**Name: Manaswi Mahesh Devekar**

**PRN:** 22510035

**Batch:** B-1

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

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

**Course:** High Performance Computing Lab

[GitHub repo](https://github.com/manaswi77/HPCL/tree/main)

**Practical No. 3**

**Title of practical:**

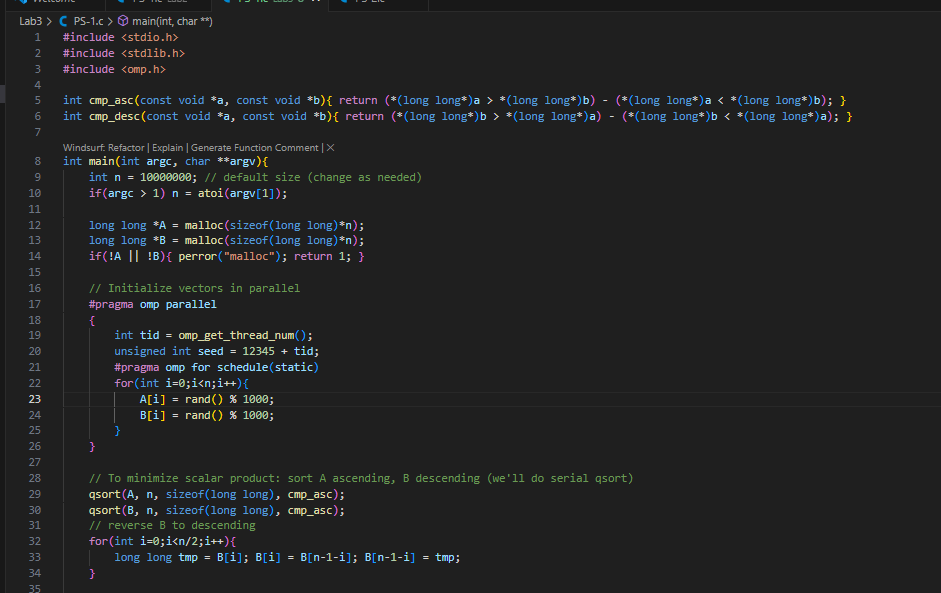
Study and Implementation of schedule, nowait, reduction, ordered and collapse clauses

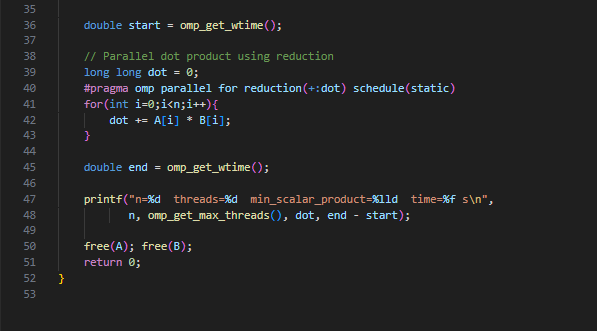
**Problem Statement 1:**

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

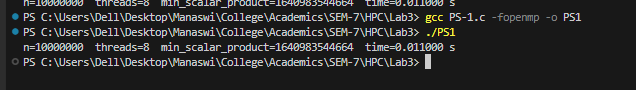
// C Program to find the minimum scalar product of two vectors (dot product)

**Screenshots:**

****

****

**Output:**

****

**Information:**

* Purpose: compute the minimal scalar (dot) product between two vectors by permuting them optimally: sort one ascending and the other descending, then compute dot product.
* OpenMP **clauses used: parallel, for, reduction, schedule(static).**

**Analysis:**

* Dot product calculation benefits from reduction — eliminates race conditions while accumulating in parallel.
* Sorting remains serial in this implementation (qsort) and becomes a serial bottleneck for very large n. (You can mention using parallel sort libraries to remove this bottleneck.)
* For large n, the time to compute dot product drops approximately inversely with thread count until memory bandwidth and cache behavior limit scaling.
* Include a table of times and calculate speedup = T1 / Tp and parallel efficiency = speedup / p. Discuss deviation from linear speedup and causes (sort time, memory bandwidth, false sharing if present).

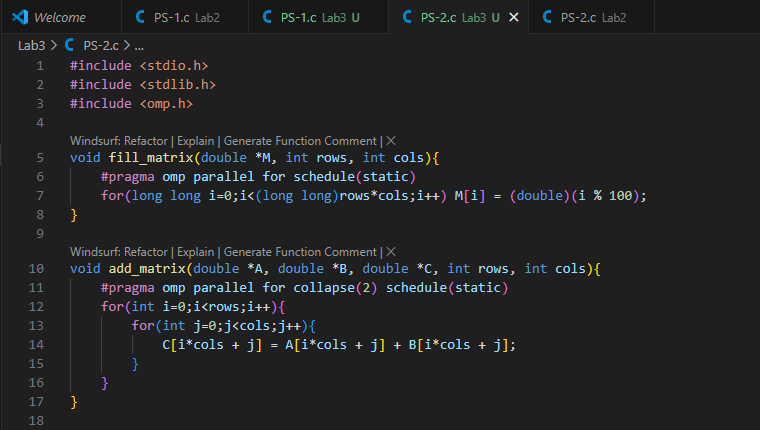
**Problem Statement 2:**

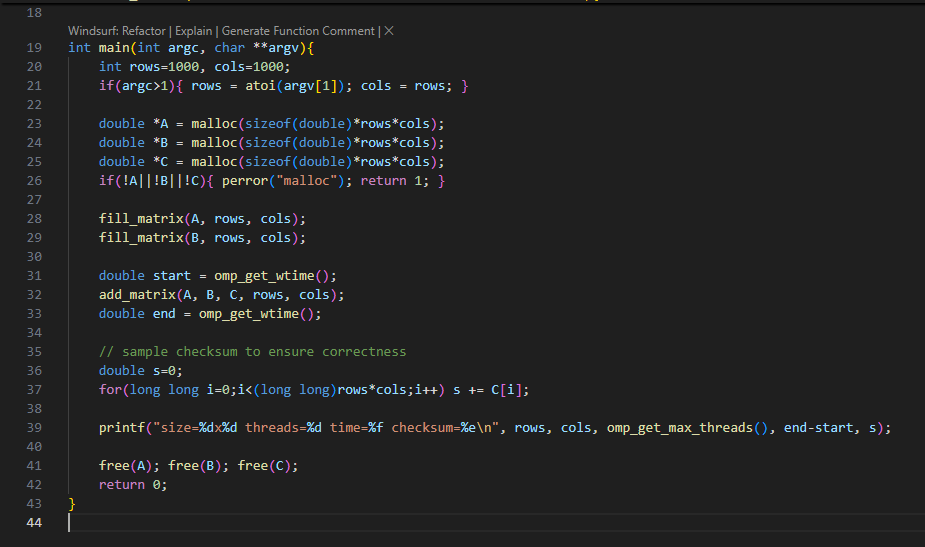
Write OpenMP code for two 2D Matrix addition, vary the size of your matrices from 250, 500, 750, 1000, and 2000 and measure the runtime with one thread (Use functions in C in calculate the execution time or use GPROF)

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

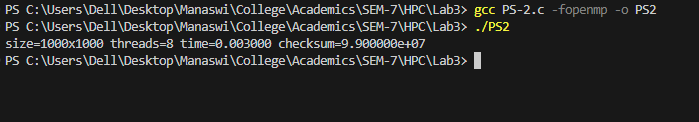
ii. Explain whether or not the scaling behaviour is as expected.

**Screenshots:**

****

****

**Output:**

****

**Information:**

* Program adds two matrices element-wise. Uses collapse(2) to allow the two nested loops to be distributed evenly across threads.
* Timings measured using omp\_get\_wtime().

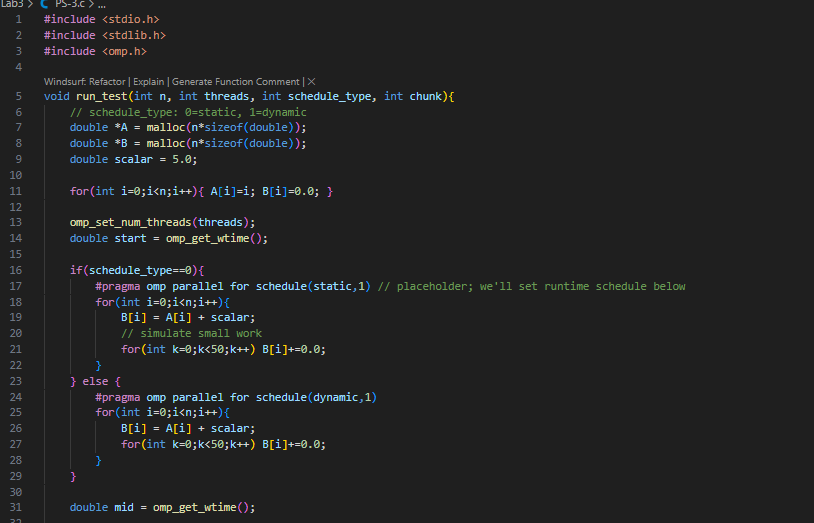
**Analysis:**

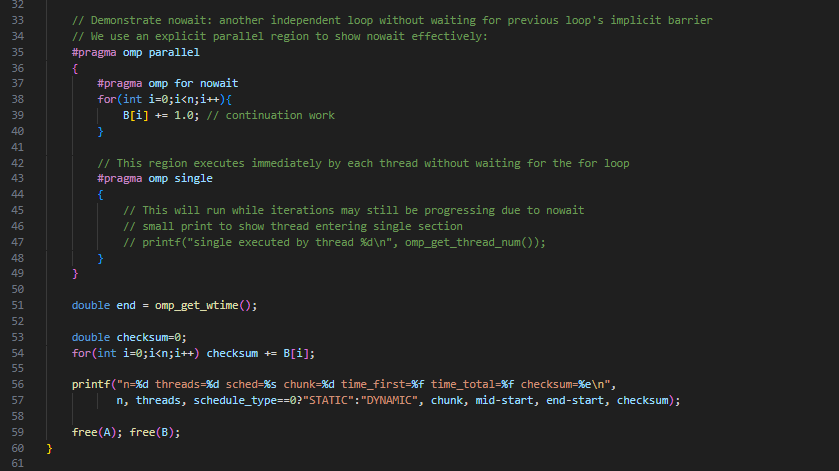
* For larger matrices (1000, 2000), compute-to-memory ratio is moderate: memory bandwidth and cache effects limit scaling.
* For small matrices (250), overhead and load imbalance may make parallel runs slower than serial.
* Ideal scaling: speedup ≈ p for p up to number of physical cores — but real behavior is often sub-linear due to:
  + Memory bandwidth limits (multiple threads saturating memory).
  + Cache misses and false sharing.
  + OpenMP overhead for thread scheduling for small workloads.
* Plot interpretation: if speedup curve flattens as threads increase, mention memory-bound nature and that adding threads yields diminishing returns.

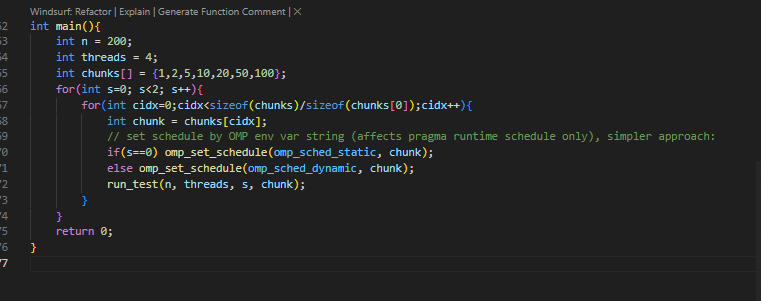
**Problem Statement 3:**

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.

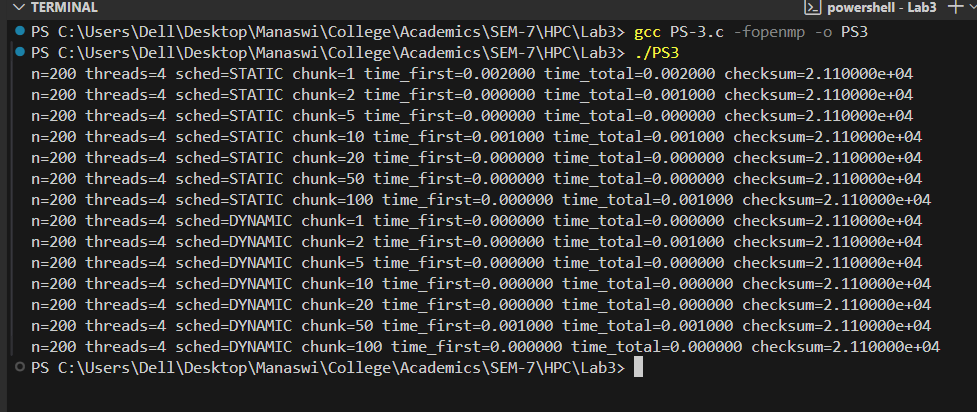
**Screenshots:**

****

****

****

**Output:**

****

**Information:**

* STATIC schedule divides iterations into equal blocks assigned to threads. chunk controls block size.
* DYNAMIC schedule assigns chunks to threads on demand; helpful for load-imbalanced loops.
* nowait removes implicit barrier at the end of for so threads that finish early proceed to next work immediately.

**Analysis:**

* STATIC: For balanced workloads (equal per-iteration work) static often gives better performance due to lower scheduling overhead and good cache locality.
* DYNAMIC: Useful when iterations have variable execution time — reduces load imbalance but increases scheduling overhead.
* Chunk size effect:
* Very small chunk sizes increase scheduling overhead (many assignments).
* Very large chunk sizes can cause imbalance for dynamic workloads.
* Choose chunk to trade overhead vs load balance. For example with 200 iterations and 4 threads, chunk=50 gives 4 blocks — good static balance.
* nowait: Useful when subsequent work does not depend on the loop results — removing barrier improves overlap and reduces idle time.
* Provide a small table of times and speedups to show which schedule/chunk was best.