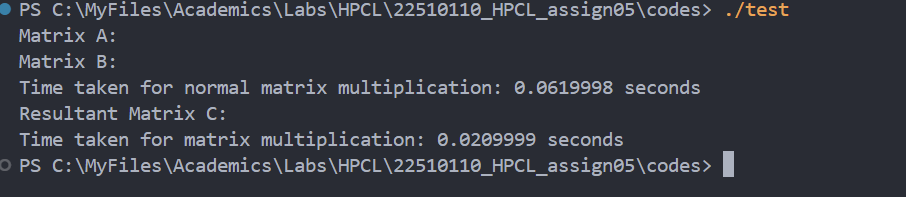
**Problem Statement 1:**

**Code:**





**Output:**

****

**Information:**

* The program multiplies two **N x N matrices** using **OpenMP** for parallelism.
* **#pragma omp parallel for collapse(2)** is used:
  + parallel for distributes iterations of loops among threads.
  + collapse(2) merges the outer two loops (i and j) for better load balancing across threads.
* Each thread computes one or more elements of the result matrix independently, avoiding data races

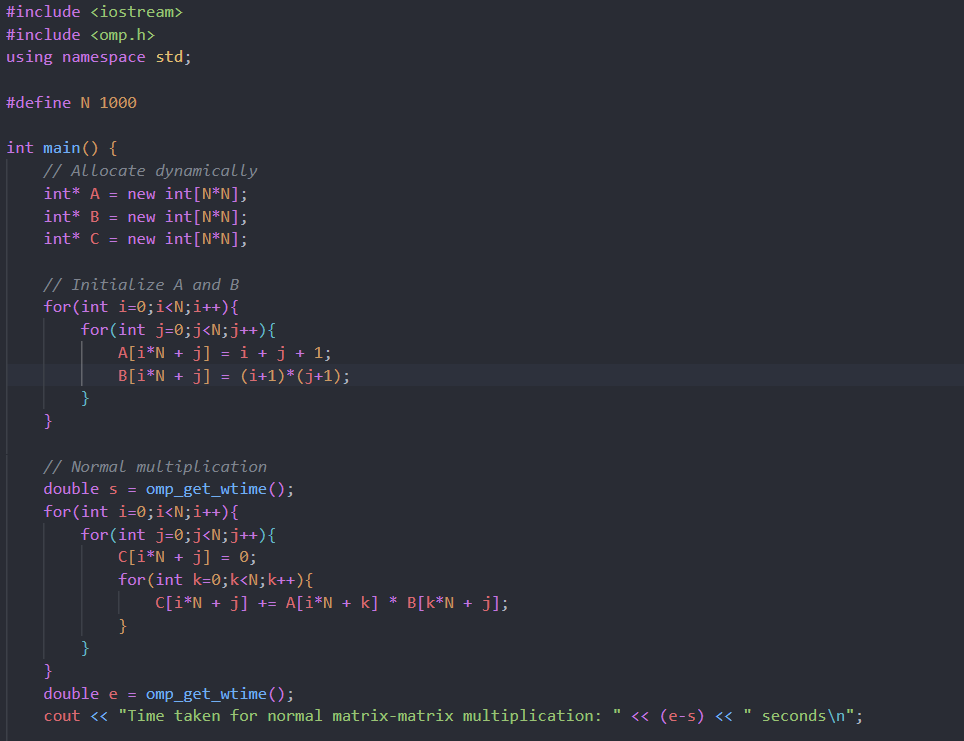
**Analysis:**

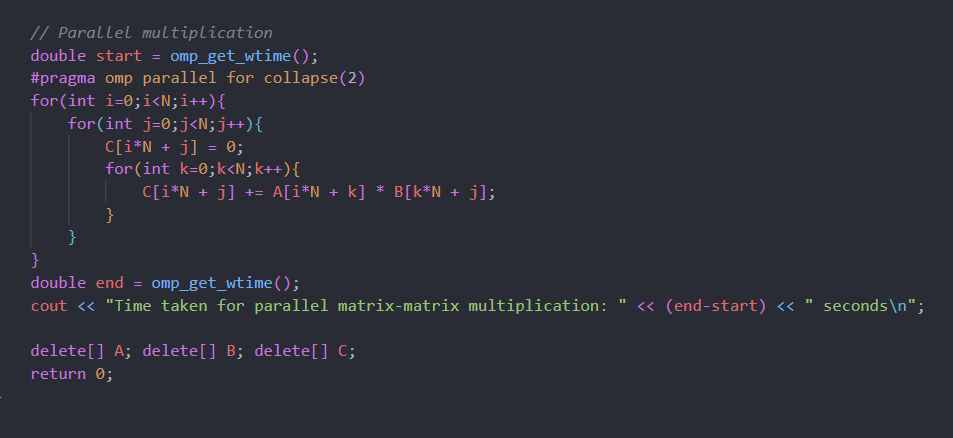
For the small size of the matrices the normal matrix multiplication was performing faster than the parallel one due to more overhead into the parallel one.

But for large size the parallel multiplication performs much faster than the normal one.

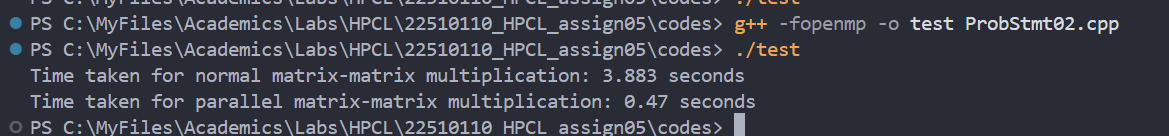
**Problem Statement 2:**

**Code:**

****

****

**Output:**

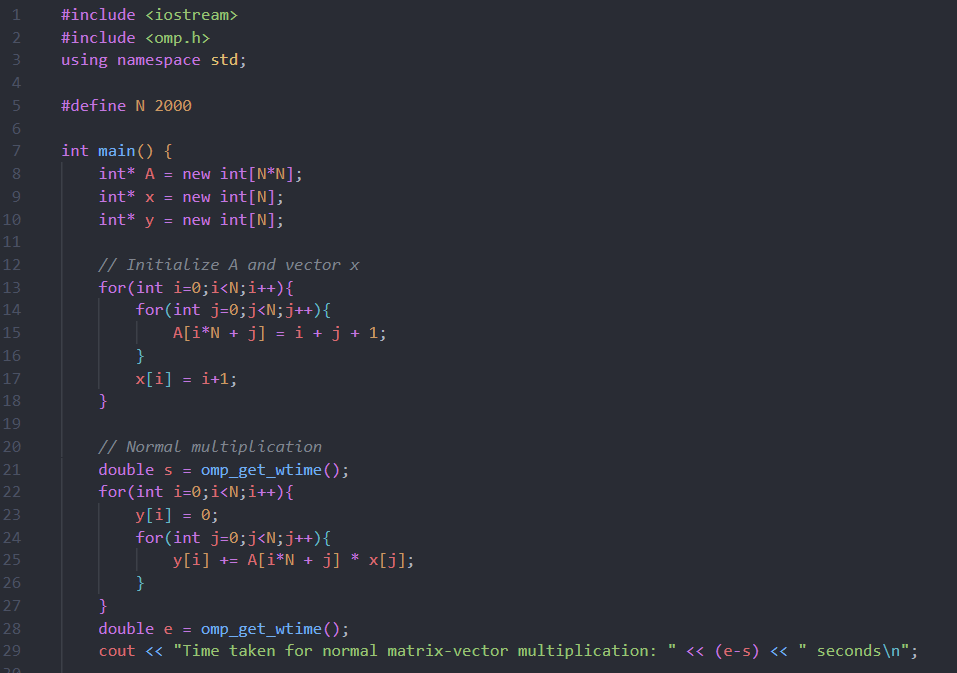
****

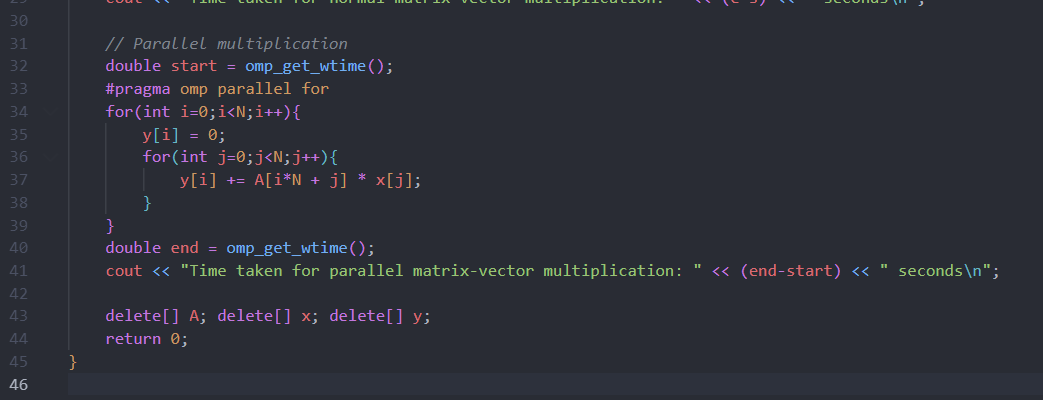
**Information:**

**Analysis:**

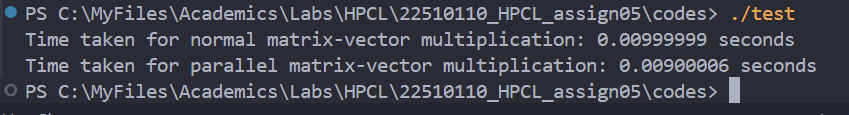
**Problem Statement 3:**

**Code:**

****

****

**Output:**

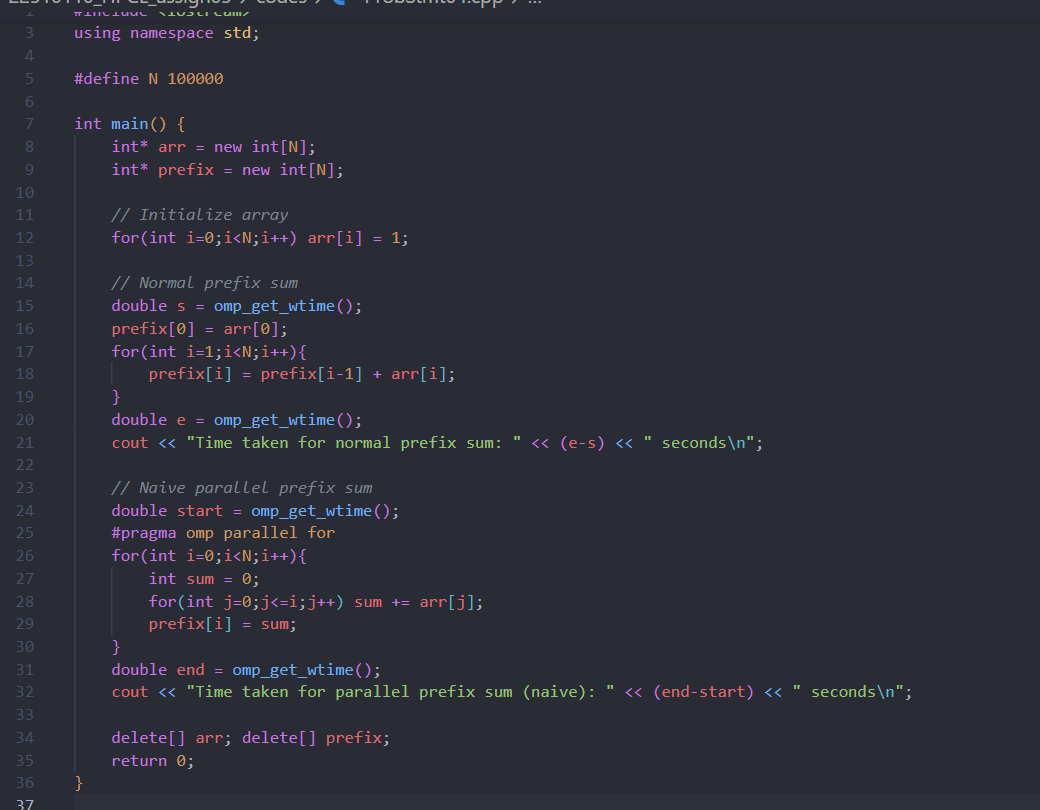
****

**Information:**

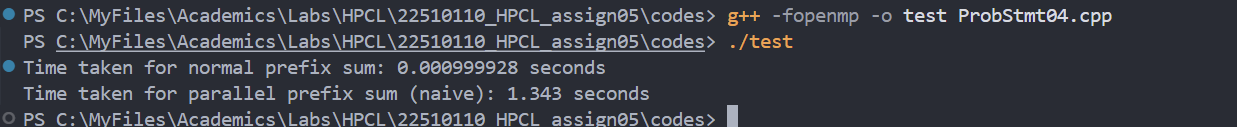
**Analysis:**

**Problem Statement 4:**

**Code:**

****

**Output:**

****

**Analysis:**

**Why parallel is slower for small N (1000 or 10,000)**

1. **Overhead dominates**
   * Spawning threads, dividing work, and synchronizing them takes much more time than just doing 1000 additions in a single CPU core.
   * Example: creating/joining threads may cost micro- to milliseconds, while 1000 additions are done in nanoseconds.
2. **Prefix sum dependency**
   * Prefix sums are inherently sequential.
   * A naive parallelization tries to assign chunks to threads, but each chunk still depends on the result of the previous chunk → adds extra steps.
   * For small arrays, this dependency resolution overhead is bigger than the work itself.
3. **Cache locality**
   * Sequential prefix sum walks linearly through the array → cache-friendly.
   * Parallel prefix sum makes threads jump around memory and cause cache misses or false sharing, which slows things down.

**Github Link:**