**Class:** Final Year B.Tech(Computer Science and Engineering)

**Year:** 2025-26 **Semester:** 1

**Course:** High Performance Computing Lab

**Practical No. 3**

**Exam Seat No: 22510046**

**Title of practical:**

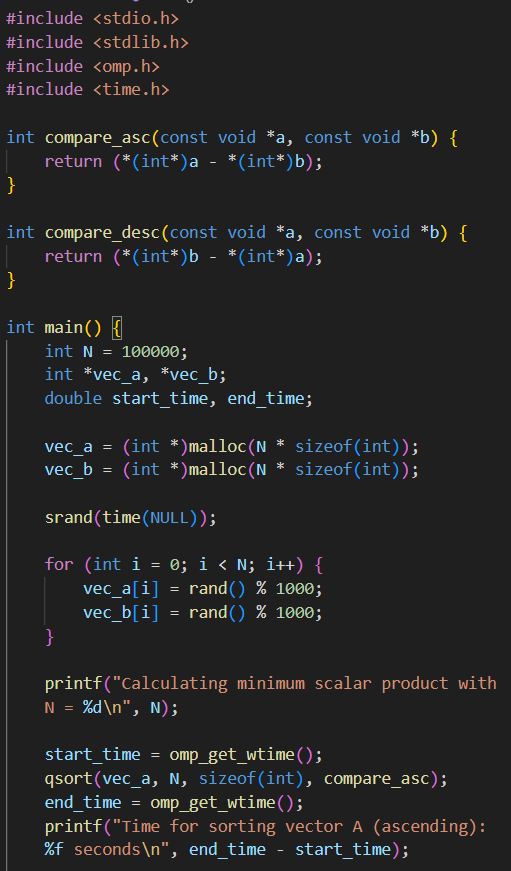
Study and Implementation of schedule, nowait, reduction, ordered and collapse clauses

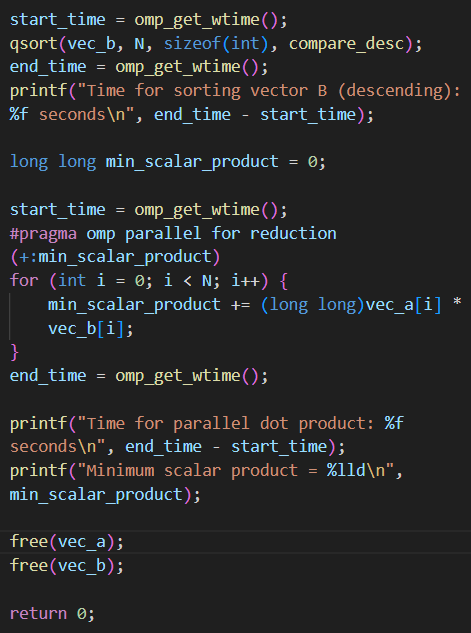
**Problem Statement 1:**

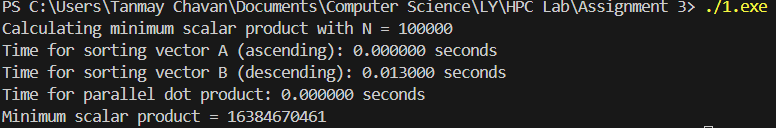
Analyse and implement a Parallel code for below program using OpenMP.

// C Program to find the minimum scalar product of two vectors (dot product)

**Screenshots:**

****

****

****

**Information and analysis:**

The key component of this code is the line #pragma omp parallel for reduction(+:min\_scalar\_product).

* #pragma omp parallel for: This creates a team of threads and divides the iterations of the following for loop among them. Each thread will work on a portion of the vector elements.
* reduction(+:min\_scalar\_product): This is the crucial part. It tells OpenMP that the variable min\_scalar\_product is a "reduction" variable with a + operation. This means:
  1. Each thread gets its own private copy of min\_scalar\_product, initialized to 0.
  2. The threads perform their calculations on their private copies. This avoids any race conditions.
  3. After the loop finishes, OpenMP automatically and safely adds all the private copies of min\_scalar\_product together and stores the final result in the original shared variable.

This approach ensures the correct result while still benefiting from parallel execution.

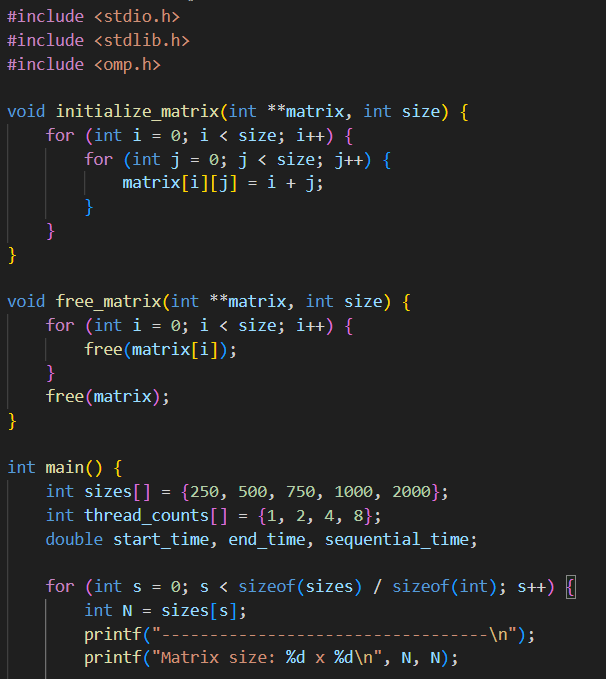
**Problem Statement 2:**

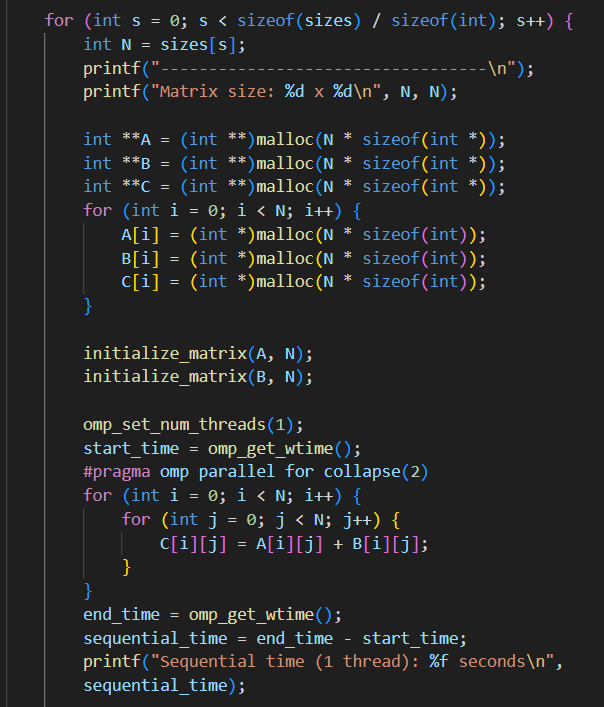
Write OpenMP code for two 2D Matrix addition, vary the size of your matrices from 250, 500, 750, 1000, and 2000 and measure the runtime with one thread (Use functions in C in calculate the execution time or use GPROF)

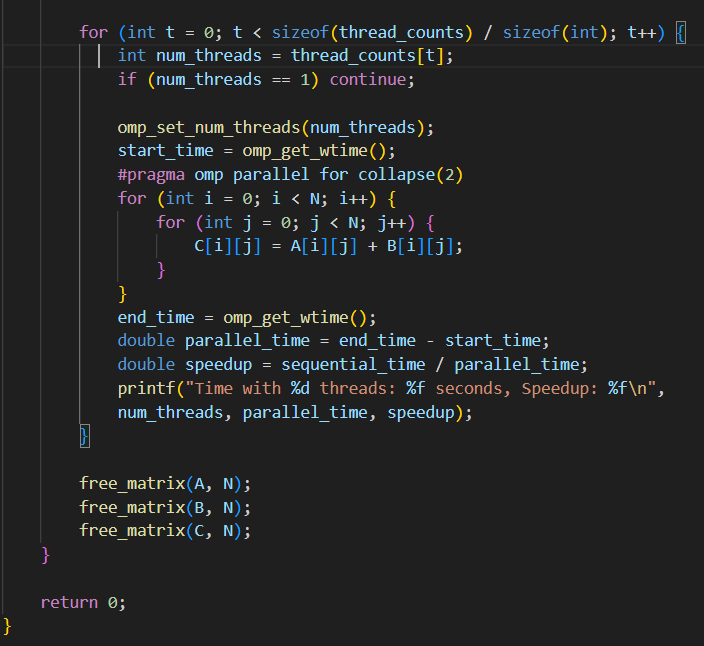
i. For each matrix size, change the number of threads from 2,4,8., and plot the speedup versus the number of threads.

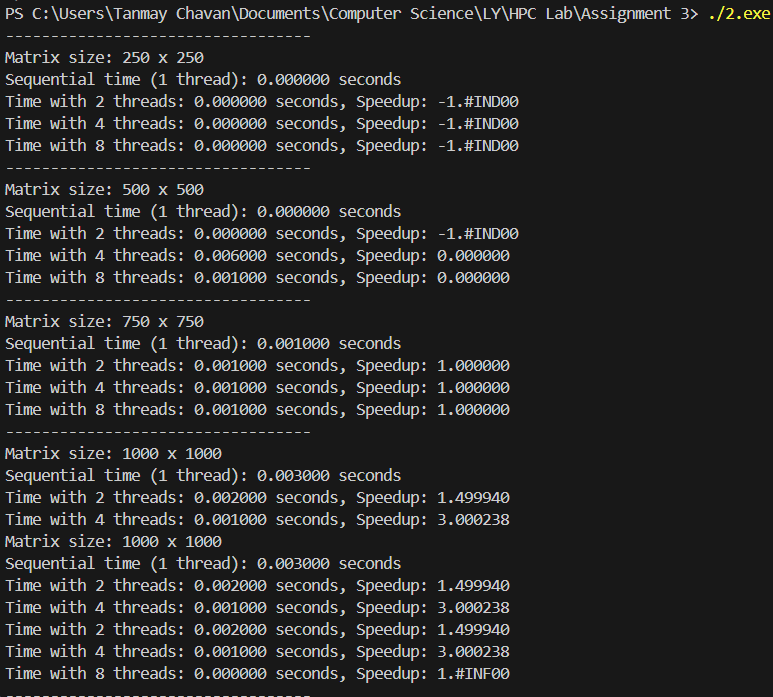
ii. Explain whether or not the scaling behaviour is as expected.

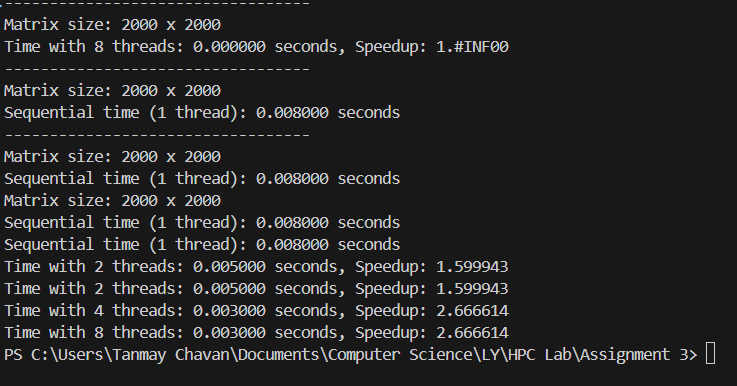
**Screenshots:**

****

****

****

****

****

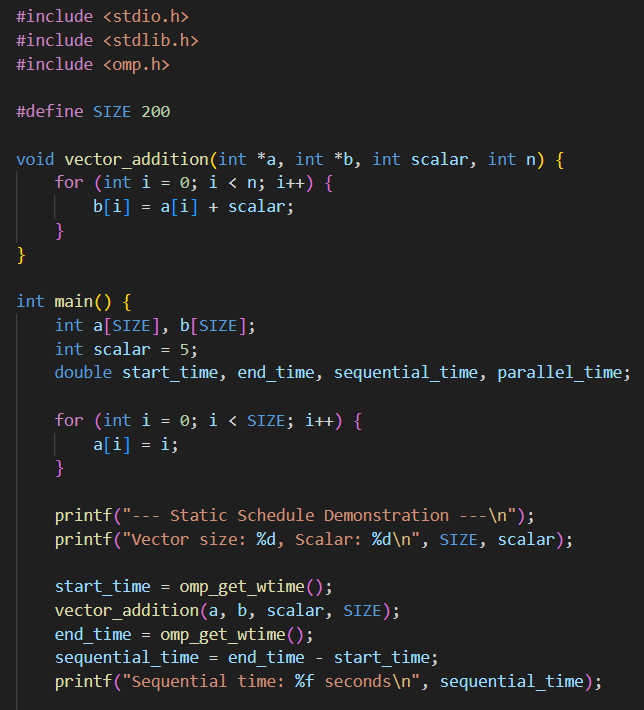
**Information and Analysis:**

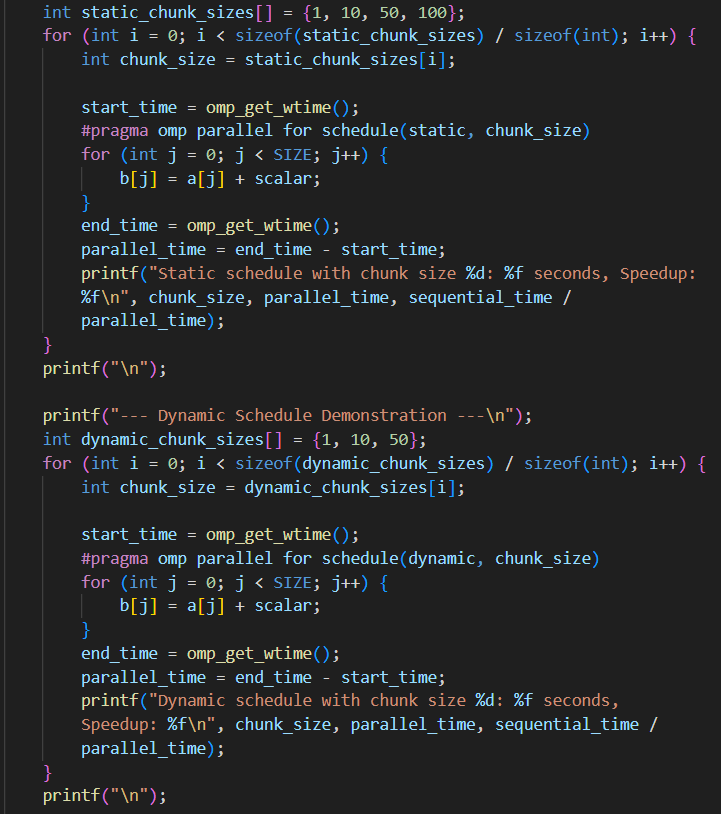
* The collapse clause on the #pragma omp parallel for directive tells OpenMP to parallelize both the i and j loops. Instead of just dividing the outer i loop iterations, it distributes all N×N iterations of the combined loop to the threads. This ensures a better workload balance, especially for smaller matrix sizes.

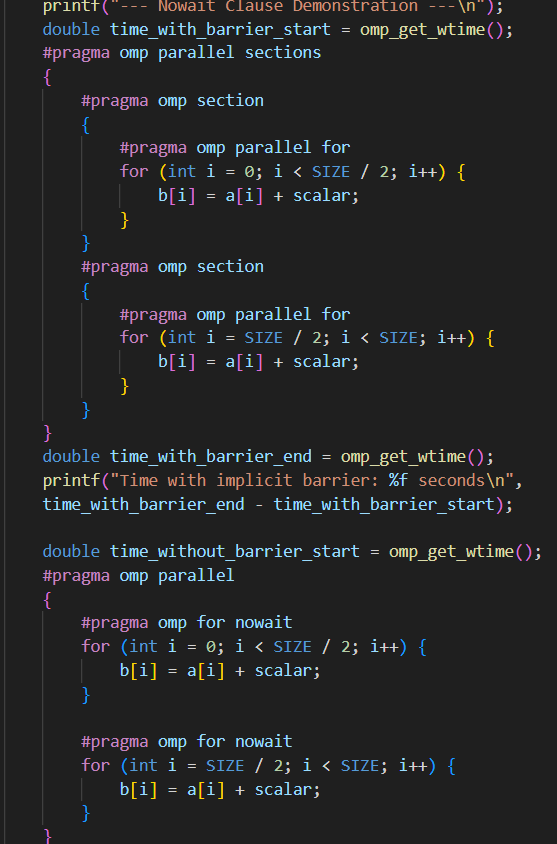
**Problem Statement 3:**

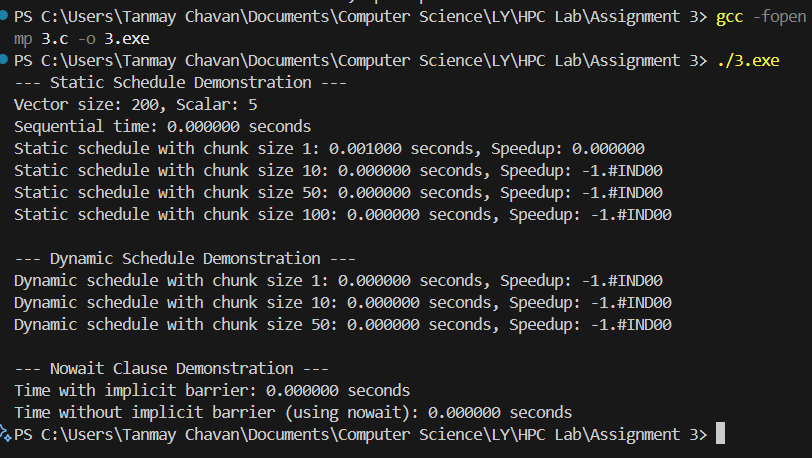
For 1D Vector (size=200) and scalar addition, Write a OpenMP code with the following: i. Use STATIC schedule and set the loop iteration chunk size to various sizes when changing the size of your matrix. Analyze the speedup. ii. Use DYNAMIC schedule and set the loop iteration chunk size to various sizes when changing the size of your matrix. Analyze the speedup. iii. Demonstrate the use of nowait clause.

**Screenshots:**

****

****

****

****

**Information and analysis:**

1. Static Schedule: The static schedule is ideal for loops where each iteration takes a roughly equal amount of time, as is the case in this simple vector addition. The workload is divided into pre-determined chunks and distributed to threads at the start.
2. Dynamic Schedule: The dynamic schedule is designed for loops with unbalanced workloads, where some iterations take longer than others. Threads are given a small chunk of work and, when finished, they dynamically request another chunk from a central queue.
3. Nowait Clause: OpenMP automatically inserts a barrier at the end of many parallel regions (like a for loop). This barrier ensures all threads have finished their work before the program moves on. nowait explicitly removes this barrier. This is useful when the code that follows does not depend on the results of the preceding parallel region.

**Github Link:**