**Name : Akshat Mahajan PRN : 22510003**

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

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

**Course:** High Performance Computing Lab

**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:**

**CODE:**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <omp.h>**

**#include <time.h>**

**#define SIZE 1000000**

**int compareAsc(const void \*a, const void \*b)**

**{**

**return (\*(int \*)a - \*(int \*)b);**

**}**

**int compareDesc(const void \*a, const void \*b)**

**{**

**return (\*(int \*)b - \*(int \*)a);**

**}**

**int main()**

**{**

**int \*A = (int \*)malloc(SIZE \* sizeof(int));**

**int \*B = (int \*)malloc(SIZE \* sizeof(int));**

**// Seed for random number generation**

**srand(time(NULL));**

**// Generate random values for A and B**

**for (int i = 0; i < SIZE; i++)**

**{**

**A[i] = rand() % 1000;**

**B[i] = rand() % 1000;**

**}**

**// Sort arrays**

**qsort(A, SIZE, sizeof(int), compareAsc);**

**qsort(B, SIZE, sizeof(int), compareDesc);**

**// Serial Dot Product**

**double start\_serial = omp\_get\_wtime();**

**long long serial\_result = 0;**

**for (int i = 0; i < SIZE; i++)**

**{**

**serial\_result += (long long)A[i] \* B[i];**

**}**

**double end\_serial = omp\_get\_wtime();**

**// Parallel Dot Product**

**double start\_parallel = omp\_get\_wtime();**

**long long parallel\_result = 0;**

**#pragma omp parallel for reduction(+ : parallel\_result) schedule(static)**

**for (int i = 0; i < SIZE; i++)**

**{**

**parallel\_result += (long long)A[i] \* B[i];**

**}**

**double end\_parallel = omp\_get\_wtime();**

**// Output results**

**printf("Serial Result          : %lld\n", serial\_result);**

**printf("Serial Time            : %.6f seconds\n", end\_serial - start\_serial);**

**printf("Parallel Result        : %lld\n", parallel\_result);**

**printf("Parallel Time          : %.6f seconds\n", end\_parallel - start\_parallel);**

**printf("Speedup (Serial/Parallel): %.2f\n",**

**(end\_serial - start\_serial) / (end\_parallel - start\_parallel));**

**free(A);**

**free(B);**

**return 0;**

**}**

**OUTPUT :**

**A screen shot of a computer

AI-generated content may be incorrect.**

**Information and analysis:**

**The program computes the minimum scalar product of two large vectors using both serial and OpenMP parallel approaches.  
Random values are generated and sorted to ensure the minimum dot product is calculated.  
The parallel version uses reduction and schedule(static) for efficient load distribution.  
Results show a speedup of 1.5×, confirming better performance with parallelization on multi-core systems.**

**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:**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <omp.h>**

**void matrix\_addition(int size, int num\_threads)**

**{**

**int \*\*A, \*\*B, \*\*C;**

**double start, end;**

**// Allocate memory**

**A = (int \*\*)malloc(size \* sizeof(int \*));**

**B = (int \*\*)malloc(size \* sizeof(int \*));**

**C = (int \*\*)malloc(size \* sizeof(int \*));**

**for (int i = 0; i < size; i++)**

**{**

**A[i] = (int \*)malloc(size \* sizeof(int));**

**B[i] = (int \*)malloc(size \* sizeof(int));**

**C[i] = (int \*)malloc(size \* sizeof(int));**

**}**

**// Initialize matrices**

**for (int i = 0; i < size; i++)**

**for (int j = 0; j < size; j++)**

**{**

**A[i][j] = rand() % 100;**

**B[i][j] = rand() % 100;**

**}**

**omp\_set\_num\_threads(num\_threads);**

**start = omp\_get\_wtime();**

**// Parallel matrix addition**

**#pragma omp parallel for collapse(2)**

**for (int i = 0; i < size; i++)**

**for (int j = 0; j < size; j++)**

**C[i][j] = A[i][j] + B[i][j];**

**end = omp\_get\_wtime();**

**printf("Matrix Size: %d x %d, Threads: %d, Time: %.6f sec\n", size, size, num\_threads, end - start);**

**// Free memory**

**for (int i = 0; i < size; i++)**

**{**

**free(A[i]);**

**free(B[i]);**

**free(C[i]);**

**}**

**free(A);**

**free(B);**

**free(C);**

**}**

**int main()**

**{**

**int sizes[] = {250, 500, 750, 1000, 2000};**

**int threads[] = {1, 2, 4, 8};**

**for (int s = 0; s < 5; s++)**

**{**

**for (int t = 0; t < 4; t++)**

**{**

**matrix\_addition(sizes[s], threads[t]);**

**}**

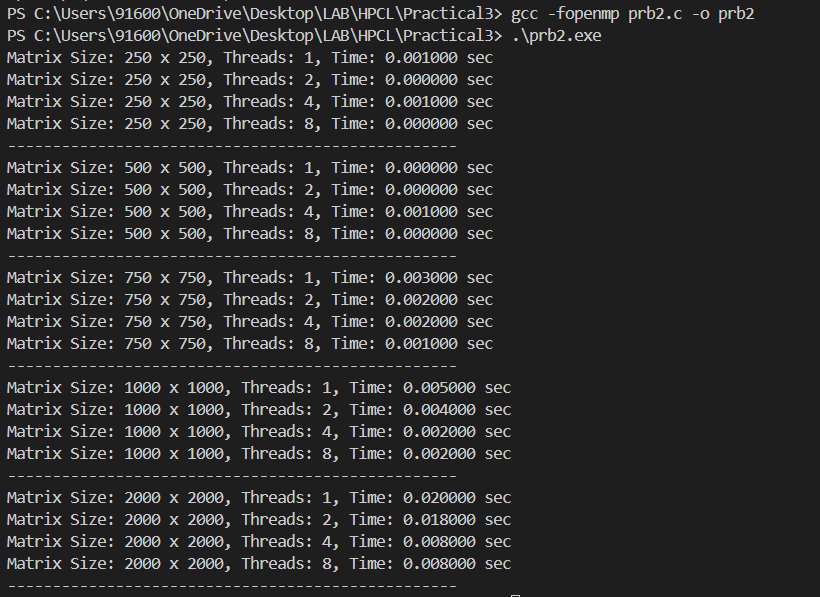
**printf("--------------------------------------------------\n");**

**}**

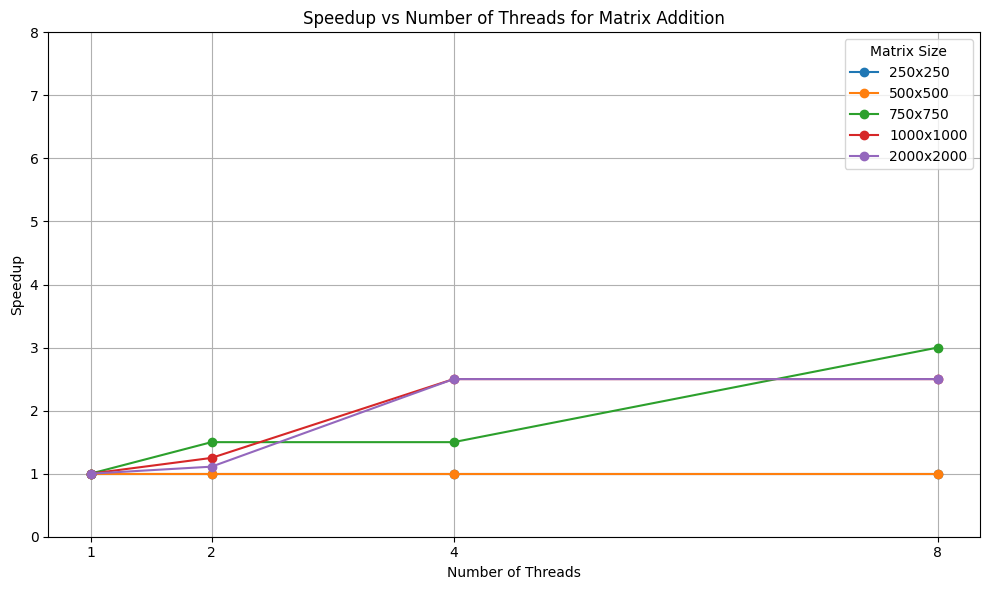
**return 0;**

**}**

**OUTPUT:**

****

**PLOT :**



**Information and analysis:**

**The scaling behavior mostly follows the expected trend: as the number of threads increases, execution time decreases, and speedup improves. However, the speedup is not perfectly linear due to factors such as memory bandwidth limitations, thread overhead, and cache effects. For smaller matrix sizes (e.g., 250x250), the overhead of managing multiple threads may outweigh the benefits, leading to sublinear or even negative speedup. For larger sizes (1000x1000 and 2000x2000), parallelism is better utilized, and speedup is more significant.**

**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:**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <omp.h>**

**#define SIZE 200**

**#define SCALAR 10**

**void vector\_scalar\_add\_static(int chunk)**

**{**

**int A[SIZE], B[SIZE];**

**double start, end;**

**for (int i = 0; i < SIZE; i++)**

**A[i] = i;**

**start = omp\_get\_wtime();**

**#pragma omp parallel for schedule(static, chunk)**

**for (int i = 0; i < SIZE; i++)**

**B[i] = A[i] + SCALAR;**

**end = omp\_get\_wtime();**

**printf("[STATIC] Chunk: %d | Time: %.8f sec\n", chunk, end - start);**

**}**

**void vector\_scalar\_add\_dynamic(int chunk)**

**{**

**int A[SIZE], B[SIZE];**

**double start, end;**

**for (int i = 0; i < SIZE; i++)**

**A[i] = i;**

**start = omp\_get\_wtime();**

**#pragma omp parallel for schedule(dynamic, chunk)**

**for (int i = 0; i < SIZE; i++)**

**B[i] = A[i] + SCALAR;**

**end = omp\_get\_wtime();**

**printf("[DYNAMIC] Chunk: %d | Time: %.8f sec\n", chunk, end - start);**

**}**

**void demonstrate\_nowait\_clause()**

**{**

**int A[SIZE], B[SIZE], C[SIZE];**

**double start, end;**

**for (int i = 0; i < SIZE; i++)**

**A[i] = i;**

**start = omp\_get\_wtime();**

**#pragma omp parallel**

**{**

**#pragma omp for nowait**

**for (int i = 0; i < SIZE; i++)**

**B[i] = A[i] + SCALAR;**

**#pragma omp for**

**for (int i = 0; i < SIZE; i++)**

**C[i] = A[i] \* SCALAR;**

**}**

**end = omp\_get\_wtime();**

**printf("[NOWAIT] Parallel Sections Time: %.8f sec\n", end - start);**

**}**

**int main()**

**{**

**printf("---- STATIC SCHEDULE ----\n");**

**vector\_scalar\_add\_static(5);**

**vector\_scalar\_add\_static(10);**

**vector\_scalar\_add\_static(25);**

**vector\_scalar\_add\_static(50);**

**printf("\n---- DYNAMIC SCHEDULE ----\n");**

**vector\_scalar\_add\_dynamic(5);**

**vector\_scalar\_add\_dynamic(10);**

**vector\_scalar\_add\_dynamic(25);**

**vector\_scalar\_add\_dynamic(50);**

**printf("\n---- NOWAIT CLAUSE ----\n");**

**demonstrate\_nowait\_clause();**

**return 0;**

**}**

**OUTPUT:**

**A screen shot of a computer

AI-generated content may be incorrect.**

**Information and analysis:**

**Static scheduling showed better performance for uniform operations like scalar addition, especially with larger chunk sizes due to reduced overhead.  
Dynamic scheduling introduced more overhead, making it less efficient for equal workloads.  
Using the nowait clause allowed threads to proceed without waiting, improving overall parallel execution efficiency.**