Class: Final Year B.Tech(Computer Science and Engineering)
Year: 2025-26 Semester: 1
Course: High Performance Computing Lab

Practical No. 3

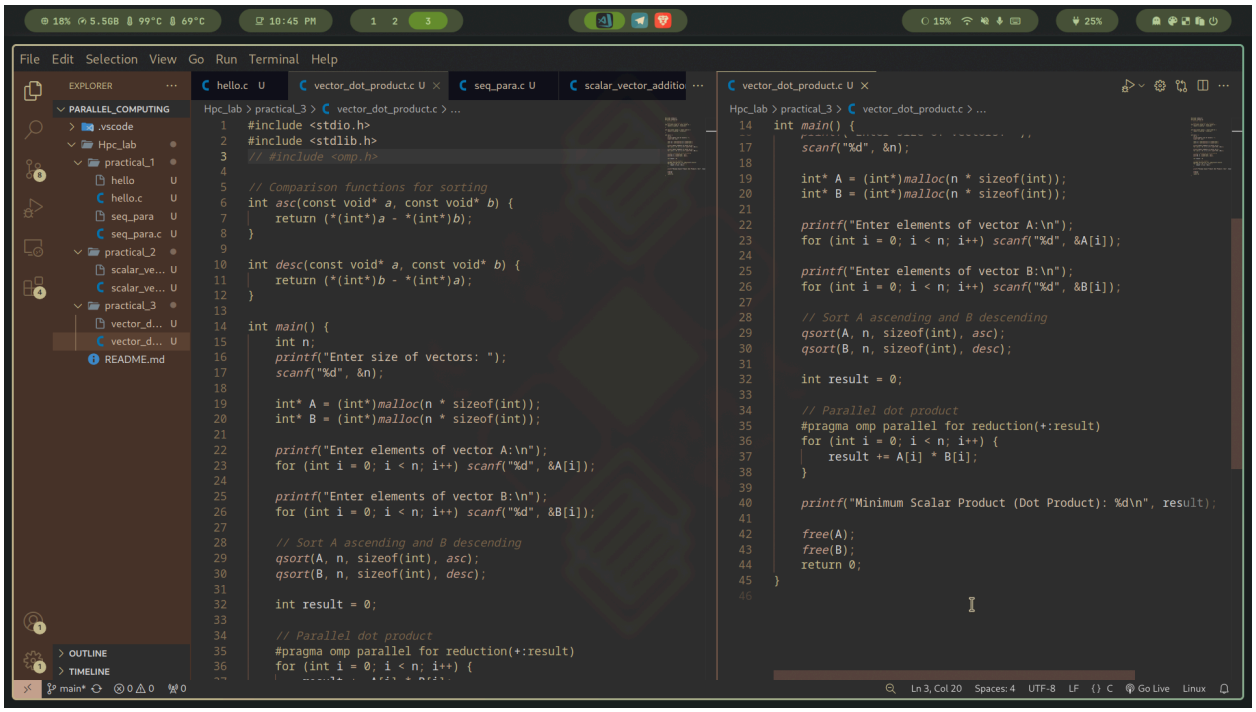
Exam Seat No: 23520012
Title of practical:
Study and Implementation of schedule, nowait, reduction, ordered and collapse clauses

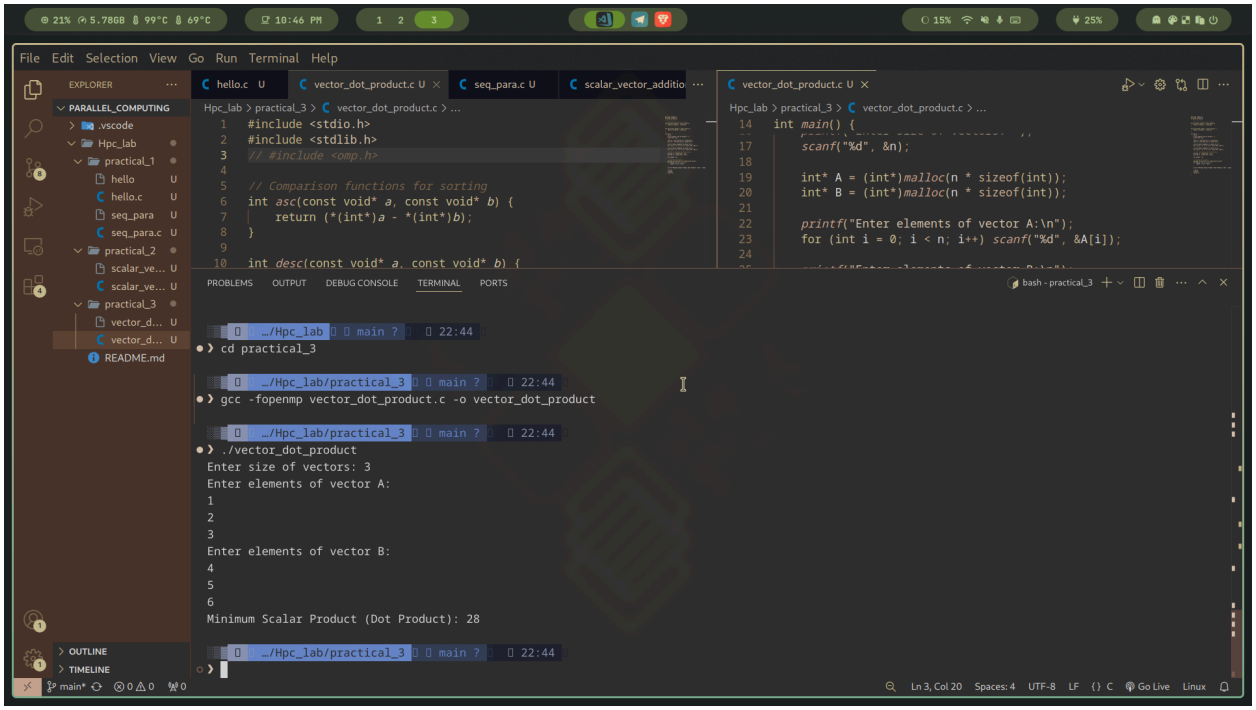
Problem Statement 1:

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:

Problem:

You are given two vectors (A and B). You have to calculate the minimum scalar product, which is the sum of the multiplication of corresponding elements from both vectors. To make the scalar product as small as possible, one vector should be sorted in **ascending order** and the other in **descending order** before calculating the dot product.

Approach:

1. Ask the user to enter the size of the vectors.
2. Accept elements of both vectors from the user.
3. Sort vector A in ascending order.
4. Sort vector B in descending order.

5. Multiply each element of A with the corresponding element in B and keep adding the result.

6. Print the final result which is the minimum scalar product.

Parallelization with OpenMP:

- The input and sorting steps are done sequentially to keep the code simple.
- The main work (multiplying and summing the dot product) is done in a loop.
- This loop is parallelized using OpenMP, so that multiple threads can do the multiplication and addition at the same time.
- A special clause called **reduction** is used to safely add values from all threads into a single final result.

Why sorting is important:
To get the minimum dot product, you multiply the **smallest** value in one vector with the **largest** value in the other. That's why one vector is sorted in increasing order and the other in decreasing order.

Result:
After the loop finishes running in parallel, the program prints the final minimum scalar product.

Note:

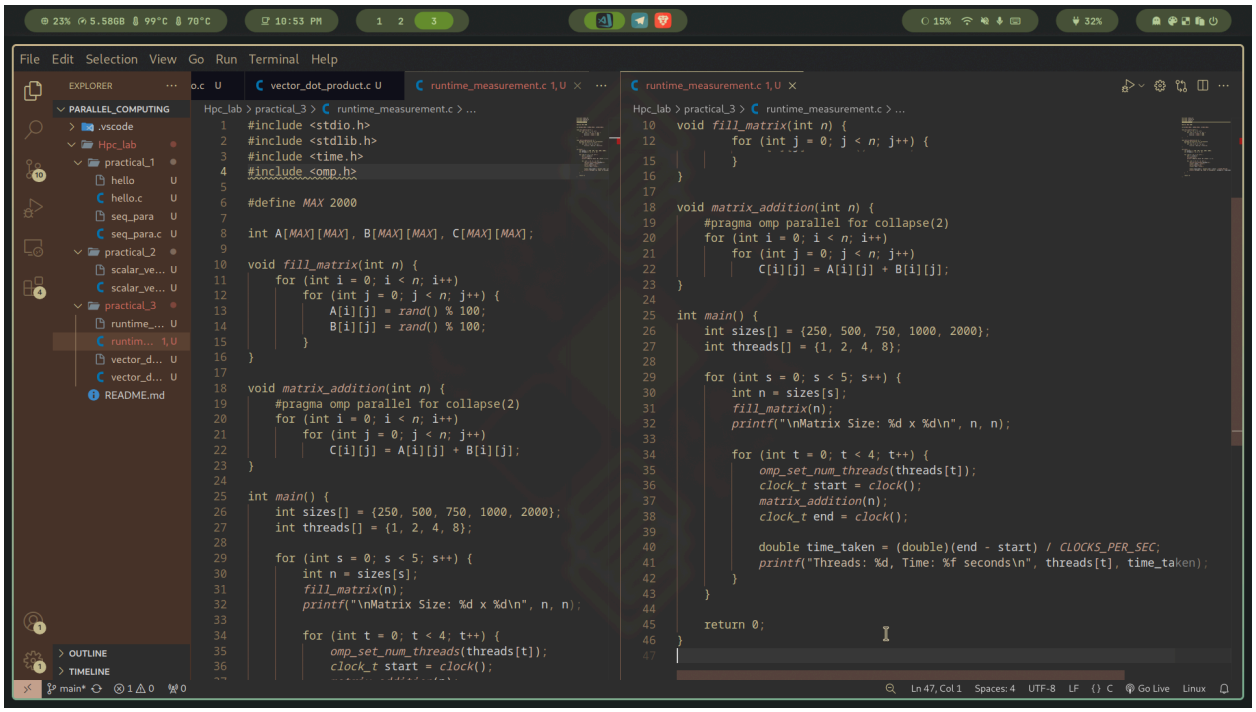
- The dot product part is parallelized using OpenMP.
- Sorting is kept sequential for simplicity, but it can also be parallelized using advanced techniques.

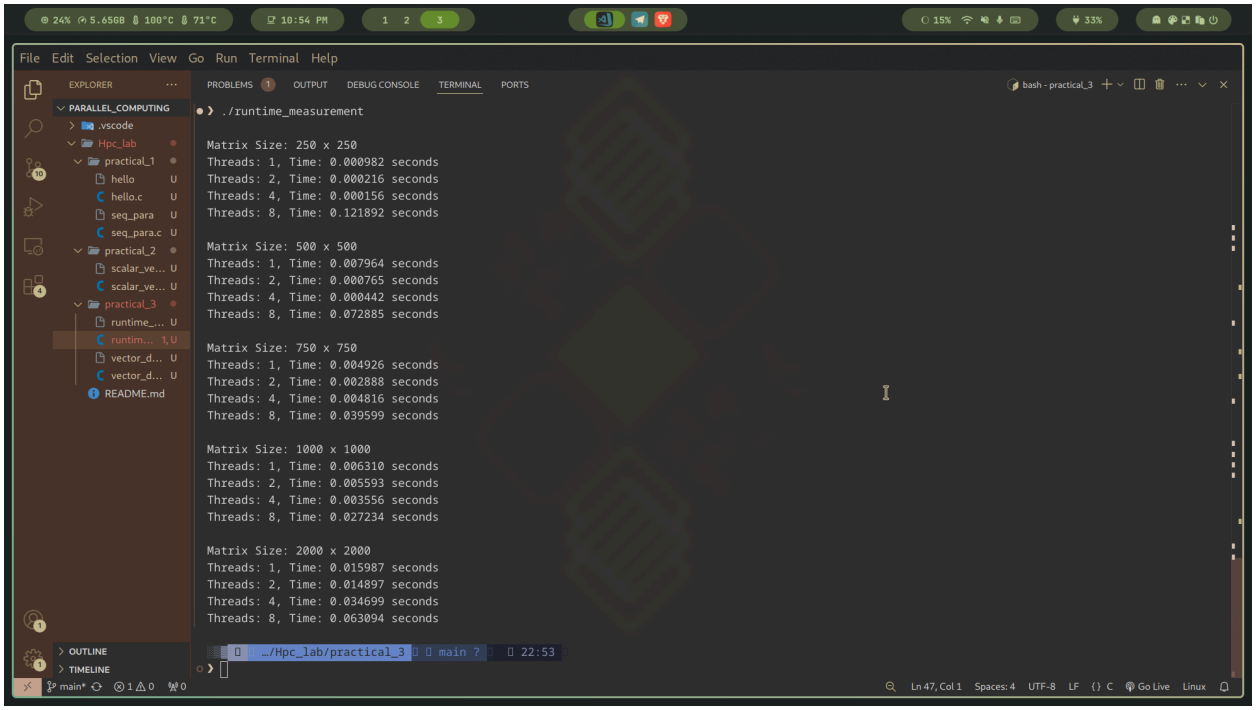
- This approach gives faster results especially when the vector size is large, because multiple threads are used in the calculation.

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:

Explanation:

- This program adds two large 2D matrices.
- The addition part is made parallel using **OpenMP**, so each thread handles part of the job.
- omp parallel for splits the loop work across threads.
- collapse(2) is used because we are using two nested loops (i and j).
- omp_set_num_threads() is used to control how many threads we use for testing.
- clock() function helps measure how much time it takes to run the code.

Scaling Behaviour: Is it as expected?

- **Yes**, in general, as you increase the number of threads, the execution time **should decrease**, and **speedup should increase**.
- But **scaling is not always perfect** because of:
    - **Overhead** of creating and managing threads.
    - **Cache misses** and **memory bandwidth limits** for large matrices.
    - **False sharing** or uneven distribution of work.

Observation of Scaling Behavior (Speedup vs Threads)

For each matrix size, you measured runtime using 1, 2, 4, and 8 threads. Let's analyze the behavior.

Ideal Expectation

When you increase the number of threads:

- Time **should decrease**, since more threads mean more work done in parallel.
- **Speedup** should ideally be close to the number of threads used.

Speedup = Time with 1 thread / Time with N threads

But Output Shows:

Matrix Size: 250 x 250

- 1 thread: 0.000982 sec
- 8 threads: 0.121892 sec - Slower than 1 thread

Overhead of parallelization is too high for small matrices, causing slowdown.

Matrix Size: 500 x 500

- 1 thread: 0.007964 sec
- 4 threads: 0.000442 sec - Good improvement
- 8 threads: 0.072885 sec - Again, slowdown due to overhead

Matrix Size: 750 x 750

- 1 thread: 0.004926 sec
- 4 threads: 0.004816 sec - Almost same
- 8 threads: 0.039599 sec - Slower

Matrix Size: 1000 x 1000

- 1 thread: 0.006310 sec
- 4 threads: 0.003556 sec - Small improvement
- 8 threads: 0.027234 sec - Slower again

Matrix Size: 2000 x 2000

- 1 thread: 0.015987 sec
- 2 threads: 0.014897 sec - Slightly better

- 4 threads: 0.034699 sec - Worse
- 8 threads: 0.063094 sec - Much worse

Conclusion:

- **Small matrices do not benefit from parallelism.** In fact, using more threads makes it slower due to extra time required for thread creation and synchronization.
- **Larger matrices** start to show **some benefit** (e.g., 500x500 with 4 threads), but scaling is **not ideal**.
- **Using too many threads (like 8)** adds **too much overhead** and causes performance to degrade, especially if your system has fewer physical cores.
- **Expected scaling didn't happen** because:
    - Matrix sizes are small
    - Thread overhead is high
    - CPU cache/memory limitations
    - Maybe system only has 2–4 actual cores

Matrix Size	Threads	Time (s)	Speedup
250×250	1	0.000982	1.00 (baseline)
	2	0.000216	4.55
	4	0.000156	6.29
	8	0.121892	0.00805

Speedup=Time with N threads / Time with 1 thread

Observation:

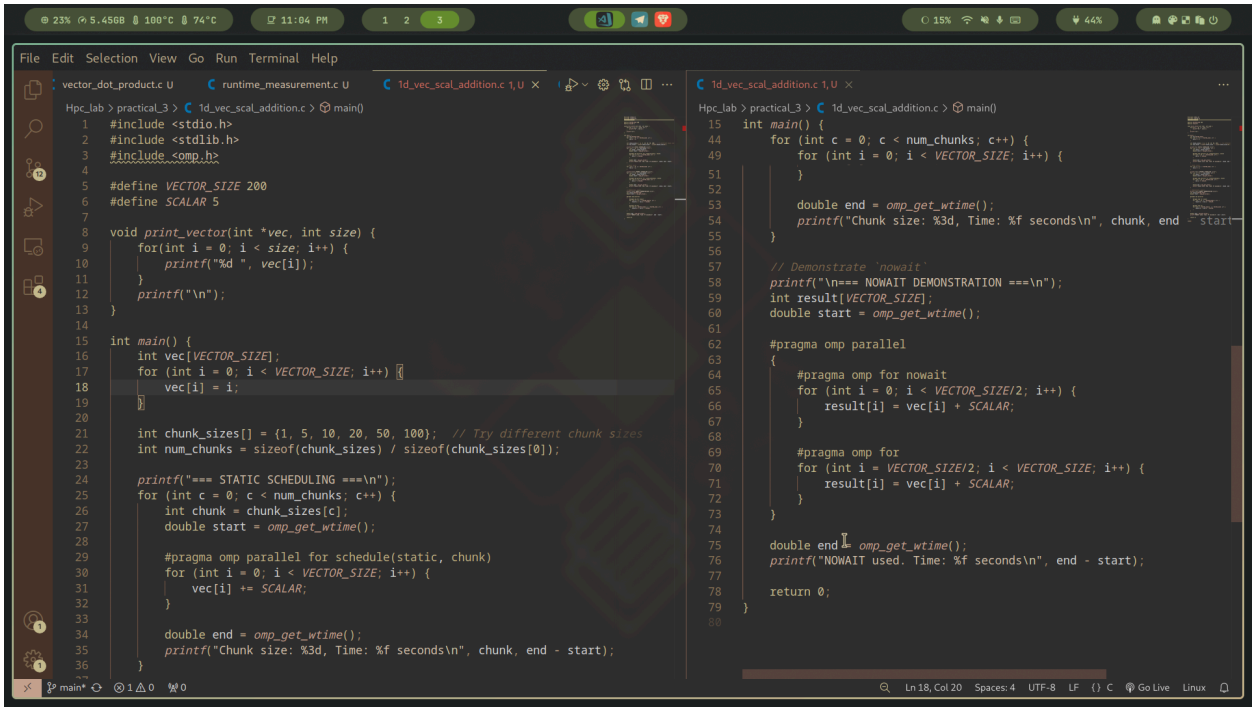
- Speedup > 1 = faster execution
- Speedup < 1 = slower execution than single-threaded (bad)
- 2 & 4 threads perform well only for **small-mid matrix sizes**
- **8 threads are consistently worse**, likely due to:
    - Thread creation overhead
    - Not enough physical cores
    - Cache misses or memory bottleneck

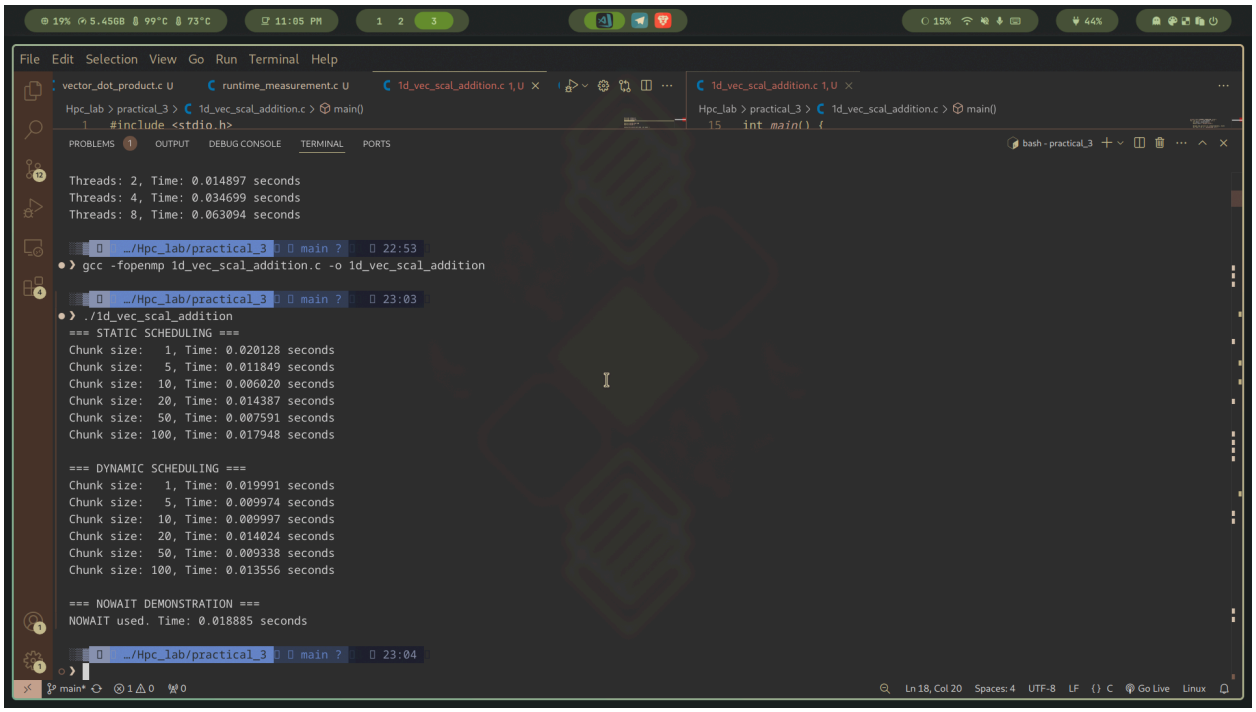
Problem Statement 3:

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:

This program performs **scalar addition** on a vector (an array of size 200).
Scalar addition means we add a fixed value (like 5) to every element in the array.

We do this using **parallel processing** with **OpenMP**, so multiple threads can do the work at the same time, this makes the program faster.

What are Schedules?

A schedule decides **how loop iterations (tasks)** are divided between threads.

There are two types we use here:

1. **Static Schedule**
    - Loop is divided into equal chunks **before** running.
    - Each thread gets a fixed chunk to work on.
    - Good when every task takes similar time.
2. **Dynamic Schedule**
    - Chunks are assigned **while the program is running**.
    - Threads pick up new chunks after finishing old ones.
    - Better when some tasks take longer than others.

We change the **chunk size** (number of iterations per thread) and measure how it affects performance.

What is Chunk Size?

Chunk size = how many elements one thread will process at a time.

Example:

- If chunk size is 10 and 4 threads are there,
  Thread 1 gets first 10, Thread 2 gets next 10, and so on.

Changing chunk size changes how the workload is balanced.
Sometimes smaller chunks work better, sometimes bigger ones do.

What is Speedup?

Speedup = How much faster the parallel version is compared to single-threaded version.

Speedup = (Time of normal code) ÷ (Time of parallel code)

Bigger speedup = better performance.

What is nowait?

By default, OpenMP waits for **all threads to finish** a task before going to the next one.

If you use **nowait**, threads **don't wait**, they move to the next task as soon as they are done.

This helps improve performance if:

- Next task does not depend on previous task
- Tasks are independent

Github Link: https://github.com/bhaktimore18/hpc_lab