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

**Year:** 2024-25 **Semester:** 1

**Course:** High Performance Computing Lab

**Practical No. 3**

**Exam Seat No: 21510055**

**Title of practical:**

Study and Implementation of schedule, no wait, reduction, ordered, and collapse clauses

**Problem Statement 1:**

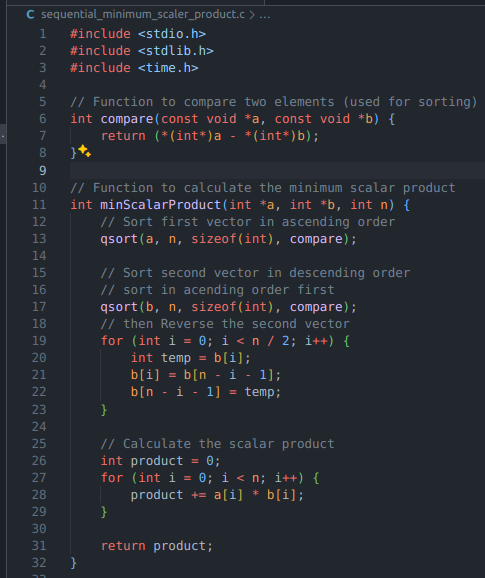
Analyse and implement a Parallel code for the below program using OpenMP.

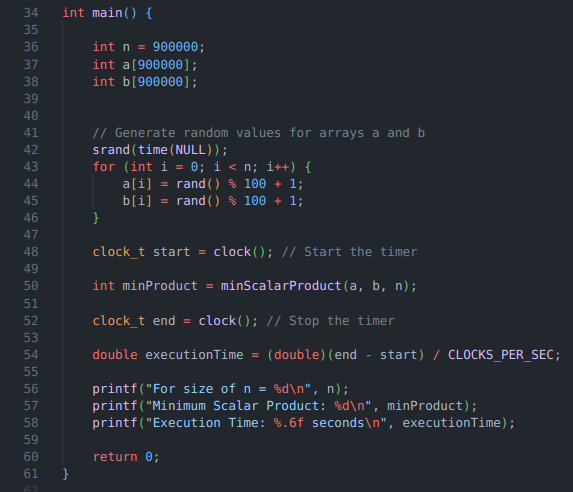
// C Program to find the minimum scalar product of two vectors (dot product)

**Screenshots:**

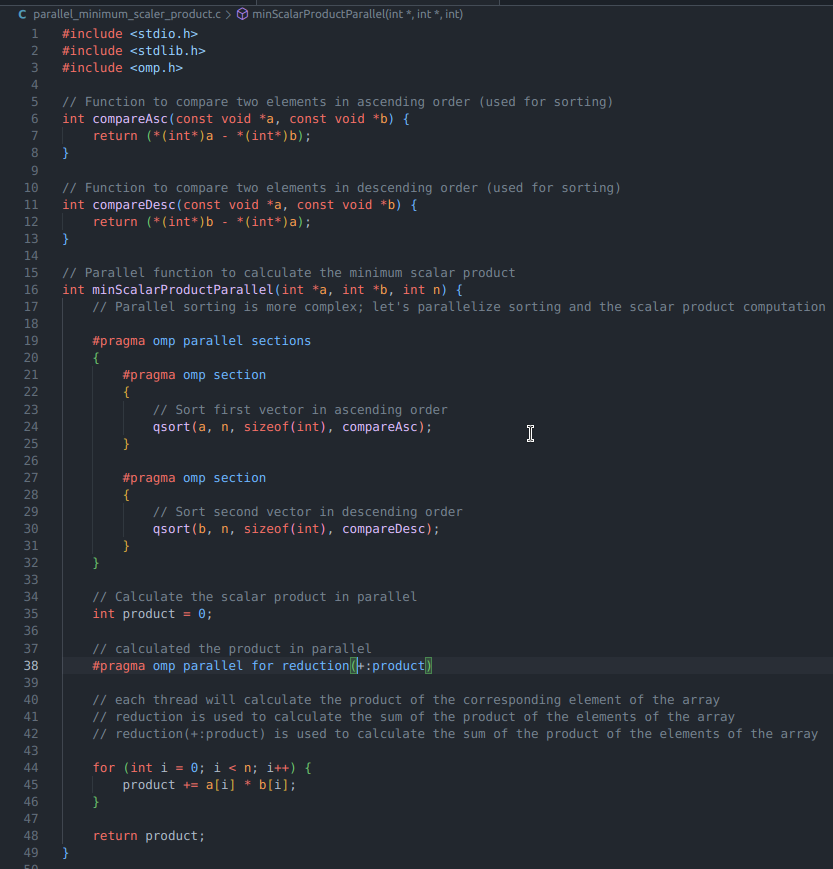
Code :

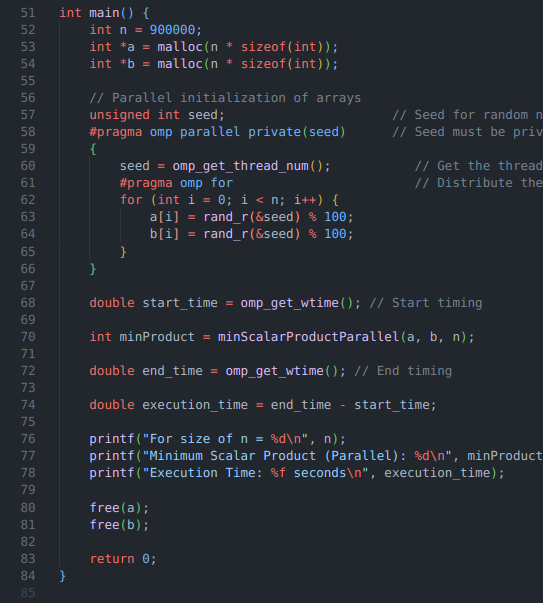
1. Sequential :



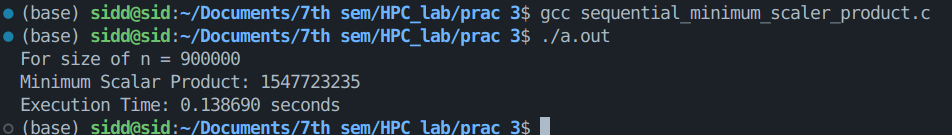


1. Parallel :

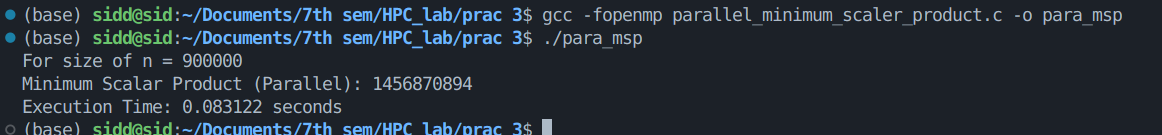




Using sequential :

****

Using parallel :



**Information and analysis:**

1. Both array values are filled using rand() function.
2. We sort array using qsort for both the codes, which is sequential.
3. After that in sequential code : we calculate the product iterating from 1 to N(9,00,000 in above case)
4. In case of parallel code : directive is used to parallelize the execution of a *for* loop across multiple threads, specifically focusing on safely handling accumulating a sum (*product*) across these threads.

**—----------------------------------------------------------------------------------------------------------------**

**Problem Statement 2:**

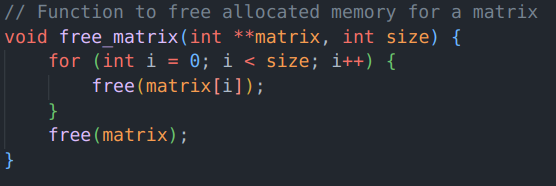
Write OpenMP code for two 2D Matrix addition, vary the size of your matrices from 250, 500, 750, 1000, and 2000 and measure the runtime with one thread (Use functions in C in calculate the execution time or use GPROF)

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.

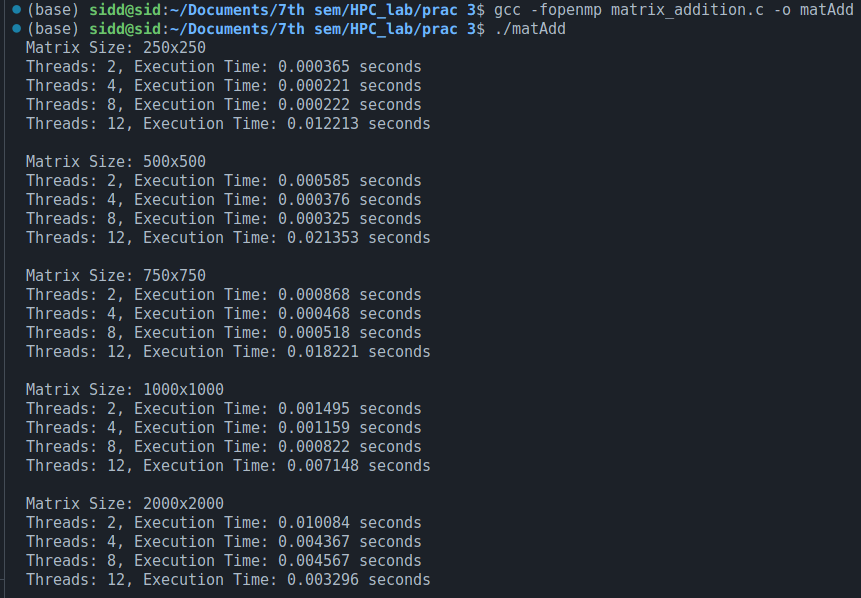
**Screenshots:**

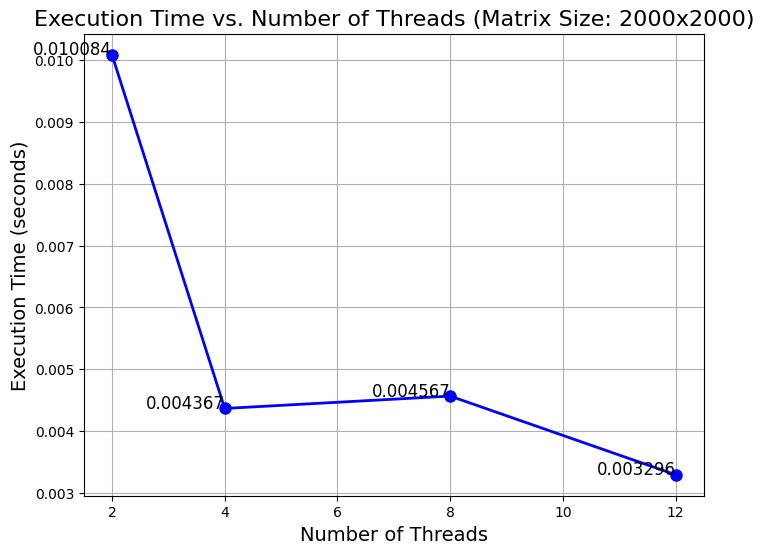
**Code :**



****

**Output :**

****

****

**Information and analysis:**

1. The time complexity of matrix addition is O(n^2).
2. #pragma omp parallel for => This line will create a team of threads to distribute the iterations of the loop across these threads, allowing multiple iterations to be executed concurrently.
3. The collapse clause is used to flatten multiple nested loops into a single loop.

for ex => if we are iterating on 3x3matrix then it will create a single loop having 9 iterations.

1. In an ideal scenario, doubling the number of threads would halve the execution time. This would suggest perfect parallelism with no overhead, which is typically reflected in a near-linear speedup.
2. **2 to 4 Threads:** The speedup is significant and close to what we might expect (almost a 2.3x reduction in time).
3. **4 to 8 Threads:** There's an unexpected slight increase in execution time. This could be due to various factors like memory bandwidth limitations, increased overhead in managing more threads, or cache contention.
4. **8 to 12 Threads:** The execution time decreases again, but the improvement is not as significant like from 2 to 4 threads.

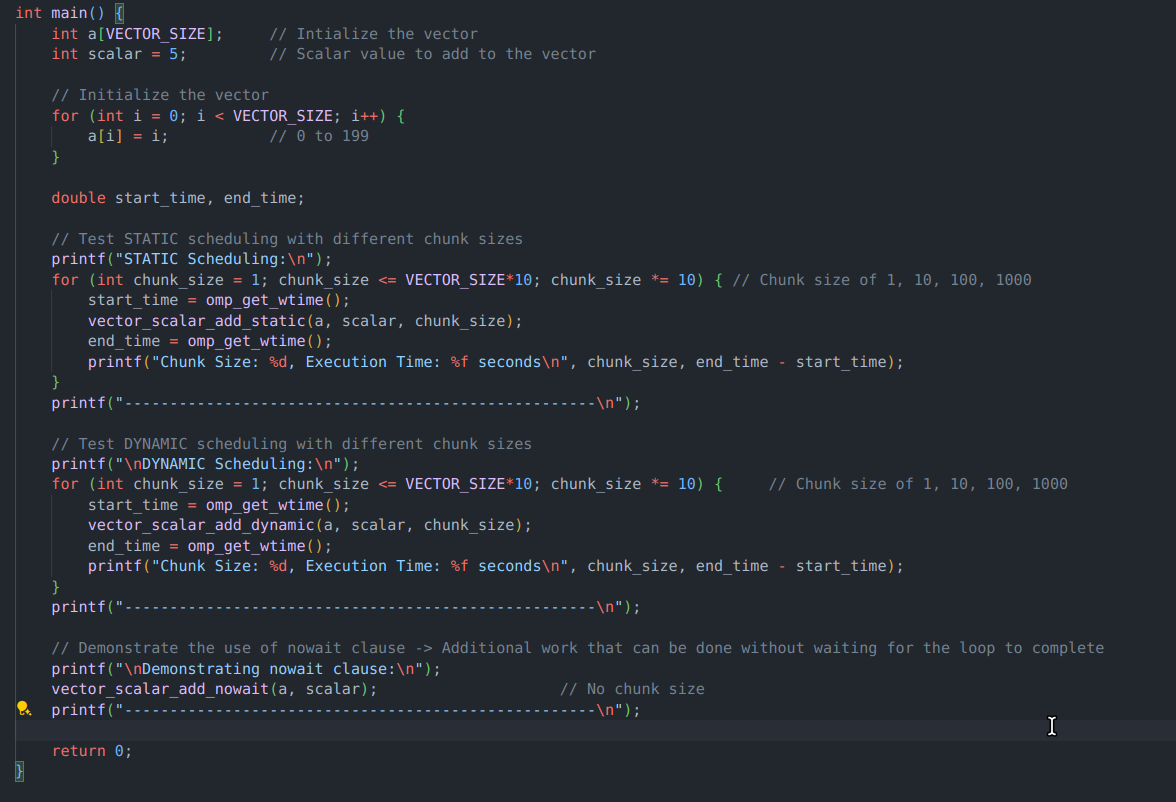
**—----------------------------------------------------------------------------------------------------------------**

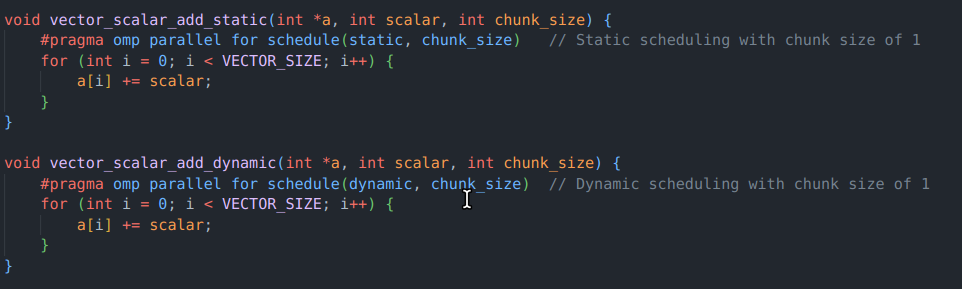
**Problem Statement 3:**

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.

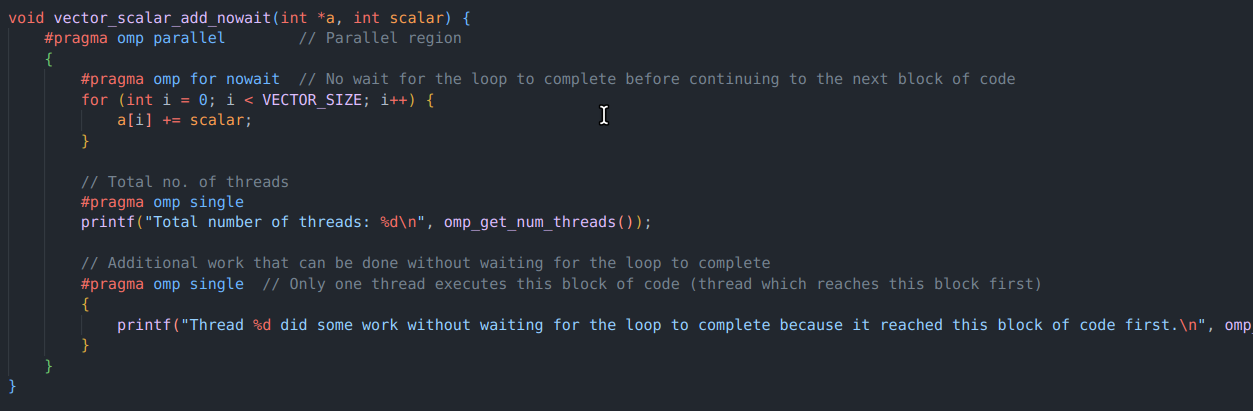
**Screenshots:**

**Code :**

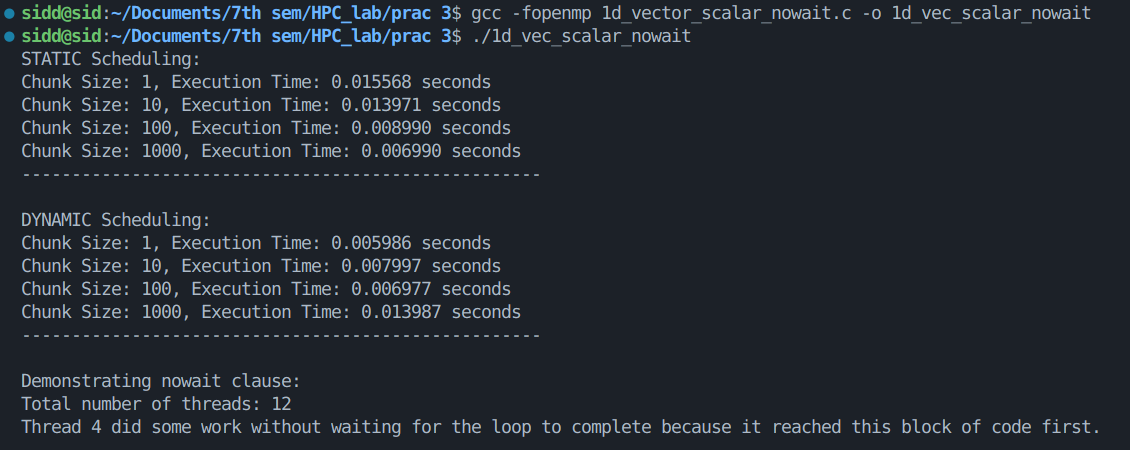
****

**2.**

**3.**

****

**Output :**

****

**Information and analysis:**

// Difference between Static and Dynamic scheduling

// Static scheduling divides the loop iterations into chunks of size chunk\_size and assigns each chunk to a thread in a round-robin fashion.

// Dynamic scheduling assigns a chunk of size chunk\_size to a thread, and when the thread finishes processing the chunk, it is assigned another chunk.

/\*

Simple example :

Let's say we have a loop with 100 iterations and 4 threads.

With Static scheduling and chunk size of 1, the iterations are divided as follows:

Thread 0: 0, 4, 8, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48, 52, 56, 60, 64, 68, 72, 76, 80, 84, 88, 92, 96,

Thread 1: 1, 5, 9, 13, 17, 21, 25, 29, 33, 37, 41, 45, 49, 53, 57, 61, 65, 69, 73, 77, 81, 85, 89, 93, 97,

Thread 2: 2, 6, 10, 14, 18, 22, 26, 30, 34, 38, 42, 46, 50, 54, 58, 62, 66, 70, 74, 78, 82, 86, 90, 94, 98,

Thread 3: 3, 7, 11, 15, 19, 23, 27, 31, 35, 39, 43, 47, 51, 55, 59, 63, 67, 71, 75, 79, 83, 87, 91, 95, 99

doesn't matter who findish first, the next chunk will be assigned in round-robin fashion.

With Dynamic scheduling and chunk size of 1, the iterations are divided as follows:

Thread 0: might get more than Thread 1

Thread 1: might get more than Thread 2

Example:

Thread 0: 0, 1, 2

Thread 1: 3, 4

Thread 0: 5, 6

Thread 2: 7, 8

Thread 3: 9, 10

Thread 0: 11, 12, 13, 14

Thread 1: 15, 16, 17

Thread 2: 18, 19

Thread 3: 20, 21

Whoever finishes first will get the next chunk.

\*/

nowait clause -> Additional work that can be done without waiting for the loop to complete

**Github Link:** [**https://github.com/Sid-1164/HPC\_lab/tree/main/prac%203**](https://github.com/Sid-1164/HPC_lab/tree/main/prac%203)