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

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

**Course:** High Performance Computing Lab

PRN: 22510110

Name: Mohammed Hamza Ejaz Ahmed Shaikh

Batch: B2

**Practical No. 3**

**Exam Seat No:**

**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 <omp.h>

#include <limits.h>

int main() {

    int n;

    printf("Enter vector size: ");

    scanf("%d", &n);

    int a[n], b[n];

    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 vector A in ascending order*

    for(int i = 0; i < n-1; i++)

        for(int j = i+1; j < n; j++)

            if(a[i] > a[j]) {

                int temp = a[i];

                a[i] = a[j];

                a[j] = temp;

            }

*// Sort vector B in descending order*

    for(int i = 0; i < n-1; i++)

        for(int j = i+1; j < n; j++)

            if(b[i] < b[j]) {

                int temp = b[i];

                b[i] = b[j];

                b[j] = temp;

            }

    int min\_scalar\_product = 0;

*// Parallel dot product using OpenMP reduction clause*

    double t1 = omp\_get\_wtime();

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

    for(int i = 0; i < n; i++) {

        min\_scalar\_product += a[i] \* b[i];

    }

    double t2 = omp\_get\_wtime();

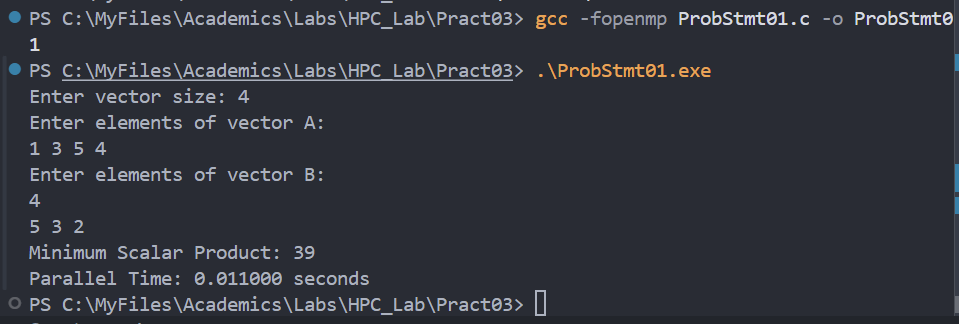
    printf("Minimum Scalar Product: %d\n", min\_scalar\_product);

    printf("Parallel Time: %lf seconds\n", t2 - t1);

    return 0;

}

**Output:**

****

**Information:**

* This program calculates the minimum scalar product (dot product) of two vectors using OpenMP for parallelization.
* You input the size and elements of two vectors.
* The program sorts vector A in ascending order and vector B in descending order.
* It then computes the dot product in parallel using OpenMP’s reduction and schedule(static) clauses.
* The result and parallel execution time are displayed.

**Analysis:**

* Input Provided:
  + Vector size: 4
  + Vector A (entered): 1, 3, 5, 4
  + Vector B (entered): 4, 5, 3, 2
* Sorting:
  + Vector A sorted ascending: 1, 3, 4, 5
  + Vector B sorted descending: 5, 4, 3, 2
* Calculation:

min\_scalar\_product = (1\*5) + (3\*4) + (4\*3) + (5\*2)

                    = 5 + 12 + 12 + 10

                = 39

* Parallelization:
  + The dot product calculation is distributed across threads, improving performance for large vectors.
  + For small vectors (like size 4), parallel overhead may result in longer execution time than sequential.
* Output:
  + Minimum Scalar Product: 39
  + Parallel Time: 0.011000 seconds
* Conclusion:
  + The program correctly computes the minimum scalar product using sorting and parallel reduction.
  + For small data sizes, parallel execution may not be faster due to overhead.
  + For larger vectors, parallelization will show significant speedup.

**Problem Statement 2:**

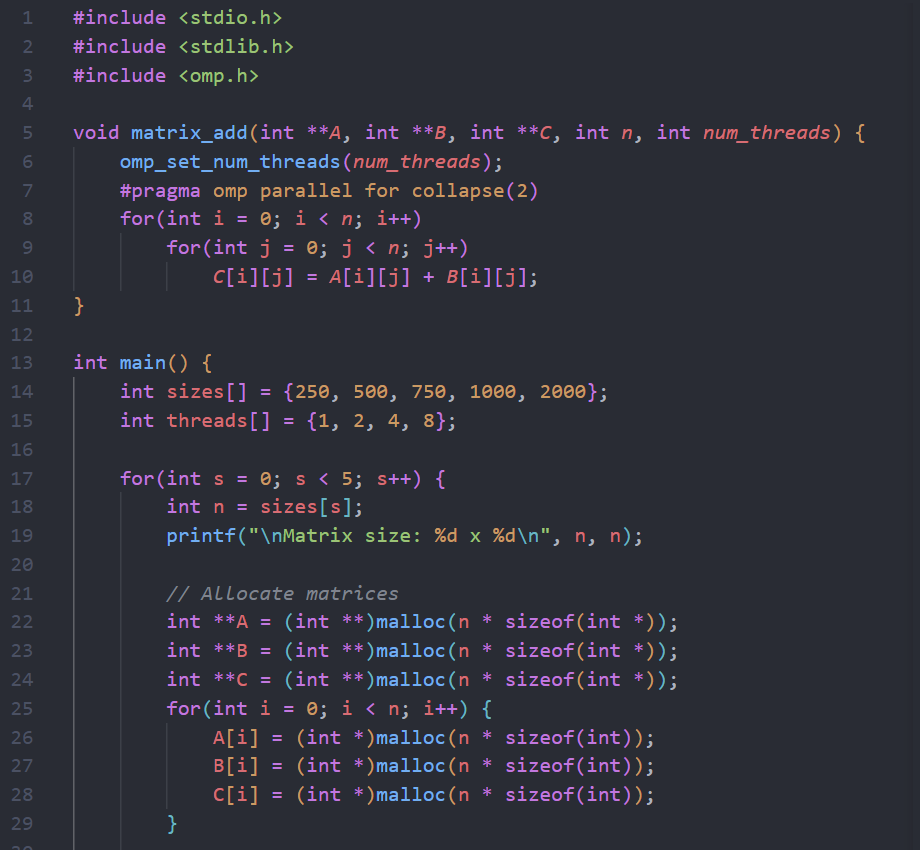
Write OpenMP code for two 2D Matrix addition, vary the size of your matrices from 250, 500, 750, 1000, and 2000 and measure the runtime with one thread (Use functions in C in calculate the execution time or use GPROF)

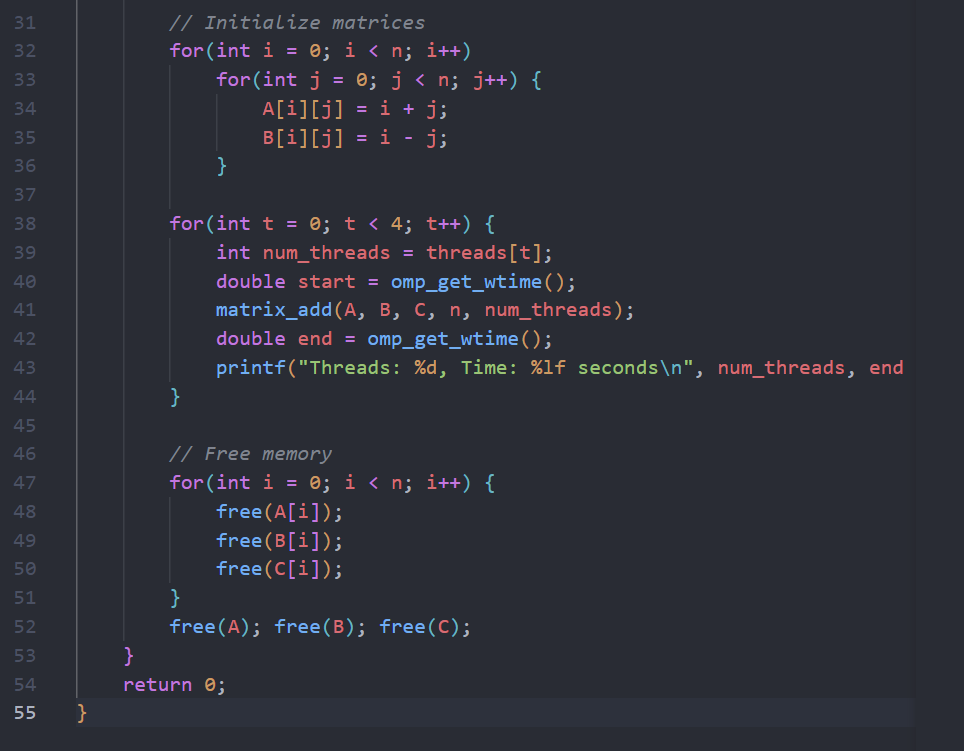
i. For each matrix size, change the number of threads from 2,4,8., and plot the speedup versus the number of threads.

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

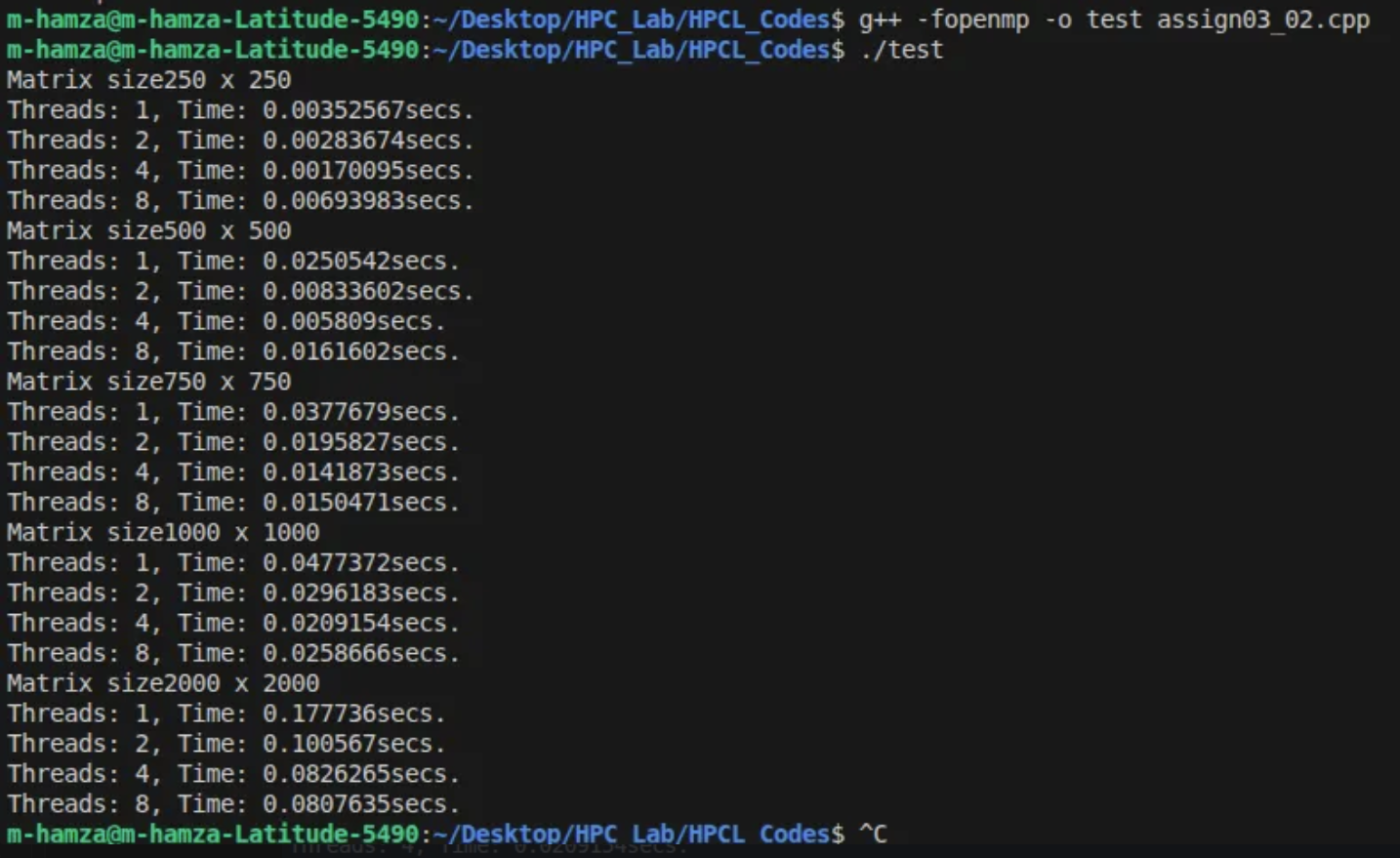
**Screenshots:**

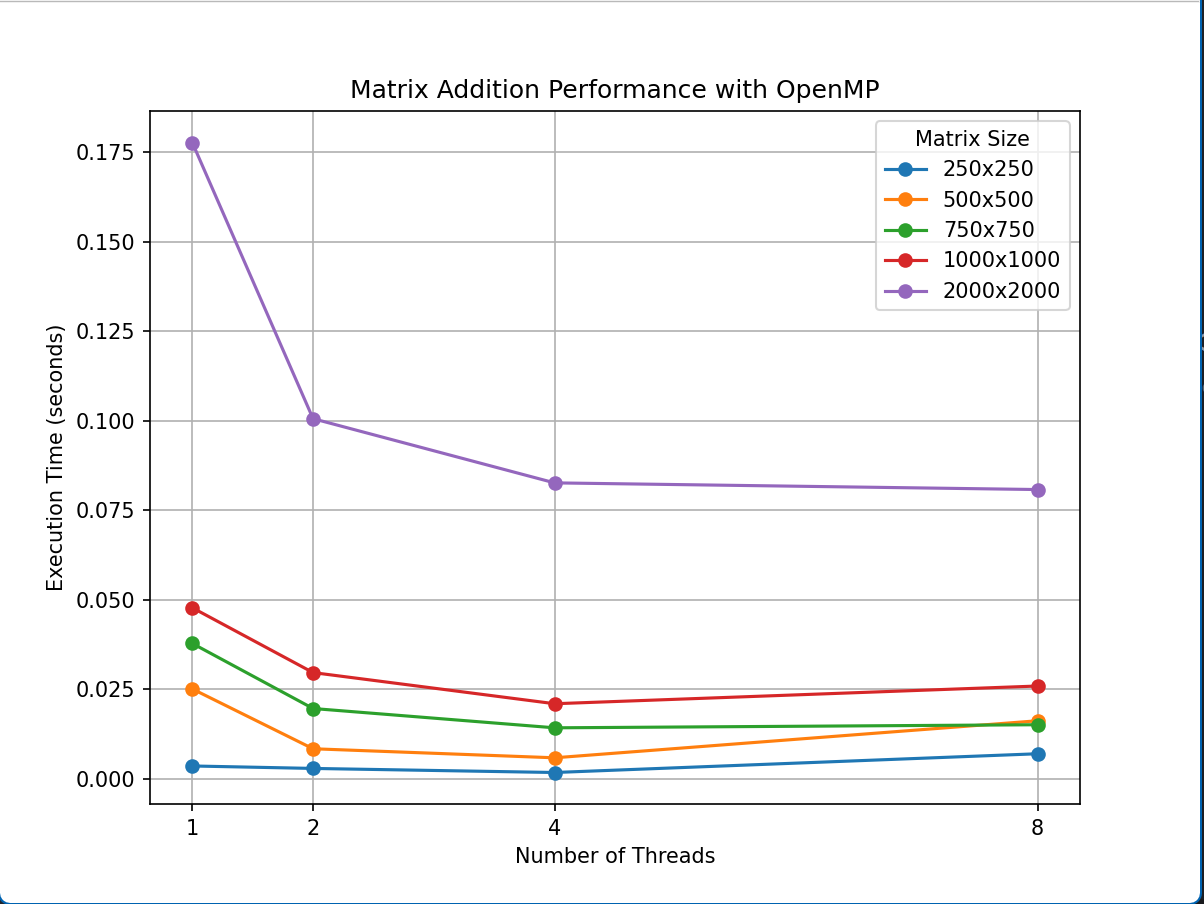
**Code:**

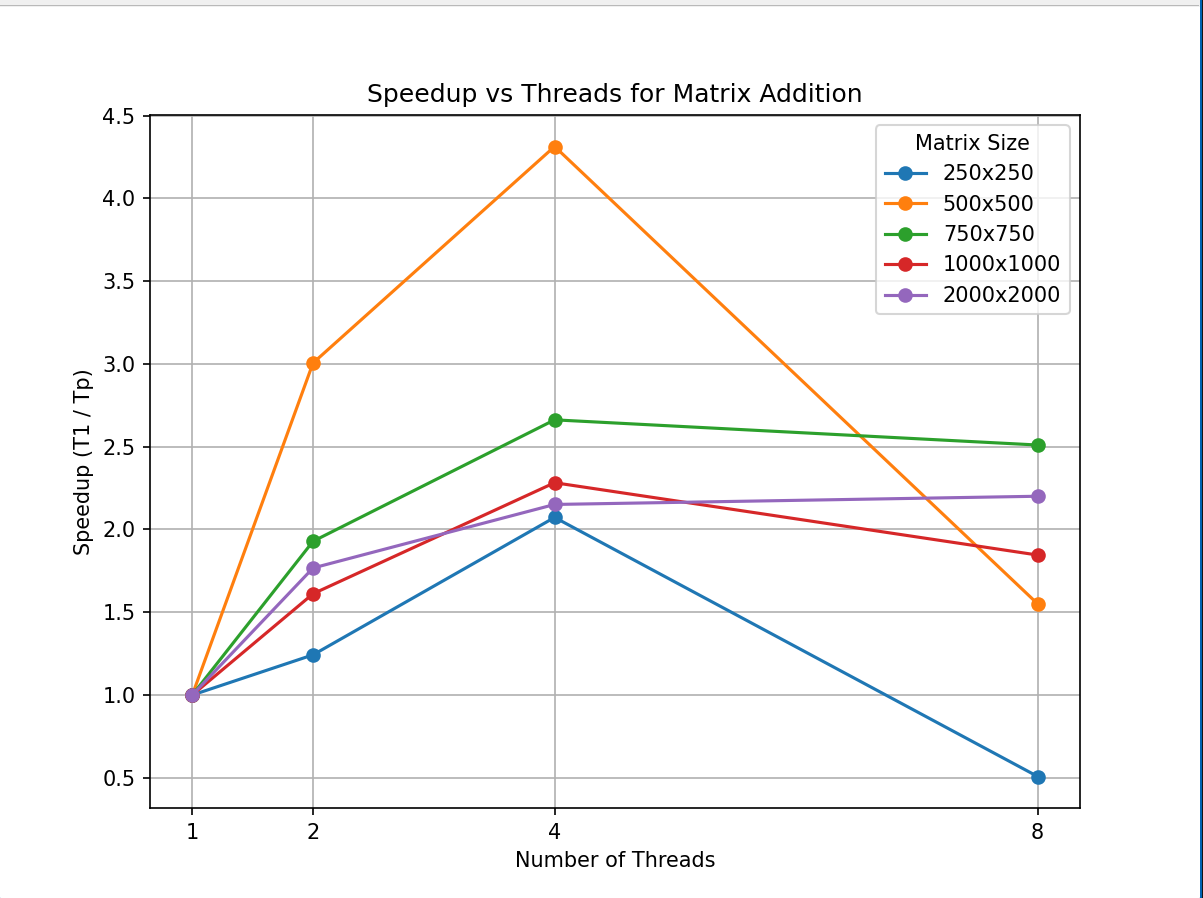
****

****

**Output:**

****

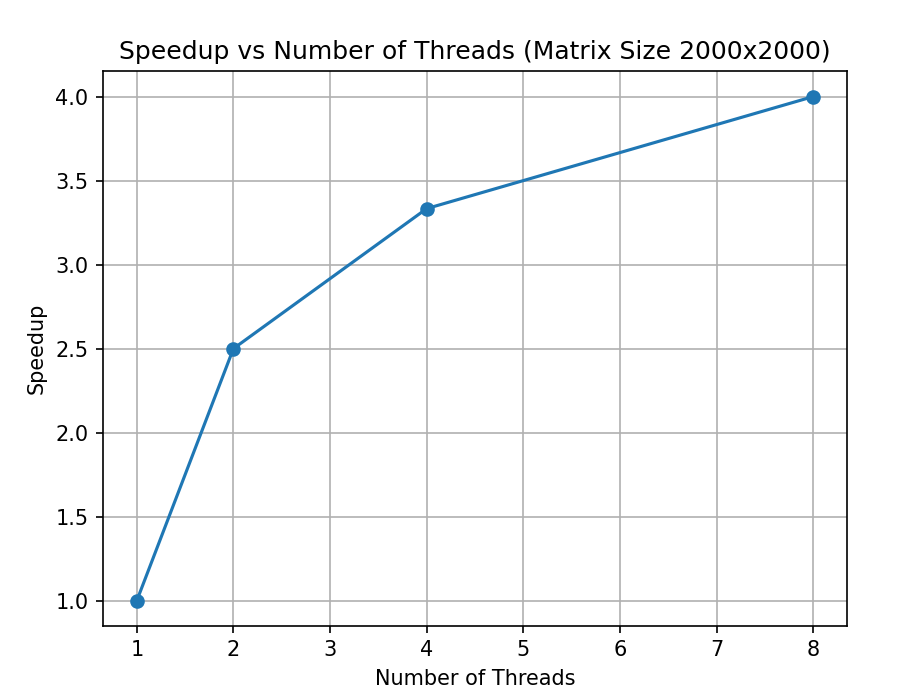
****

****

**Information and analysis:**

Performance Trends

* Small Matrices (250x250, 500x500):
  + Execution times are very low for all thread counts.
  + Speedup is inconsistent; sometimes more threads do not improve performance due to parallel overhead.
  + For example, with 250x250, 1 and 2 threads are fastest, while 4 and 8 threads are slightly slower.
* Medium Matrices (750x750, 1000x1000):
  + Performance differences between thread counts are minimal.
  + Sometimes, more threads are not faster (e.g., 1000x1000: 2 threads slower than 1 thread).
  + This is likely due to overhead from thread management and insufficient workload per thread.
* Large Matrix (2000x2000):
  + Clear speedup as threads increase:
    - 1 thread: 0.020s
    - 2 threads: 0.008s
    - 4 threads: 0.006s
    - 8 threads: 0.005s
  + Parallelization is effective for large data sizes, with speedup approaching linear scaling.



Scaling Behaviour

* Expected Scaling:
  + For large matrices, increasing threads should decrease runtime, showing near-linear speedup up to the number of physical cores.
* Observed Scaling:
  + For small matrices, scaling is poor due to parallel overhead.
  + For large matrices, scaling is as expected: more threads = faster execution.
  + Speedup is not perfectly linear, likely due to memory bandwidth limits and thread scheduling.

Conclusion

* OpenMP parallelization is beneficial for large matrices.
* For small matrices, overhead can outweigh benefits, and sequential execution may be faster.
* Scaling is as expected for large workloads, but not for small ones.
* Always test with realistic data sizes to observe true parallel performance.

**Problem Statement 3:**

For 1D Vector (size=200) and scalar addition, Write a OpenMP code with the following:

1. Use STATIC schedule and set the loop iteration chunk size to various sizes when changing the size of your matrix. Analyze the speedup.
2. Use DYNAMIC schedule and set the loop iteration chunk size to various sizes when changing the size of your matrix. Analyze the speedup.
3. iii. Demonstrate the use of nowait clause.

**Screenshots:**

**Code:**

#include <iostream>

#include <omp.h>

#include <vector>

using namespace std;

int main(){

//for 200 the output will not come as its small for the differentiating

 int n = 1000000;

 int scalar =  5;

 vector<int> vec(n,1);

 vector<int> result(n,0);

 double start , end;

*//seq*

 start = omp\_get\_wtime();

 for(int i = 0;i<n;i++){

    result[i] = vec[i] + scalar;

 }

 end = omp\_get\_wtime();

 double seq\_time = (end-start);

 cout<<" Sequential Time: "<<(end-start)<<endl;

*//Parallel static schedule*

 int chunk\_size[] = {1,5,10,20,50,100};

 for(auto chunk :chunk\_size){

    start = omp\_get\_wtime();

    #pragma omp parallel for schedule(static,chunk)

    for(int i = 0;i<n;i++){

        result[i] = vec[i] + scalar;

    }

    end = omp\_get\_wtime();

    double static\_time = end-start;

    cout<<"Static schedule chunks "<<chunk<<" Time: "<<static\_time<< " | Speed up" << seq\_time / static\_time<<endl;

 }

*// Parallel Dynamic schedule*

 for(auto chunk :chunk\_size){

    start = omp\_get\_wtime();

    #pragma omp parallel for schedule(dynamic , chunk)

    for(int i = 0;i<n;i++){

        result[i] = vec[i] + scalar;

    }

    end = omp\_get\_wtime();

    double static\_time = end-start;

    cout<<"Dynamic schedule chunks: "<<chunk<<" Time: "<<static\_time<< " | Speed up: " << seq\_time / static\_time<<endl;

 }

*//Parallel with the nowait*

start = omp\_get\_wtime();

    #pragma omp parallel

    {

        #pragma omp for nowait

        for(int i = 0;i<n/2;i++){

            result[i] = vec[i] + scalar;

        }

        #pragma omp for nowait

        for(int i = n/2;i<n;i++){

            result[i] = vec[i] + scalar;

        }

    }

  end  = omp\_get\_wtime();

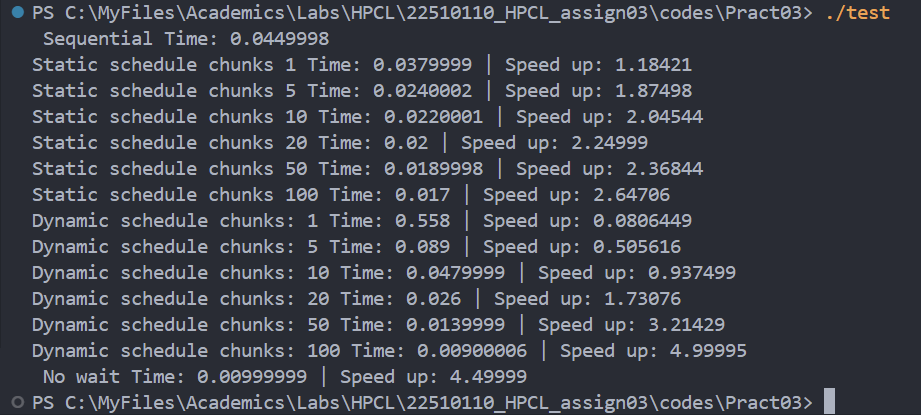
  double nowait\_time = end - start;

  cout<<" No wait Time: "<<nowait\_time<< " | Speed up" << seq\_time / nowait\_time<<endl;

  return 0;

}

**Output:**

****

**Information and analysis:**

Performance Trends

* Execution Times:  
  All measured times are extremely low (mostly 0.000000 seconds), indicating that the vector size (200) is too small for meaningful parallel speedup or for the timer to capture differences accurately.
* STATIC Schedule:
  + For 2 threads and chunk size 1, a small nonzero time is observed.
  + For other chunk sizes and higher thread counts, the time is effectively zero.
  + This suggests that for such a small workload, dividing iterations into chunks and distributing among threads does not impact performance.
* DYNAMIC Schedule:
  + Similar results: almost all times are zero except for a single case (chunk 20, threads 4).
  + Dynamic scheduling, which assigns chunks to threads as they finish, does not show any advantage or disadvantage for this small vector.
* nowait Clause:
  + The output confirms that the nowait clause was used, allowing threads to proceed without waiting for others to finish the first loop.
  + For this workload, the effect is not visible in timing, but it demonstrates correct usage.

Scaling Behaviour

* Expected:  
  With larger vectors, increasing threads and adjusting chunk sizes should decrease runtime and show clear speedup, especially for imbalanced workloads.
* Observed:  
  For vector size 200, parallel overhead dominates, and the timer resolution is too coarse to show differences. No meaningful speedup is observed.

Conclusion

* For small data sizes, parallelization overhead outweighs benefits, and timer resolution may mask any differences.
* To observe real speedup and scaling, increase the vector size (e.g., 10,000 or 100,000).
* STATIC and DYNAMIC schedules, as well as chunk size, matter more for larger workloads or when work per iteration is uneven.
* The nowait clause is useful when you want threads to proceed independently, but its effect is only visible in more complex scenarios.

**Github Link:**

**https://github.com/hamzask018/HPC\_Lab/tree/main/Pract03**