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

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

**Course:** High Performance Computing Lab

**Practical No. 3**

**PRN: 22510047**

**Name: Aadarsh Venkatesh Nandedkar**

**Batch: B8**

**Title of practical:**

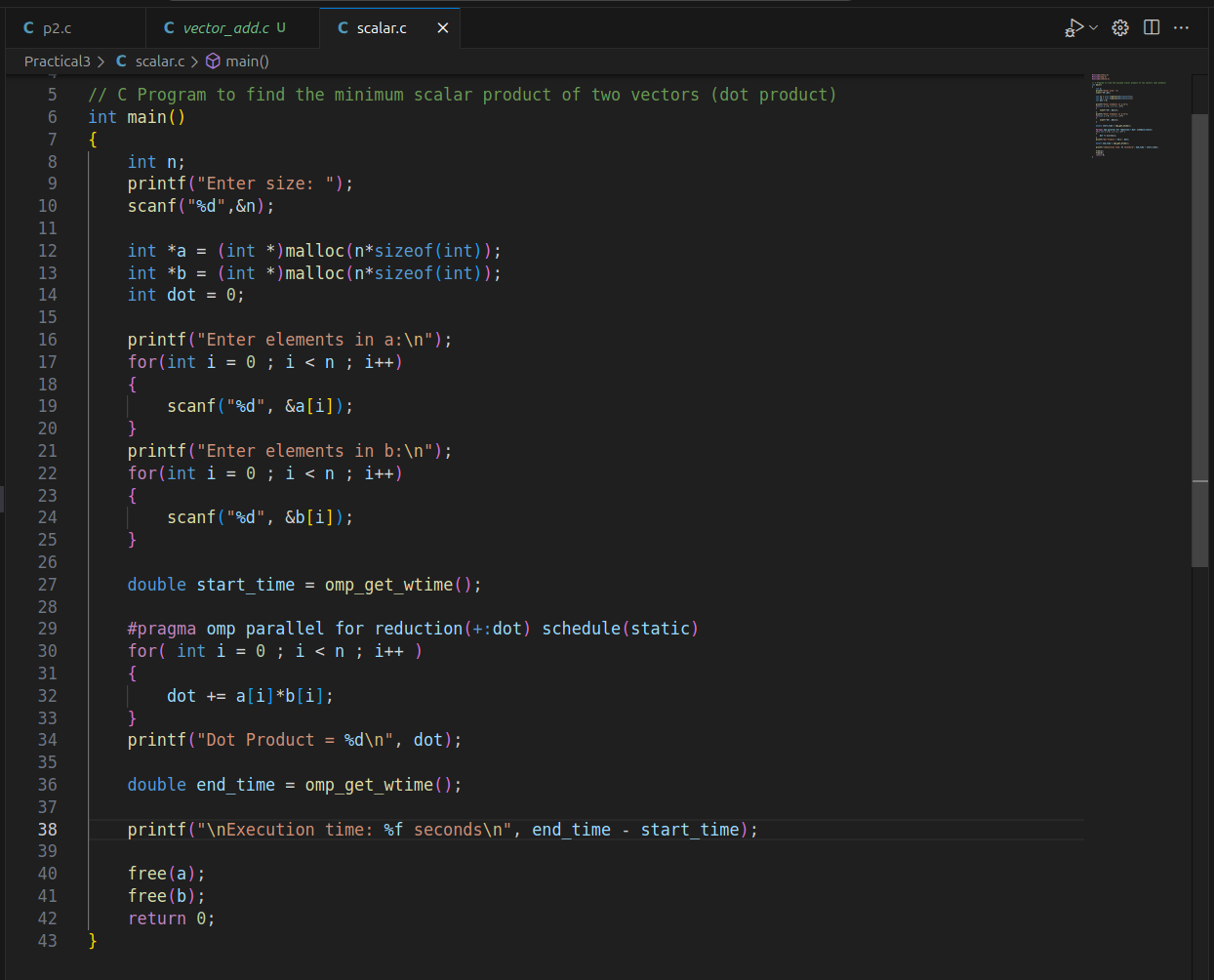
Study and Implementation of schedule, nowait, reduction, ordered and collapse clauses

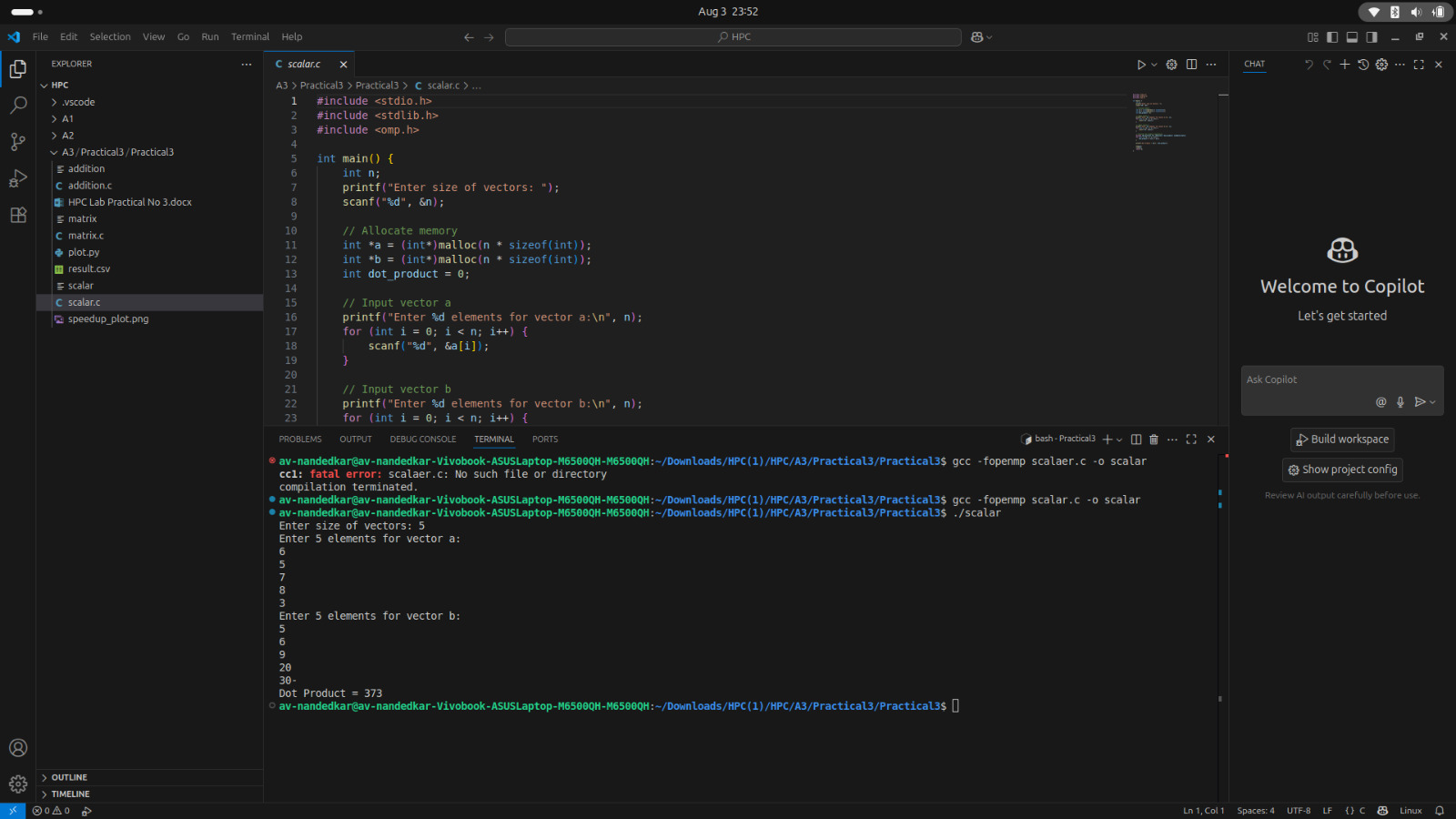
**Problem Statement 1:**

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:**

### schedule(type, chunk\_size)

Purpose: Controls how loop iterations are divided among threads.

Types:

static: Pre-divides equal chunks (good for balanced workloads).

dynamic: Assigns chunks on-the-fly (good for uneven workloads).

guided: Starts with large chunks, reduces over time.

### nowait

Purpose: Removes implicit barrier after the loop, allowing threads to proceed immediately.

Use Case: When next loop doesn’t depend on current results.

**reduction(op:var)**

Purpose: Combines private copies of var from all threads using op (e.g., +, \*, max). Avoids race conditions when accumulating results.

**ordered**

Purpose: Ensures a block executes in sequential order (rarely needed).

Use Case: When loop iterations must be processed in order.

### collapse(n)

Purpose: Parallelizes nested loops by flattening them into a single loop.

Use Case: For nested loops with no dependencies.

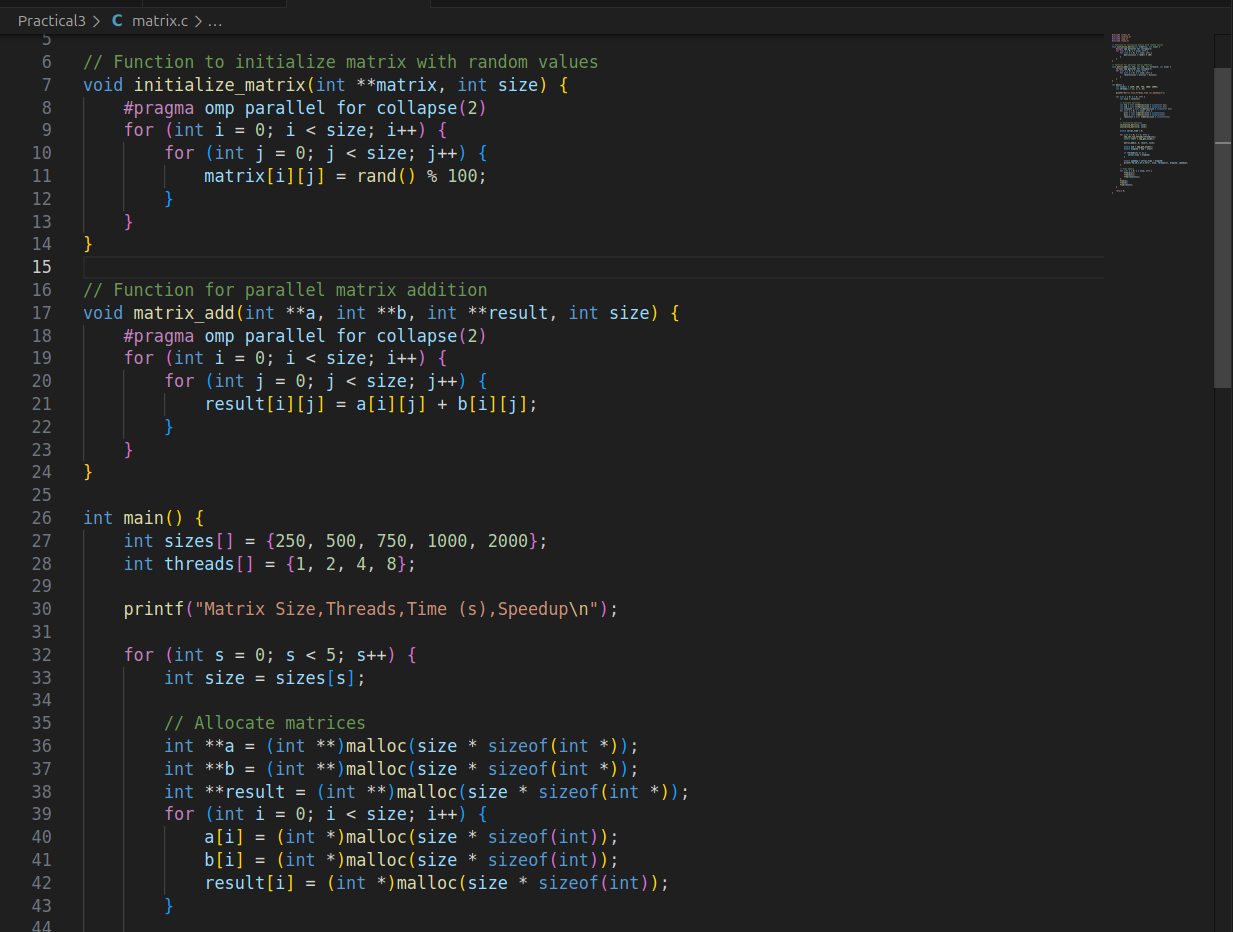
**Problem Statement 2:**

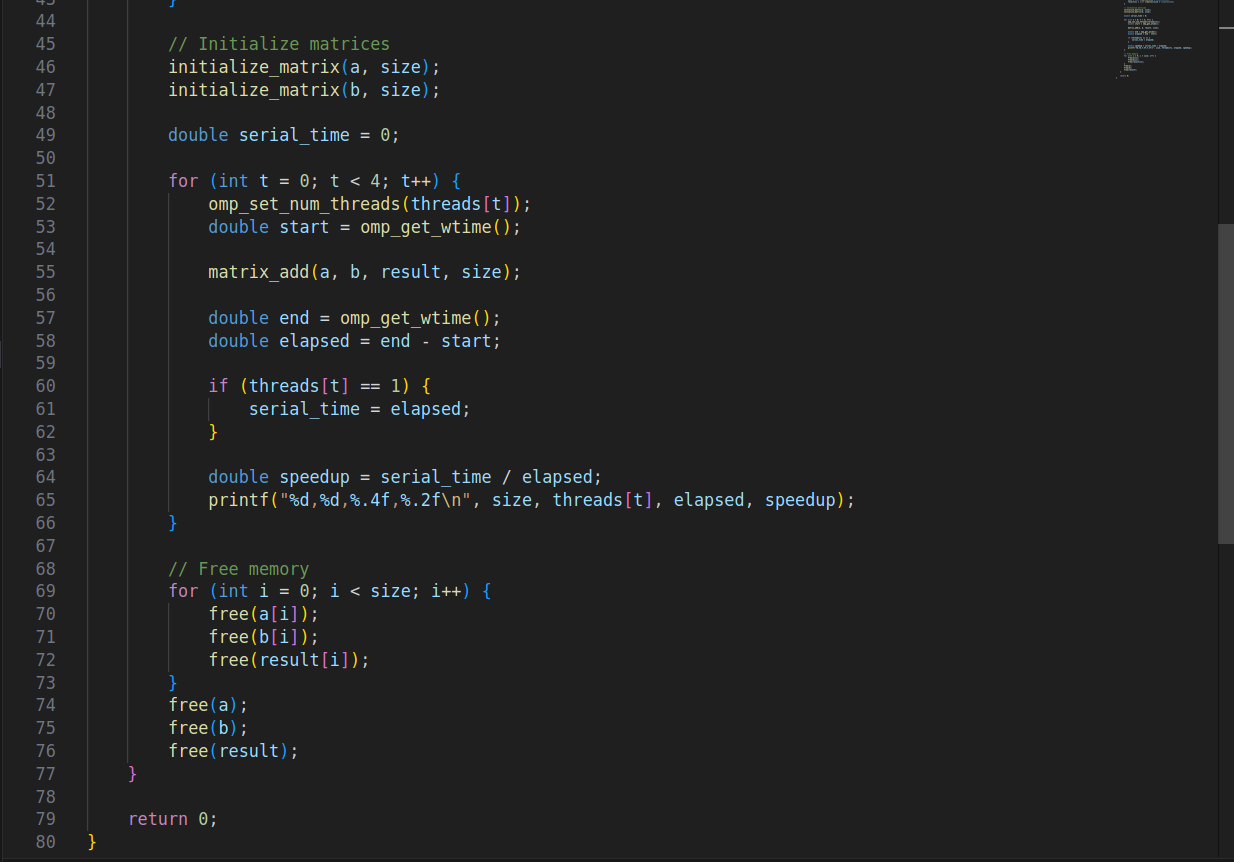
Write OpenMP code for two 2D Matrix addition, vary the size of your matrices from 250, 500, 750, 1000, and 2000 and measure the runtime with one thread (Use functions in C in calculate the execution time or use GPROF)

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.

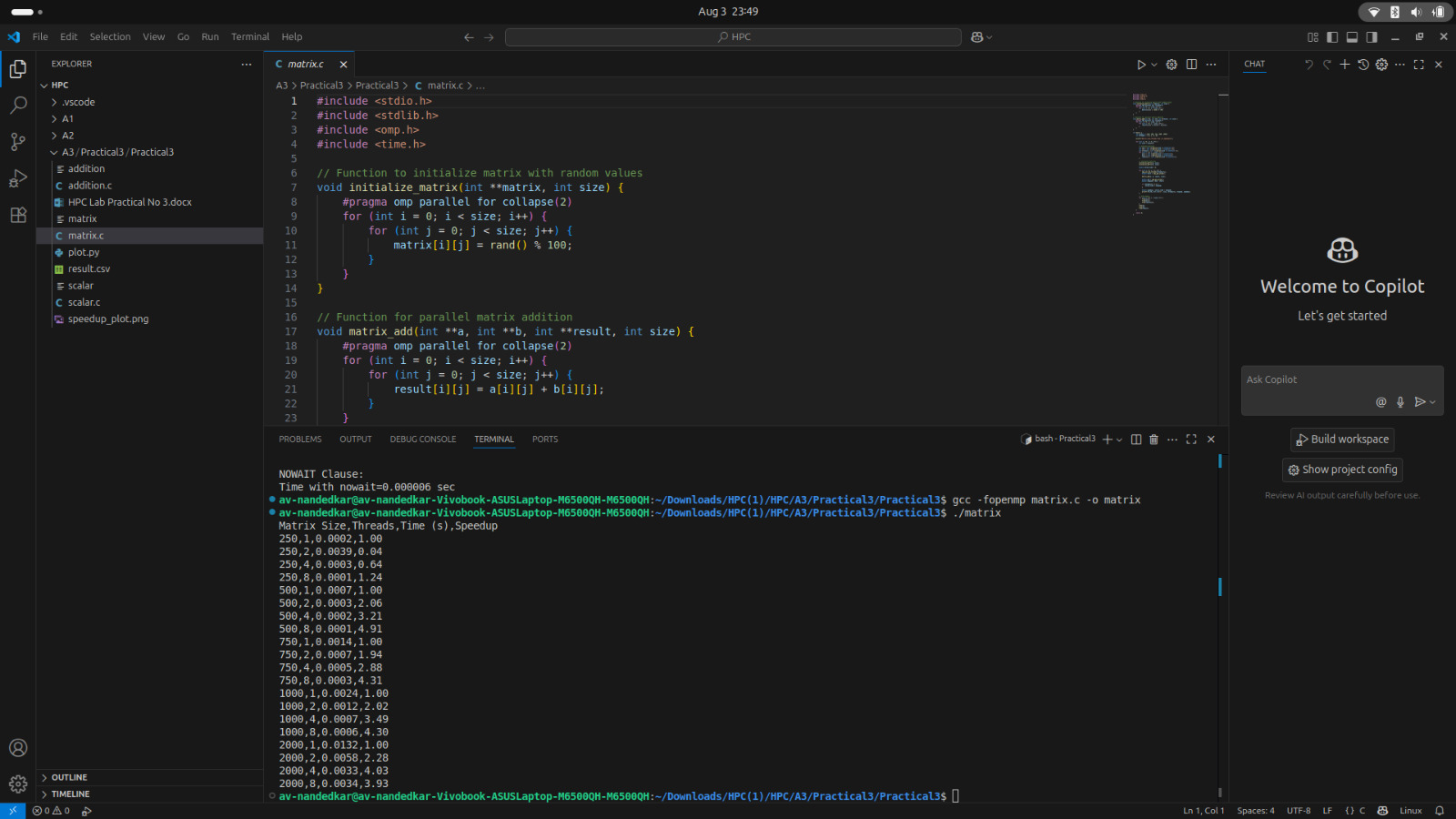
**Screenshots:**

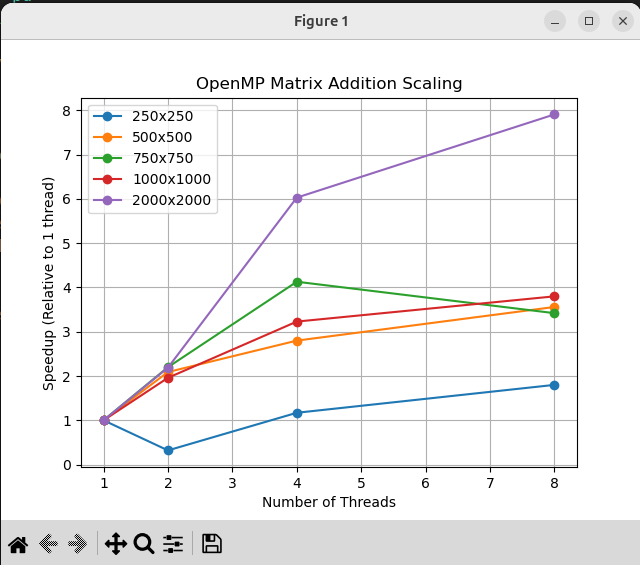
****

****

****

**Information and analysis:**



****

### Small Matrices (250×250)

High overhead dominates: Thread setup takes more time than the actual computation.

Limited speedup: Parallel gains can't overcome the fixed costs of going parallel.

### 2. Medium Matrices (500×750)

Better utilization: More work per thread reduces overhead impact.

Sublinear scaling: Speedup improves but hits memory/cache bottlenecks.

### 3. Large Matrices (1000×2000)

Computation wins: Work is large enough to mask parallel costs.

Near-linear scaling: More threads = proportional speedup (ideal case).

### 4. Why Scaling Isn’t Perfect

Parallel overhead: Creating/managing threads isn’t free.

Memory limits: Too many threads fight for memory bandwidth.

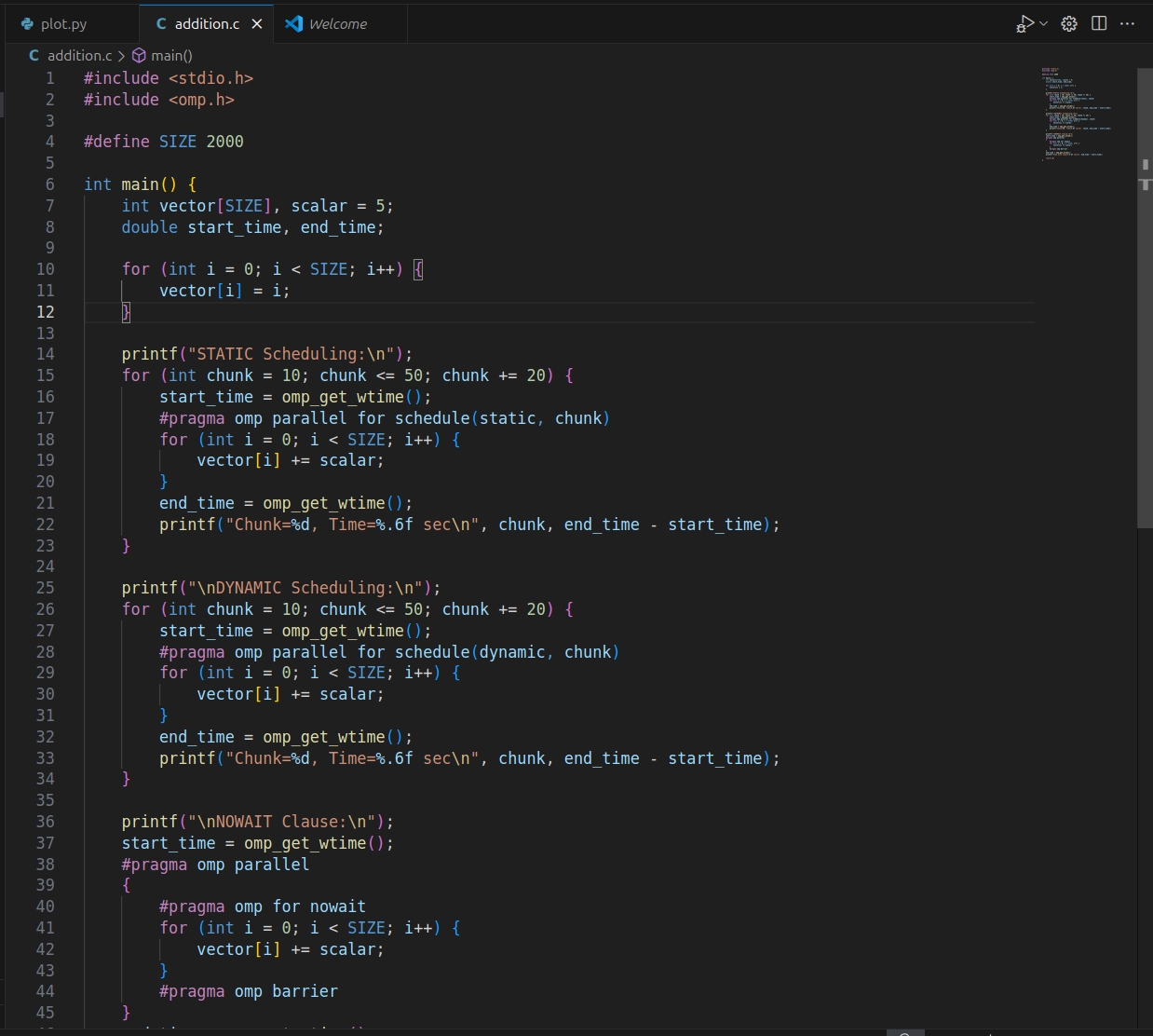
Cache issues: Threads may accidentally share cache lines (false sharing).

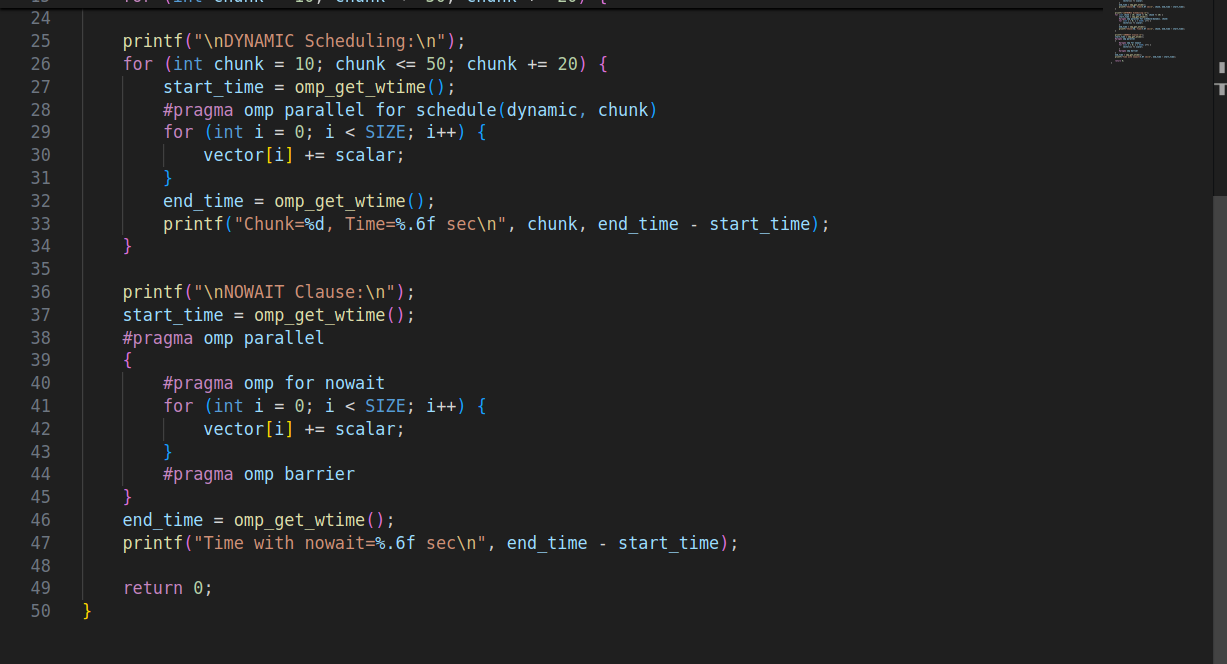
Load imbalance: Static work splits may leave threads idle.

**Problem Statement 3:**

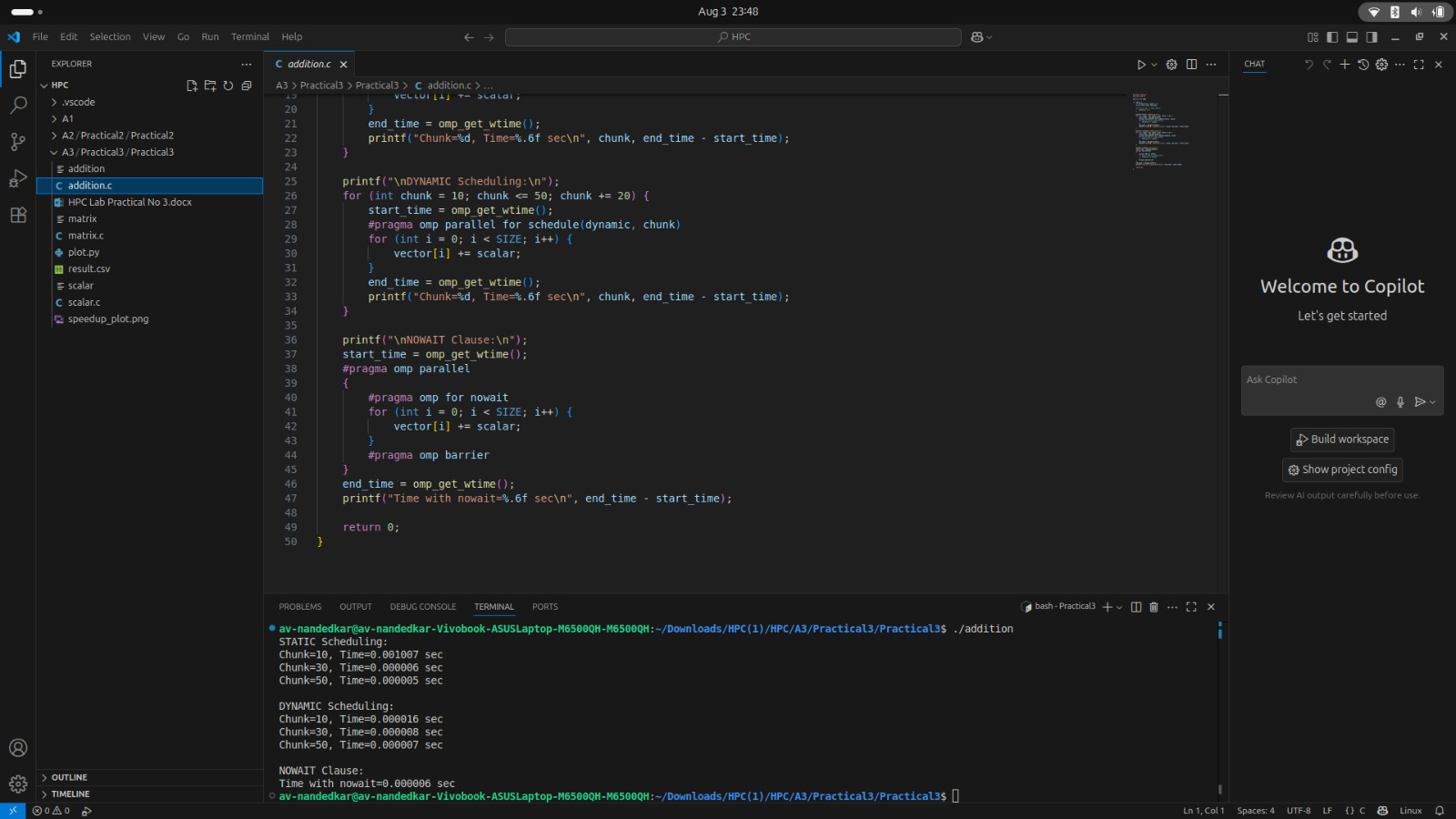
For 1D Vector (size=200) and scalar addition, Write a OpenMP code with the following: i. Use STATIC schedule and set the loop iteration chunk size to various sizes when changing the size of your matrix. Analyze the speedup. ii. Use DYNAMIC schedule and set the loop iteration chunk size to various sizes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

**Screenshots:**

****

****

**OUTPUT:**

****

### ****OpenMP Scheduling and Synchronization Technique****

schedule(static, chunk):

**Static scheduling** pre-assigns fixed-sized chunks of iterations to threads before the loop begins.

Ideal when iteration times are uniform.

Example: With 4 threads and chunk size 10, thread 0 might handle iterations 0–9, 40–49, etc.

Varying the chunk size (e.g., 10 to 50) helps analyze how granularity affects load balance and performance.

schedule(dynamic, chunk):

**Dynamic scheduling** assigns chunks to threads on-the-fly as they become available.

Suitable when iteration workloads vary, ensuring better utilization since idle threads can take on new work immediately.

Example: If thread 0 finishes early, it fetches the next available chunk instead of waiting.

#pragma omp for nowait:

Runs the loop in parallel without forcing threads to wait at the end.

Eliminates the default synchronization barrier, allowing threads to proceed independently.

#pragma omp barrier

Manually enforces synchronization at specific points.

Useful when you want explicit control over thread coordination instead of relying on default barriers.

**Github Link:**

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