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

**Year:** 2024-25 **Semester:** 7

**Course:** High Performance Computing Lab

**Practical No. 3**

**Exam Seat No: 21510018**

**Name: Jay Sandeep Chatpalliwar**

**Batch : B1**

**Title of practical:**

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

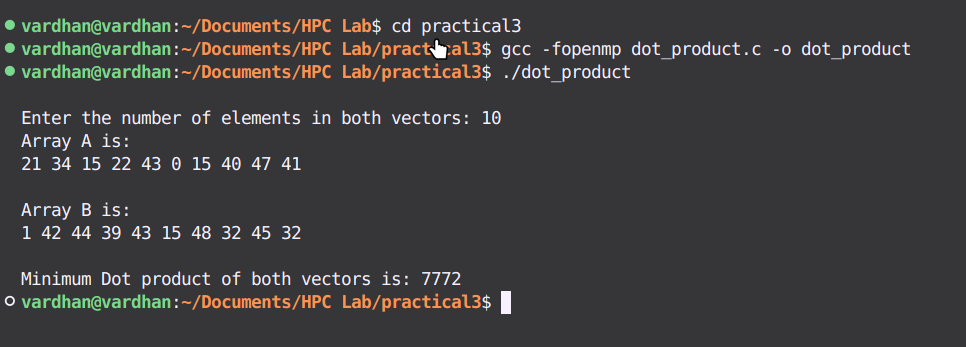
**Problem Statement 1:**

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:**

**Information**

* Objective: To calculate the minimum scalar product (dot product) of two vectors using OpenMP for parallelism.
* Vectors: Two vectors A and B are generated with random integers between 0 and 50.
* Sorting: Vector A is sorted in ascending order, and vector B is sorted in ascending order and then reversed to achieve descending order.
* Parallelism: OpenMP is used to parallelize the dot product calculation.
* Function Definitions: Comparison functions are used with qsort to sort the vectors.

### Analysis

1. Comparison Functions:
   * compare\_asc: Sorts vector A in ascending order.
   * compare\_desc: Sorts vector B in ascending order, then the array is reversed to get descending order. This avoids the need for a separate descending order comparison function.
2. Random Number Generation:
   * Random values between 0 and 50 are generated for the vectors, ensuring variability in the results.
3. Sorting and Reversal:
   * **qsort**: Used for sorting vectors A and B in ascending order.
   * Reversal: Reverses the sorted B to achieve descending order. This simplifies sorting B in descending order.
4. Parallel Computation:
   * OpenMP: The #pragma omp parallel for reduction(+:min\_scalar\_product) directive parallelizes the dot product calculation, efficiently distributing the work among multiple threads.
   * Reduction Clause: Ensures correct accumulation of the dot product result across threads by combining the results safely.
5. Compilation and Execution:
   * The program is compiled with -fopenmp to enable OpenMP support and executed to compute the minimum scalar product.

**Problem Statement 2:**

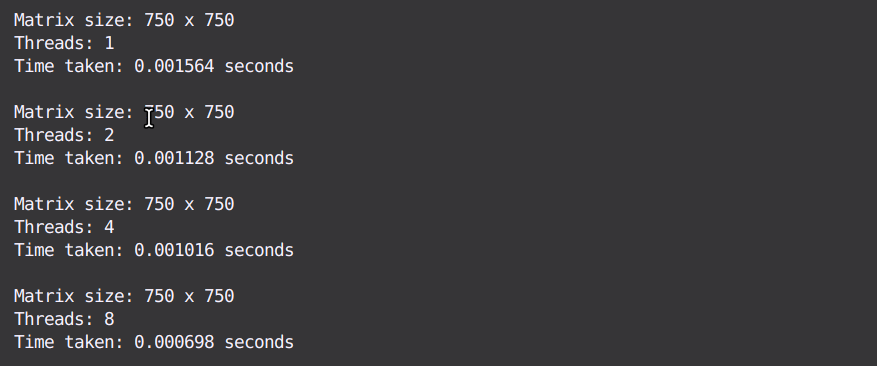
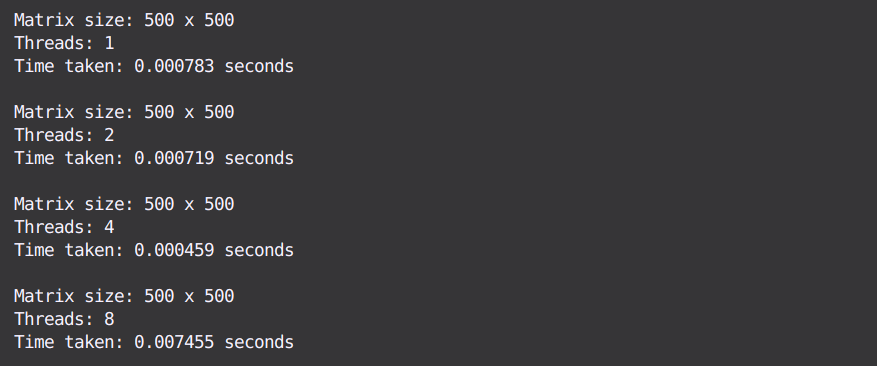
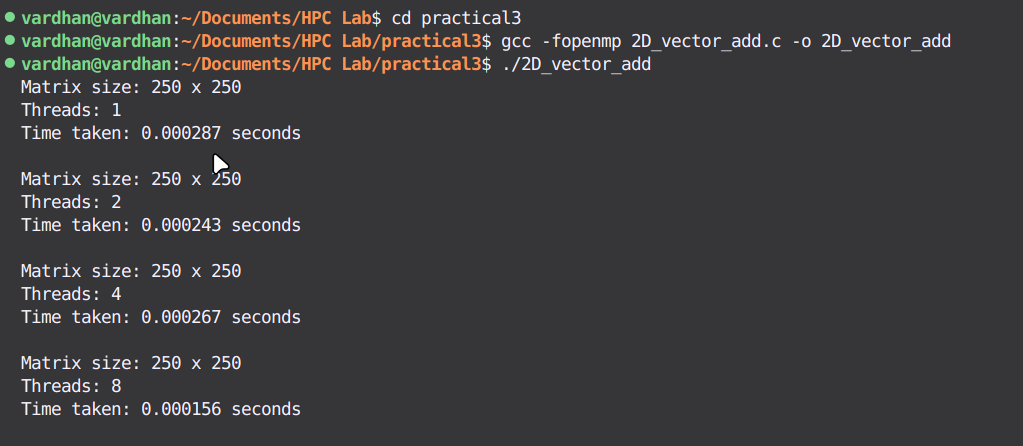
Write OpenMP code for two 2D Matrix addition, vary the size of your matrices from 250, 500, 750, 1000, and 2000 and measure the runtime with one thread

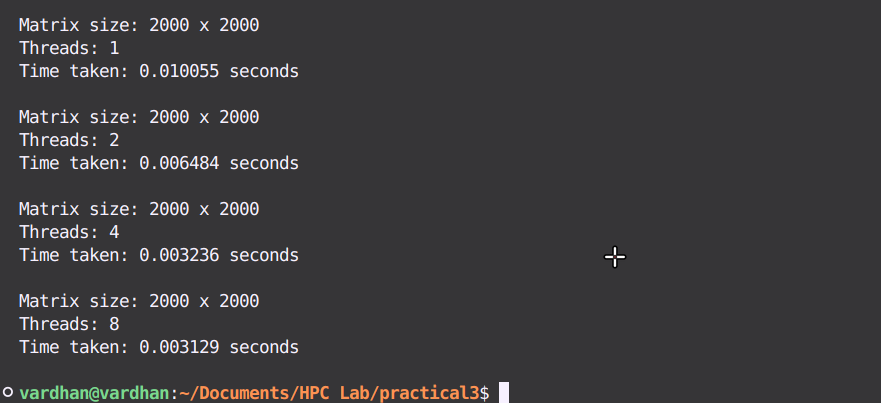
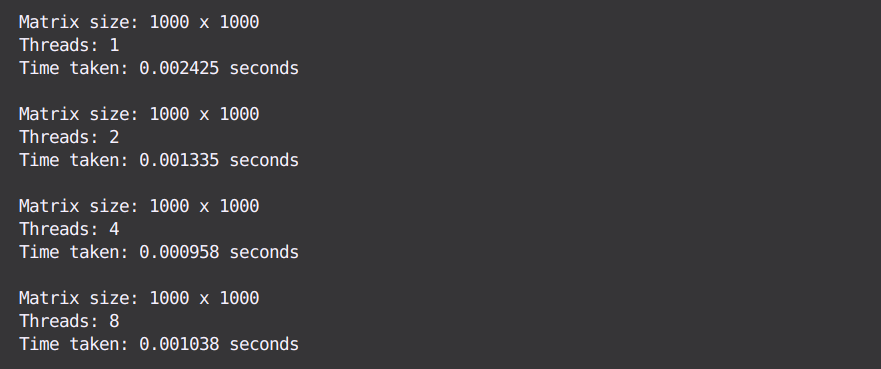
i. For each matrix size, change the number of threads from 2,4,8., and plot the speedup versus the number of threads.

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

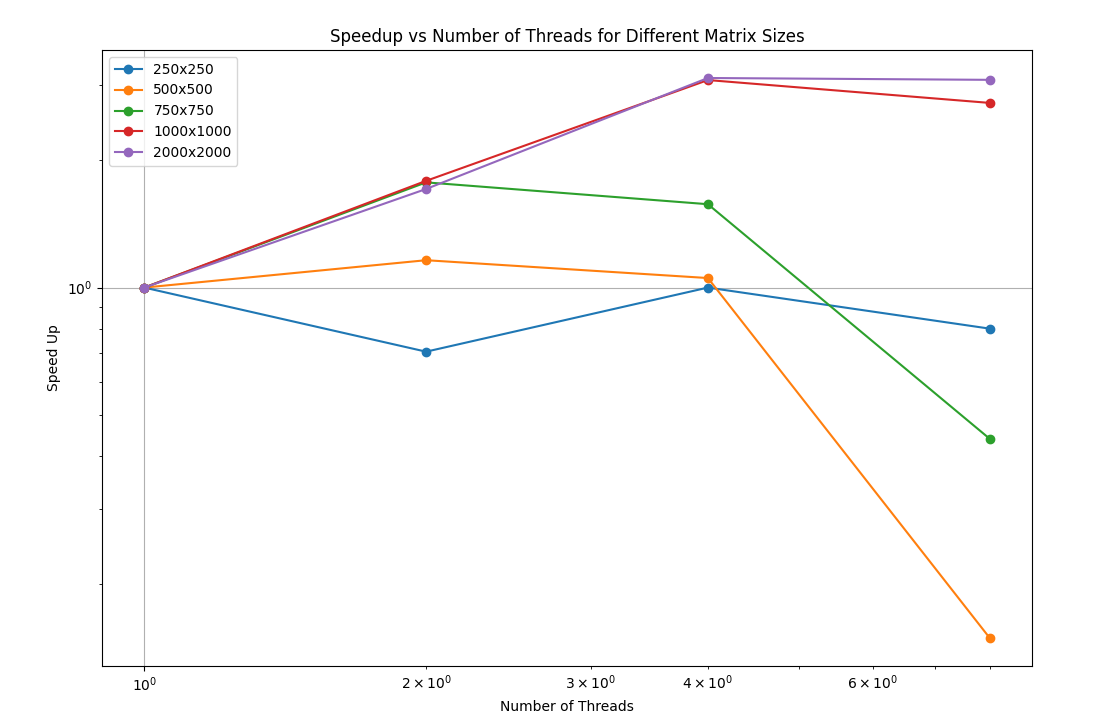
**Screenshots:**

****

****

****

**Speed Up Graph:**

****

**Explain whether or not the scaling behaviour is as expected:**

**-->**

Expected Behavior:

More threads should make matrix addition faster by dividing the work, leading to better speedup with increasing thread count.

Observed Behavior:

* Small matrices show little to no speedup with more threads due to overhead.
* Large matrices show good speedup as more threads are added.
* Reason: Overhead from thread management and memory contention affects performance with smaller matrices. Larger matrices benefit more from parallel processing.
* Thus, The scaling behavior aligns with expectations: more noticeable speedup with larger matrices and diminishing returns with smaller ones due to overhead.

### Information:

* Objective: Perform matrix addition using OpenMP to evaluate runtime performance with varying matrix sizes and thread counts.
* Matrices: Two matrices, A and B, of sizes 250x250, 500x500, 750x750, 1000x1000, and 2000x2000, are generated with random integer values between 0 and 99.
* Parallelism: OpenMP is used to parallelize the matrix addition operation by varying the number of threads (1, 2, 4, 8) to observe the effect on runtime.
* Performance Measurement: The runtime for each matrix size and thread count is recorded to analyze how the addition scales with increasing numbers of threads.
* Speedup Calculation: Speedup is calculated by comparing the runtime with one thread against the runtimes with multiple threads.

### Analysis:

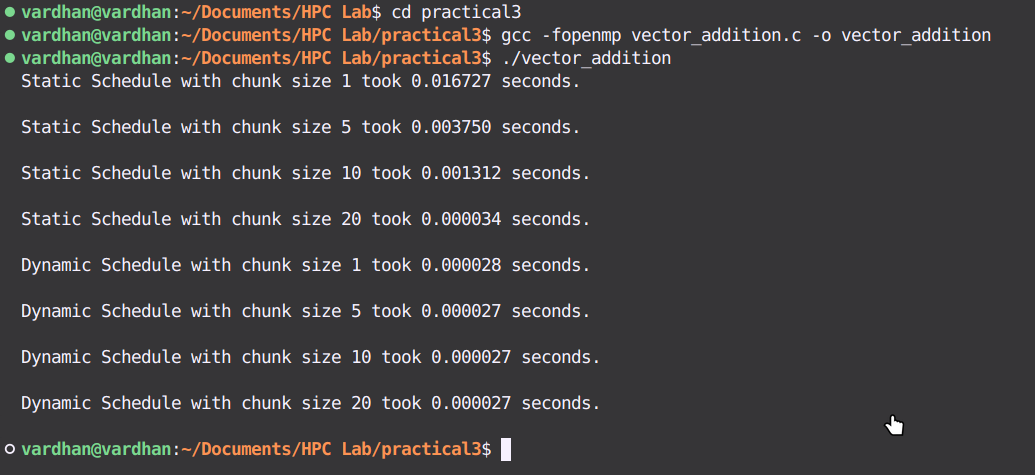
* Small Matrices: Limited speedup due to overhead from managing threads, sometimes leading to no performance improvement or slowdown.
* Large Matrices: Good speedup observed as more threads effectively handle the increased workload.
* Overhead: Thread management and memory contention can reduce efficiency, particularly for smaller matrices.
* Diminishing Returns: After a certain point, adding more threads yields little to no additional benefit due to serial code portions.
* Conclusion: Larger matrices show expected scaling; smaller matrices do not benefit significantly from parallelism due to overhead.

**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:**



****

### Analysis of Speedup

* Static Schedule: The loop iterations are divided into chunks of a fixed size and assigned to threads. By varying the chunk size, you can analyze how the division of work impacts the execution time. Typically, smaller chunk sizes may result in more balanced work distribution, especially when the workload is uneven.
* Dynamic Schedule: In this mode, chunks of loop iterations are assigned to threads dynamically as they finish processing their current chunk. This can reduce load imbalance but might introduce some overhead due to frequent task assignment.

**Use of nowait clause:**

**-->**

The nowait clause is used to prevent threads from synchronizing at the end of a parallel for loop. This can be useful when you have independent sections of code that don’t need to wait for each other to complete. The no-wait clause allows threads to proceed without waiting for others, which can be useful in scenarios where the subsequent code doesn't depend on the completion of the loop.

**Information and analysis:**

Static Scheduling

Breaks the loop into equal-sized chunks and assigns these chunks to threads before starting.

* Working:
  + Small chunks (e.g., 1, 5): Each thread gets small tasks, leading to better load distribution. However, this can increase the time spent managing tasks (overhead).
  + Large chunks (e.g., 10, 20): Fewer, bigger tasks mean less time managing them, but some threads might finish early and stay idle, causing imbalance.

### **Dynamic Scheduling**

Assigns chunks of work to threads as they finish previous tasks, rather than all at once.

* Working:
  + Small chunks: Threads get new work as soon as they finish, which keeps them busy and balances the load well. But, it can increase overhead due to frequent task assignment.
  + Large chunks: Less frequent task assignment reduces overhead, but it might not fully utilize threads if the workload varies.

### nowait **Clause**

Allows threads to skip waiting at the end of a loop, so they can start the next task right away.

* Working: Speeds up programs by letting threads move on to independent tasks without waiting for others, reducing delays.

**Github Link:** [**Practical 3 Repository**](https://github.com/jay-chatpalliwar/HPC-Lab/tree/master/practical3)