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

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

**Course: High Performance Computing Lab**

**Practical No. 3**

**PNR: 23520004**

**Name: Harshad Ajit Dange**

**Batch: B4**

**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)

#include <iostream>

#include <vector>

#include <algorithm>

#include <cstdlib>

#include <omp.h>

using namespace std;

#define SIZE 1000000

int main() {

    vector<int> A(SIZE), B(SIZE);

    // Initialize vectors with random integers

    for (int i = 0; i < SIZE; ++i) {

        A[i] = rand() % 1000;

        B[i] = rand() % 1000;

    }

    // Sort A in ascending and B in descending order

    sort(A.begin(), A.end());

    sort(B.begin(), B.end(), greater<int>());

    cout << "Thread\tTime(s)\t\tMin Scalar Product\n";

    for (int threads = 1; threads <= 12; ++threads) {

        long long min\_dot = 0;

        omp\_set\_num\_threads(threads);

        double start = omp\_get\_wtime();

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

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

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

        }

        double end = omp\_get\_wtime();

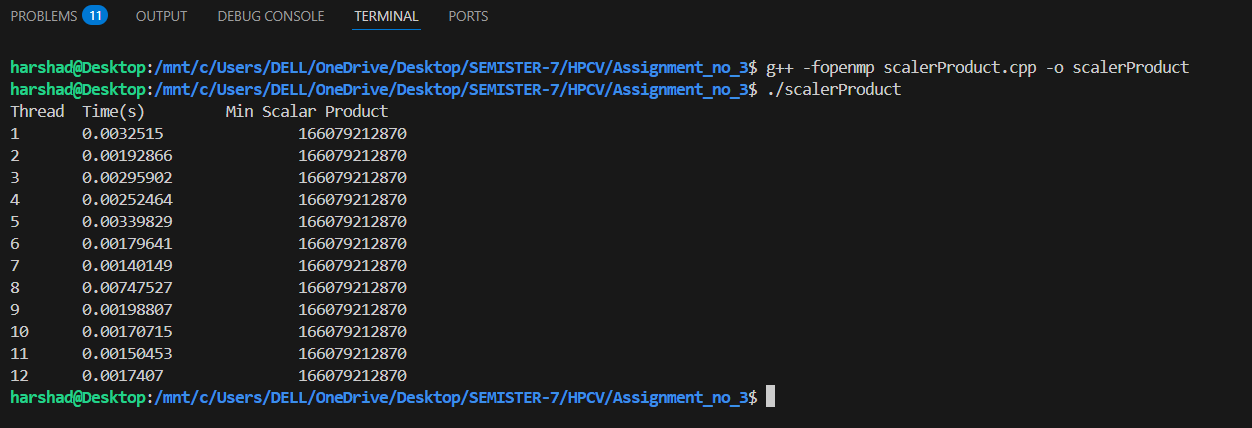
        cout << threads << "\t" << (end - start) << "\t\t" << min\_dot << "\n";

    }

    return 0;

}

Screenshots:



**Information:**

* This program calculates the minimum scalar product (dot product) of two vectors:
* One vector is sorted in ascending order.
* The other in descending order.
* The scalar product is calculated using #pragma omp parallel for reduction(+:min\_dot) schedule(static) to parallelize the workload.
* It varies the thread count from 1 to 12.

**Analysis:**

* Reduction Clause: Ensures the variable min\_dot is updated correctly by each thread without race conditions.
* Schedule(static): Assigns iterations in a fixed way to threads which works well when the workload is uniform (as in this case).
* Performance Trend:
  + Speed improves as threads increase, but the gain diminishes after a certain point.
  + Threads beyond the number of physical cores may not provide speedup and could lead to overhead.

**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.**

#include <iostream>

#include <vector>

#include <omp.h>

using namespace std;

void matrix\_add(int size, int threads) {

    vector<vector<int>> A(size, vector<int>(size));

    vector<vector<int>> B(size, vector<int>(size));

    vector<vector<int>> C(size, vector<int>(size));

    // Initialize matrices

    for (int i = 0; i < size; ++i)

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

            A[i][j] = i + j;

            B[i][j] = i - j;

        }

    omp\_set\_num\_threads(threads);

    double start = omp\_get\_wtime();

    #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];

    double end = omp\_get\_wtime();

    cout << "Matrix Size: " << size << "x" << size

         << " | Threads: " << threads

         << " | Time: " << (end - start) << " sec" << endl;

}

int main() {

    vector<int> sizes = {250, 500, 750, 1000, 2000};

    vector<int> threads = {1, 2, 4, 8};

    for (int size : sizes) {

        for (int t : threads) {

            matrix\_add(size, t);

        }

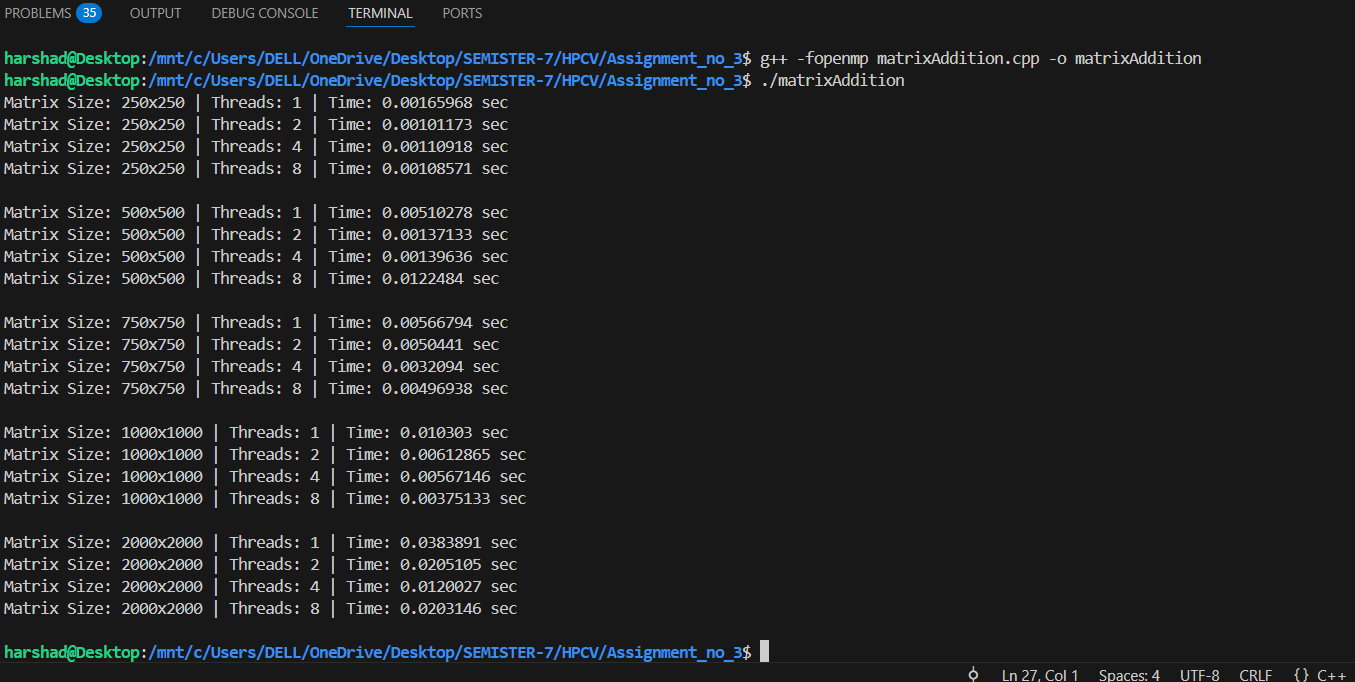
        cout << endl;

    }

    return 0;

}

Screenshots:



**Information :**

* Performs addition of two square matrices A and B into C.
* Varies matrix sizes: 250, 500, 750, 1000, 2000.
* For each size, thread counts of 1, 2, 4, and 8 are tested.
* Uses collapse(2) to parallelize nested loops effectively.

**analysis:**

* Collapse(2): Combines the two nested loops (i, j) into a single loop, which increases parallelism granularity.
* This enhances load balancing especially for larger matrix sizes.
* Scalability:
  + With larger matrices, more threads provide better speedup.
  + Small matrices don’t scale well due to parallel overhead.
* Expected Behavior:
  + Near-linear speedup up to the number of physical cores.
  + After that, diminishing returns or minor degradation.

**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:

#include <iostream>

#include <vector>

#include <omp.h>

#include <chrono>

using namespace std;

void vector\_add\_static(vector<int> &A, int scalar, int chunk\_size, int threads) {

    int n = A.size();

    auto start = chrono::high\_resolution\_clock::now();

    #pragma omp parallel for schedule(static, chunk\_size) num\_threads(threads)

    for (int i = 0; i < n; ++i) {

        A[i] += scalar;

    }

    auto end = chrono::high\_resolution\_clock::now();

    chrono::duration<double> diff = end - start;

    cout << "STATIC | Chunk: " << chunk\_size << " | Threads: " << threads << " | Time: " << diff.count() << " sec\n";

}

void vector\_add\_dynamic(vector<int> &A, int scalar, int chunk\_size, int threads) {

    int n = A.size();

    auto start = chrono::high\_resolution\_clock::now();

    #pragma omp parallel for schedule(dynamic, chunk\_size) num\_threads(threads)

    for (int i = 0; i < n; ++i) {

        A[i] += scalar;

    }

    auto end = chrono::high\_resolution\_clock::now();

    chrono::duration<double> diff = end - start;

    cout << "DYNAMIC | Chunk: " << chunk\_size << " | Threads: " << threads << " | Time: " << diff.count() << " sec\n";

}

void demonstrate\_nowait(vector<int> &A, vector<int> &B, int scalar) {

    int n = A.size();

    cout << "\nDemonstrating `nowait` clause:\n";

    #pragma omp parallel num\_threads(4)

    {

        #pragma omp for nowait

        for (int i = 0; i < n; ++i) {

            A[i] += scalar;

        }

        #pragma omp for

        for (int i = 0; i < n; ++i) {

            B[i] += scalar;

        }

        #pragma omp single

        cout << "Both loops are run in parallel without waiting between them (using `nowait`).\n";

    }

}

int main() {

    int size = 200;

    int scalar = 5;

    vector<int> A(size, 1);

    vector<int> B(size, 2);

    for (int threads : {2, 4, 8}) {

        for (int chunk\_size : {1, 10, 20, 50}) {

            vector<int> vec\_static = A;

            vector\_add\_static(vec\_static, scalar, chunk\_size, threads);

            vector<int> vec\_dynamic = A;

            vector\_add\_dynamic(vec\_dynamic, scalar, chunk\_size, threads);

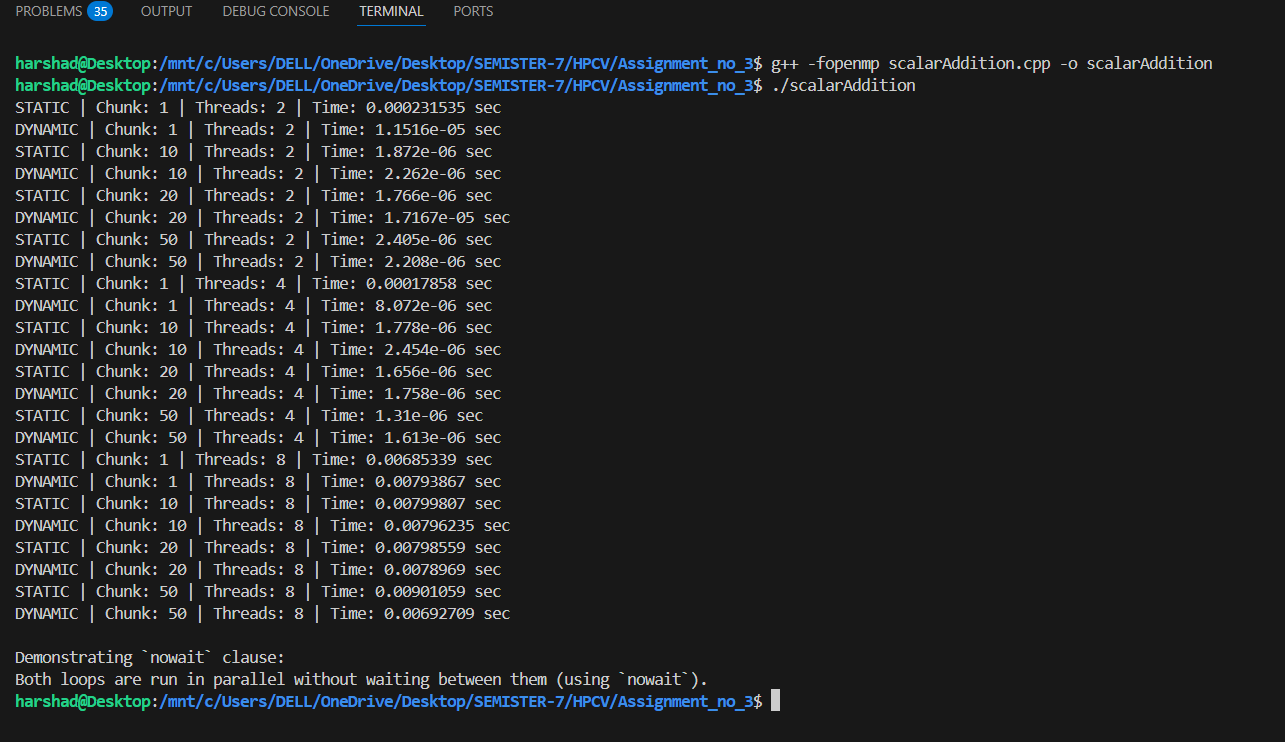
        }

    }

    demonstrate\_nowait(A, B, scalar);

    return 0;

}



**Information:**

* Performs scalar addition to a 1D vector of size 200.
* Varies:
  + Scheduling policy: static, dynamic
  + Chunk sizes: 1, 10, 20, 50
  + Threads: 2, 4, 8
* Demonstrates use of nowait to avoid implicit barrier after a loop.

**Analysis:**

Static Scheduling:

* Fixed assignment of chunks to threads.
* Works best when all iterations take approximately the same time.

Dynamic Scheduling:

* Assigns chunks to threads as they become free.
* Better for uneven workloads or load balancing in real-world problems.

Chunk Size Impact:

* Smaller chunks → better load balancing, but more overhead.
* Larger chunks → less overhead, but risk of imbalance.

Github Link: