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

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

**Course:** High Performance Computing Lab

**Practical No. 5**

**Exam Seat No:**

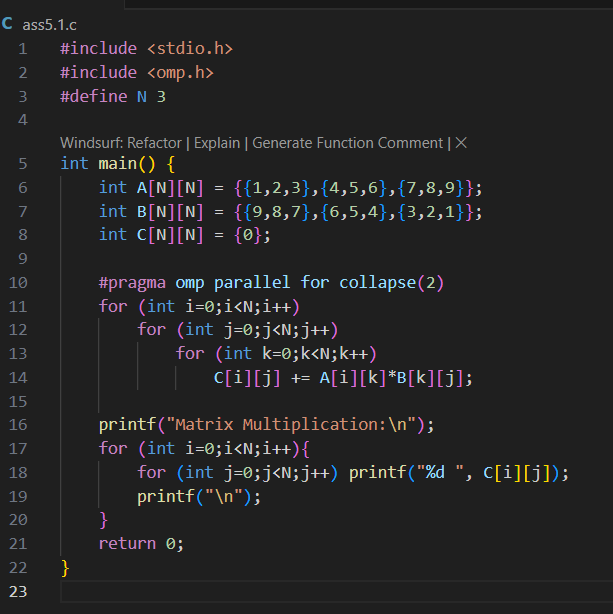
**Title of practical: Implementation of OpenMP programs.**

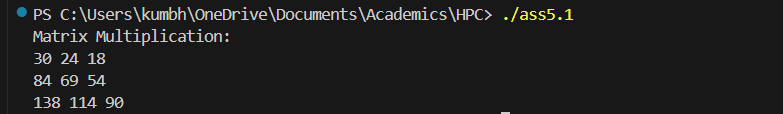
Implement following Programs using OpenMP with C:

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:**

**Screenshots:**

****

****

**Information:**

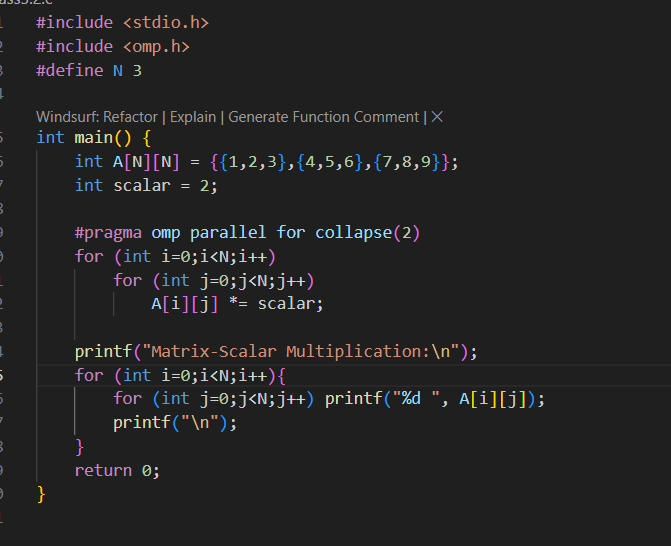
* **Uses #pragma omp parallel for collapse(2) to parallelize both the outer and middle loops, distributing the computation of each element of C across threads.**
* **Prints the resulting matrix.**
* **Normally: Only outer loop (i) is parallelized, inner loop (j) stays serial inside each thread.**
* **With collapse(2):**
  + **OpenMP flattens the 2 loops → total iterations = 3 \* 4 = 12.**
  + **Threads share these 12 iterations directly.**

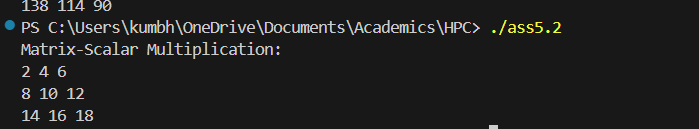
**Analysis:**

* Efficient for small matrices, but for larger matrices, memory access patterns and cache usage become important.
* The collapse(2) clause helps utilize more threads by flattening two loops, improving load balancing.

**Problem Statement 2:**

**Screenshots:**

****

****

**Information:**

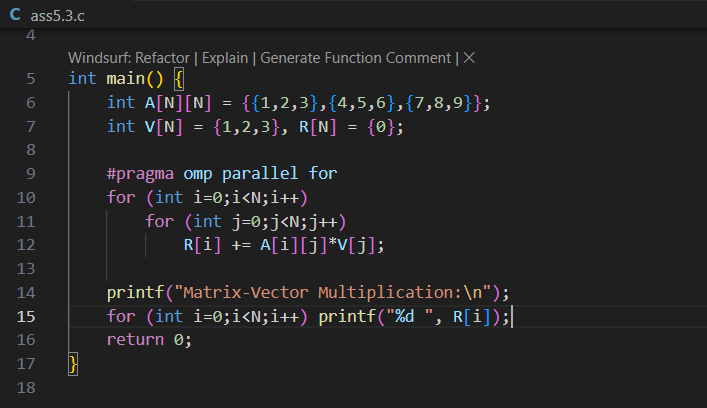
* **Uses #pragma omp parallel for collapse(2) to parallelize both loops, so each element multiplication can be done by a different thread.**
* **Prints the resulting matrix.**

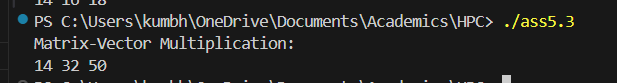
**Analysis:**

* **Demonstrates how element-wise operations on matrices can be easily parallelized.**
* **For very large matrices, this approach can significantly speed up computation.**

**Problem Statement 3:**

**Screenshots:**

****

****

**Information:**

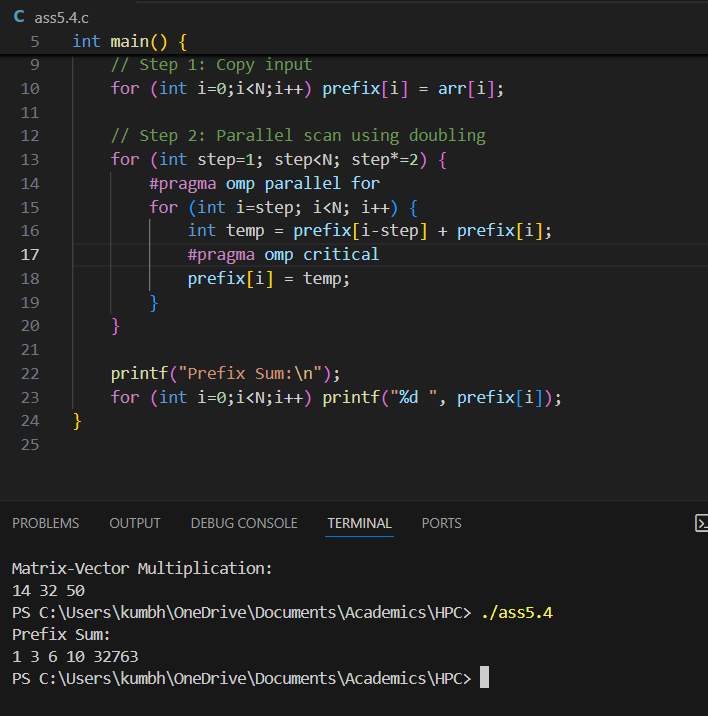
* **Uses #pragma omp parallel for to parallelize the computation of each element of the result vector.**
* **Each thread computes one row of the result.**

**Analysis:**

* **Matrix-vector multiplication is a common operation in scientific computing and is highly parallelizable.**
* **For larger matrices, this approach scales well with more threads.**

**Problem Statement 4:**

**Screenshots:**

****

* **Algorithm Explained**

The algorithm used is a

parallel scan (or prefix sum) using doubling. It works in

log₂(N) steps.

1. Step 1 (step=1): Each element adds its immediate left neighbor to its value.
2. Step 2 (step=2): Each element adds the value of the element two places to its left.
3. Step k (step = 2^k-1): Each element adds the value of the element 2^(k-1) places to its left.

process continues until the

step size is equal to or greater than the size of the array. At the end of the algorithm, each element

prefix[i] will hold the sum of all elements from arr[0] to arr[i]

**Information:**

* **Uses #pragma omp parallel for to parallelize the loop that computes the prefix sum.**
* Each outer loop (step) lets you add up numbers that are farther apart.
* First, you add each number to its immediate left neighbor.
* Next, you add each number to the sum two places to the left.
* Then, four places to the left, and so on.
* This way, each element accumulates the sum of all previous elements, building up the prefix sum in log₂(N) steps.

**Analysis:**

* **The code efficiently computes the prefix sum in parallel using a tree-based approach.**
* **For small arrays, the benefit is minor, but for large arrays, this approach can be much faster than a serial loop**

**Github Link: https://github.com/Anjali1874/HPC-Lab**