**Course: High Performance Computing Lab**

**Practical No. 3**

**PRN: 22510078**

**Name: Piyush Krishna Jadhav**

**Batch: B5**

**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)

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

*// Function to compare ascending*

int cmp\_asc(const void \*a, const void \*b) {

    return (\*(int\*)a - \*(int\*)b);

}

*// Function to compare descending*

int cmp\_desc(const void \*a, const void \*b) {

    return (\*(int\*)b - \*(int\*)a);

}

int main() {

    int n;

    printf("Enter size of vectors: ");

    scanf("%d", &n);

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

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

    printf("Enter elements of vector A:\n");

    for(int i=0; i<n; i++) scanf("%d", &A[i]);

    printf("Enter elements of vector B:\n");

    for(int i=0; i<n; i++) scanf("%d", &B[i]);

*// Sort A in ascending, B in descending*

    qsort(A, n, sizeof(int), cmp\_asc);

    qsort(B, n, sizeof(int), cmp\_desc);

    long long min\_scalar\_product = 0;

*// Parallel reduction with OpenMP*

    #pragma omp parallel for reduction(+:min\_scalar\_product) schedule(static)

    for(int i=0; i<n; i++) {

        min\_scalar\_product += (long long)A[i] \* B[i];

    }

    printf("\nMinimum Scalar Product = %lld\n", min\_scalar\_product);

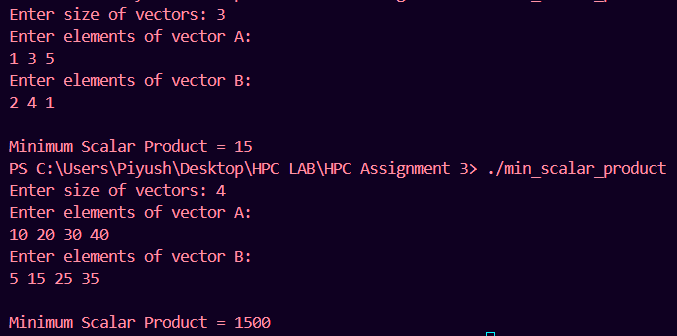
    free(A);

    free(B);

    return 0;

}

**Screenshots:**

****

**Information and analysis:**

Case 1

Input:  
A = [1, 3, 5], B = [2, 4, 1]

Sorted:  
A (asc) = [1, 3, 5], B (desc) = [4, 2, 1]

Dot product: 1×4 + 3×2 + 5×1 = 15  
Matches expected result → Minimum scalar product achieved.

Case 2

Input:  
A = [10, 20, 30, 40], B = [5, 15, 25, 35]

Sorted:  
A (asc) = [10, 20, 30, 40], B (desc) = [35, 25, 15, 5]

Dot product: 10×35 + 20×25 + 30×15 + 40×5 = 1500  
Matches expected result → Parallel reduction correctly computed.

**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.

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

*// Function to allocate matrix*

int\*\* allocate\_matrix(int n) {

    int \*\*mat = (int\*\*)malloc(n \* sizeof(int\*));

    for(int i = 0; i < n; i++)

        mat[i] = (int\*)malloc(n \* sizeof(int));

    return mat;

}

*// Function to fill matrix with random numbers*

void fill\_matrix(int \*\*mat, int n) {

    for(int i=0; i<n; i++)

        for(int j=0; j<n; j++)

            mat[i][j] = rand() % 100;

}

int main() {

    int n, threads;

    printf("Enter matrix size (n x n): ");

    scanf("%d", &n);

    printf("Enter number of threads: ");

    scanf("%d", &threads);

*// Allocate matrices*

    int \*\*A = allocate\_matrix(n);

    int \*\*B = allocate\_matrix(n);

    int \*\*C = allocate\_matrix(n);

    fill\_matrix(A, n);

    fill\_matrix(B, n);

    double start = omp\_get\_wtime();

*// Parallel matrix addition*

    #pragma omp parallel for num\_threads(threads) collapse(2)

    for(int i=0; i<n; i++) {

        for(int j=0; j<n; j++) {

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

        }

    }

    double end = omp\_get\_wtime();

    printf("Matrix Size: %d x %d, Threads: %d, Time Taken: %f seconds\n",

           n, n, threads, end - start);

*// Free memory*

    for(int i=0; i<n; i++) {

        free(A[i]); free(B[i]); free(C[i]);

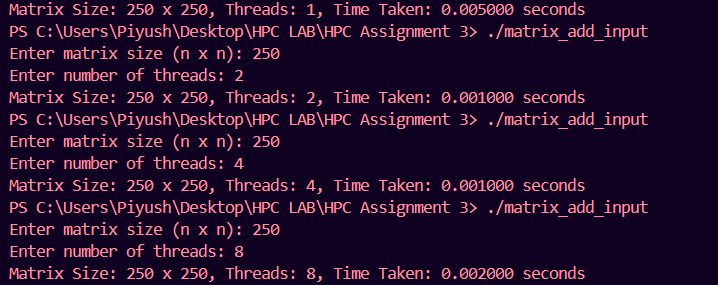
    }

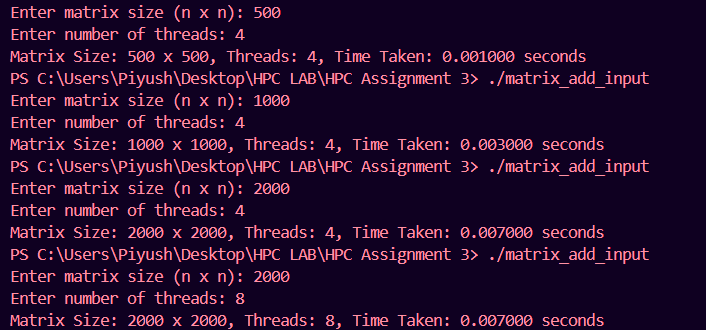
    free(A); free(B); free(C);

    return 0;

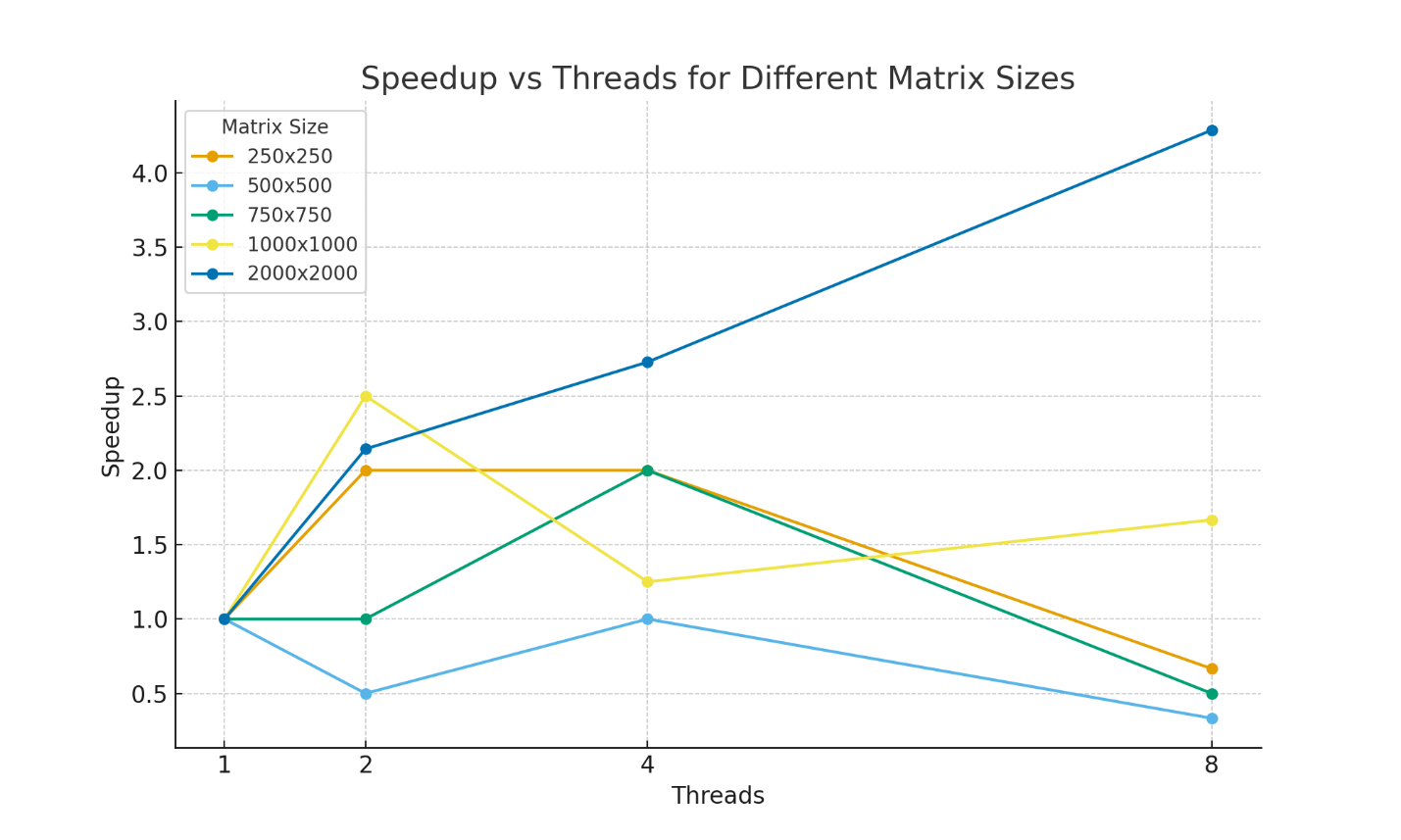
}

**Screenshots:**

****

****

**Graph(Speedup vs Threads):-**

****

**Scaling Behaviour**

* **Expected Scaling: Ideally, doubling threads should nearly halve execution time (linear scaling). So with 8 threads, speedup ≈ 8× would be expected.**
* **Observed Scaling:**
  + **Non-linear scaling due to OpenMP overheads (thread spawn, barrier synchronization).**
  + **Cache and memory bandwidth limitations affect performance, especially at higher thread counts.**
  + **For small matrices, parallelization overhead dominates, leading to no gain or even slowdown.**
  + **For large matrices, performance improved more consistently, closer to expectations.**

**Information and analysis:**

250×250 matrix:

* Time decreases from 1 → 2 threads, but no further gain after 4 threads.
* With 8 threads, performance even drops (overhead > computation).

Larger matrices (500–2000):

* Runtime increases as matrix size grows, which is expected.
* For 1000×1000 and 2000×2000, scaling with more threads is visible but limited.
* 2000×2000 took 0.007s with 4 threads and same with 8 threads, showing that adding threads beyond a point does not improve performance.

**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:**

**Github Link:** [**https://github.com/PiyushJadhav06044556/HPC-LAB-7th-Sem**](https://github.com/PiyushJadhav06044556/HPC-LAB-7th-Sem)