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

**Year:** 2025-26 **Semester:** Odd

**Course:** Cutting Edge Technologies Lab

**Course code:** 7CS352

**Practical No. 6**

**Exam Seat No:** 23610067

**Title of practical:**

Study and Implementation of Reduction operations and nested loop parallelism using collapse.

1. Matrix Matrix Multiplication
2. Vector Dot Product

**Problem Statement 1:**

Matrix Matrix Multiplication

**Screenshots:**

**// Matrix Matrix Multiplication**

**#include <iostream>**

**#include <omp.h>**

**#include <vector>**

**#include <iomanip>**

**using namespace std;**

**int main(){**

**int m, n;**

**cout <<"Enter number of rows and columns ";**

**cin >> n ;**

**vector<vector<int>> A(n, vector<int>(n));**

**vector<vector<int>> B(n, vector<int>(n));**

**vector<vector<int>> C(n, vector<int>(n, 0));**

**double start\_seq, start\_parallel, start\_collapse, end\_seq, end\_parallel, end\_collapse;**

**for (int i = 0; i<n ;i ++){**

**for (int j = 0; j<n; j++){**

**A[i][j] = i\*3.0 + j\*2.0;**

**B[i][j] = i\*2.0 + j\*1.5;**

**}**

**}**

**cout<<"-----------------------Sequential----------------------"<< endl;**

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

**for (int i = 0; i<n ; i++){**

**for(int j = 0; j<n; j++){**

**C[i][j] = 0;**

**for(int k = 0; k<n; k++ ){**

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

**}**

**}**

**}**

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

**cout<<"-----------------------Parallel----------------------" << endl;**

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

**#pragma omp parallel for**

**for (int i = 0; i<n ; i++){**

**for(int j = 0; j<n; j++){**

**C[i][j] = 0;**

**for(int k = 0; k<n; k++ ){**

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

**}**

**}**

**}**

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

**cout<<"-----------------------Collapse----------------------" << endl;**

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

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

**for (int i = 0; i<n ; i++){**

**for(int j = 0; j<n; j++){**

**C[i][j] = 0;**

**for(int k = 0; k<n; k++ ){**

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

**}**

**}**

**}**

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

**for (int i = 0; i<n ; i++){**

**for(int j = 0; j<n; j++){**

**cout << C[i][j] <<" ";**

**}**

**cout<<endl;**

**}**

**cout << fixed << setprecision(9);**

**double seq\_time = (end\_seq - start\_seq);**

**double parallel\_time = (end\_parallel - start\_parallel);**

**double collapse\_time = (end\_collapse - start\_collapse);**

**cout << "Sequential Time: " << seq\_time << " seconds\n";**

**cout << "Parallel Time: " << parallel\_time << " seconds\n";**

**cout << "Collapse Time: " << collapse\_time << " seconds\n";**

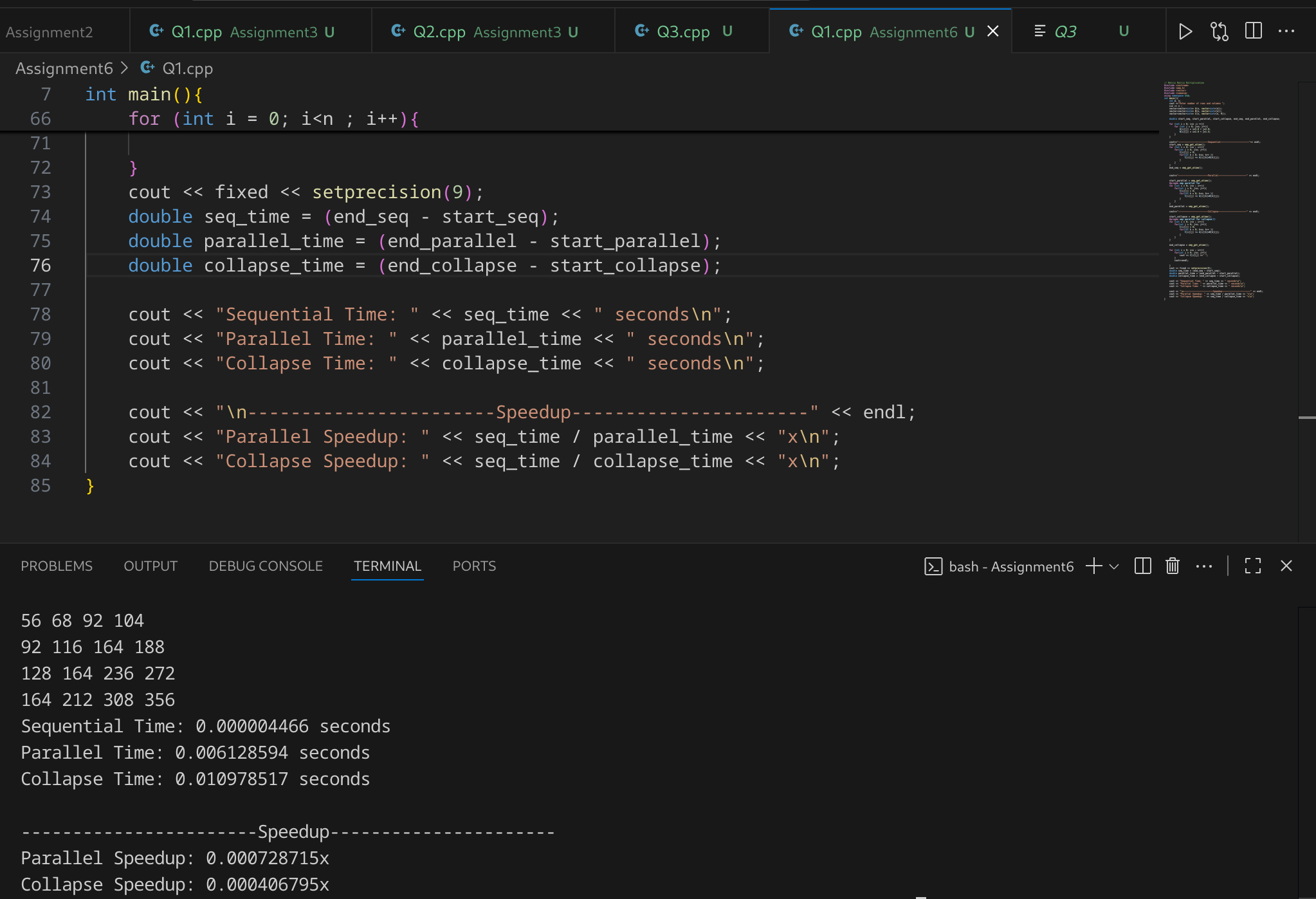
**cout << "\n-----------------------Speedup----------------------" << endl;**

**cout << "Parallel Speedup: " << seq\_time / parallel\_time << "x\n";**

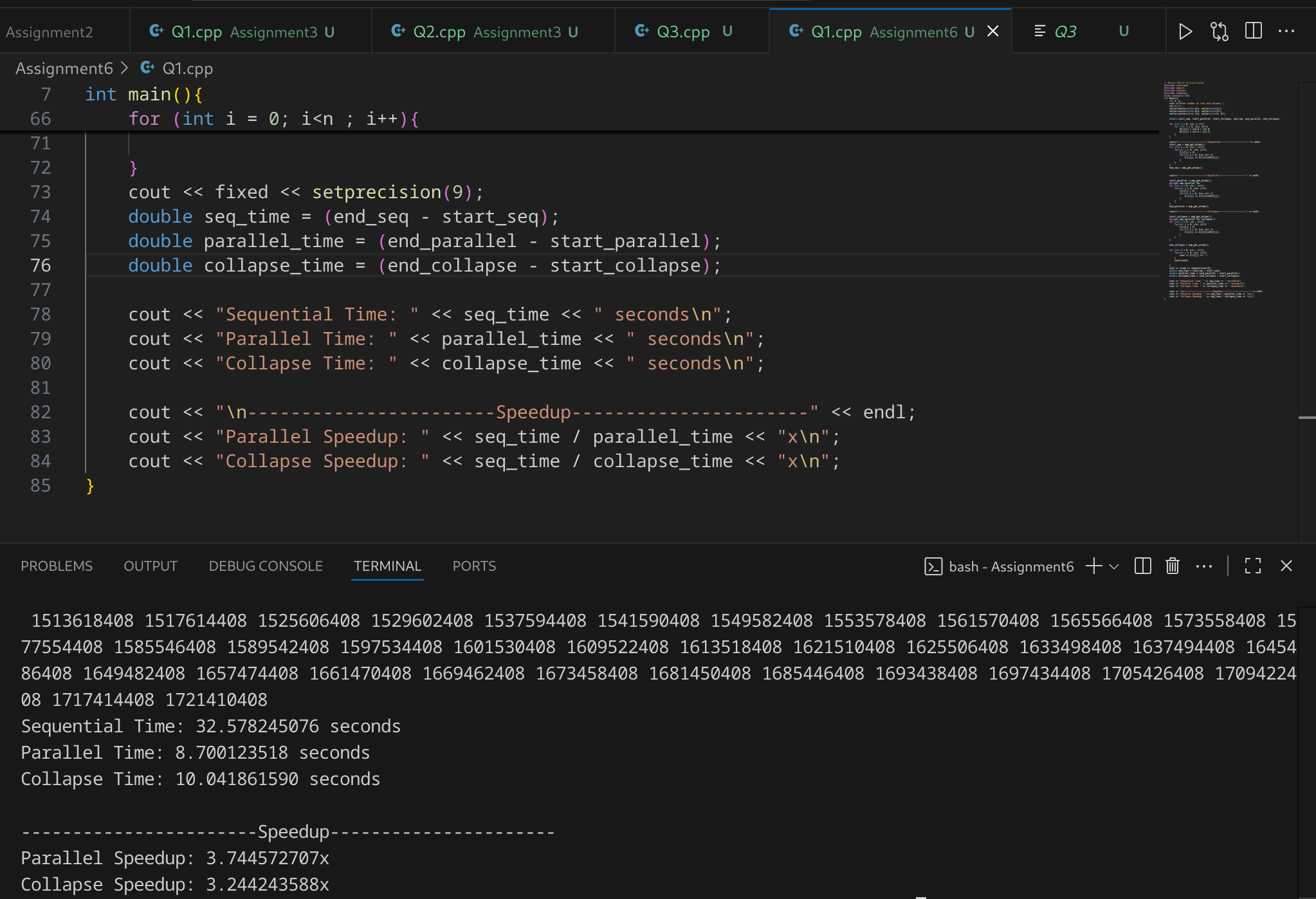
**cout << "Collapse Speedup: " << seq\_time / collapse\_time << "x\n";**

**}**

**for small n:** n = 4

****

**for greater n:** n = 1000

****

**Information and analysis: Time analysis, Speedup analysis for all programs**

#### Sequential Execution

* Time complexity: **O(n³)** (three nested loops).
* For n = 4, time is negligible (few operations).
* For n = 1000, time becomes very large (billions of operations).
* Acts as the baseline for speedup.

#### Parallel Execution (Outer Loop Parallelism)

* Only the **outer loop (rows)** is parallelized.
* Each thread handles one or more rows of the result matrix.
* Advantage: **cache reuse** – once a row of A is loaded, it is reused across multiple columns of B.
* Speedup improves significantly for large n because:  
  + Work per thread is large.
  + Synchronization overhead is small compared to computation.

**Speedup Trend:**

* For small n, parallel overhead > computation → speedup is low.
* For large n (like 1000), threads work efficiently → near-linear speedup depending on number of cores.

#### Parallel Execution with Collapse(2)

* Collapses both i and j loops into one iteration space.
* Each thread works on scattered (i, j) pairs instead of entire rows.
* Issue: **cache locality is reduced** → more cache misses.
* Extra scheduling overhead in distributing (i, j) pairs.

**Speedup Trend:**

* For small n, collapse sometimes performs equally or slightly better.
* For large n, collapse is slower than outer loop parallelism because memory access pattern dominates performance.

**Problem Statement 2:**

Vector dot product

**Screenshots:**

**#include <iostream>**

**#include <vector>**

**#include <omp.h>**

**using namespace std;**

**int main() {**

**int n;**

**cout << "Enter size of vectors: ";**

**cin >> n;**

**vector<double> A(n), B(n);**

**for (int i = 0; i < n; i++) {**

**A[i] = i \* 1.5;**

**B[i] = i \* 2.0;**

**}**

**cout << "------------Without reduction ----------------------"<< endl;**

**double sum = 0;**

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

**#pragma omp parallel for**

**for (int i = 0; i < n; i++) {**

**sum += A[i] \* B[i];**

**}**

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

**cout << "Dot Product = " << sum << endl;**

**cout << "Time Taken = " << (end - start) << " seconds\n";**

**sum = 0;**

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

**#pragma omp parallel for reduction(+:sum)**

**for (int i = 0; i < n; i++) {**

**sum += A[i] \* B[i];**

**}**

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

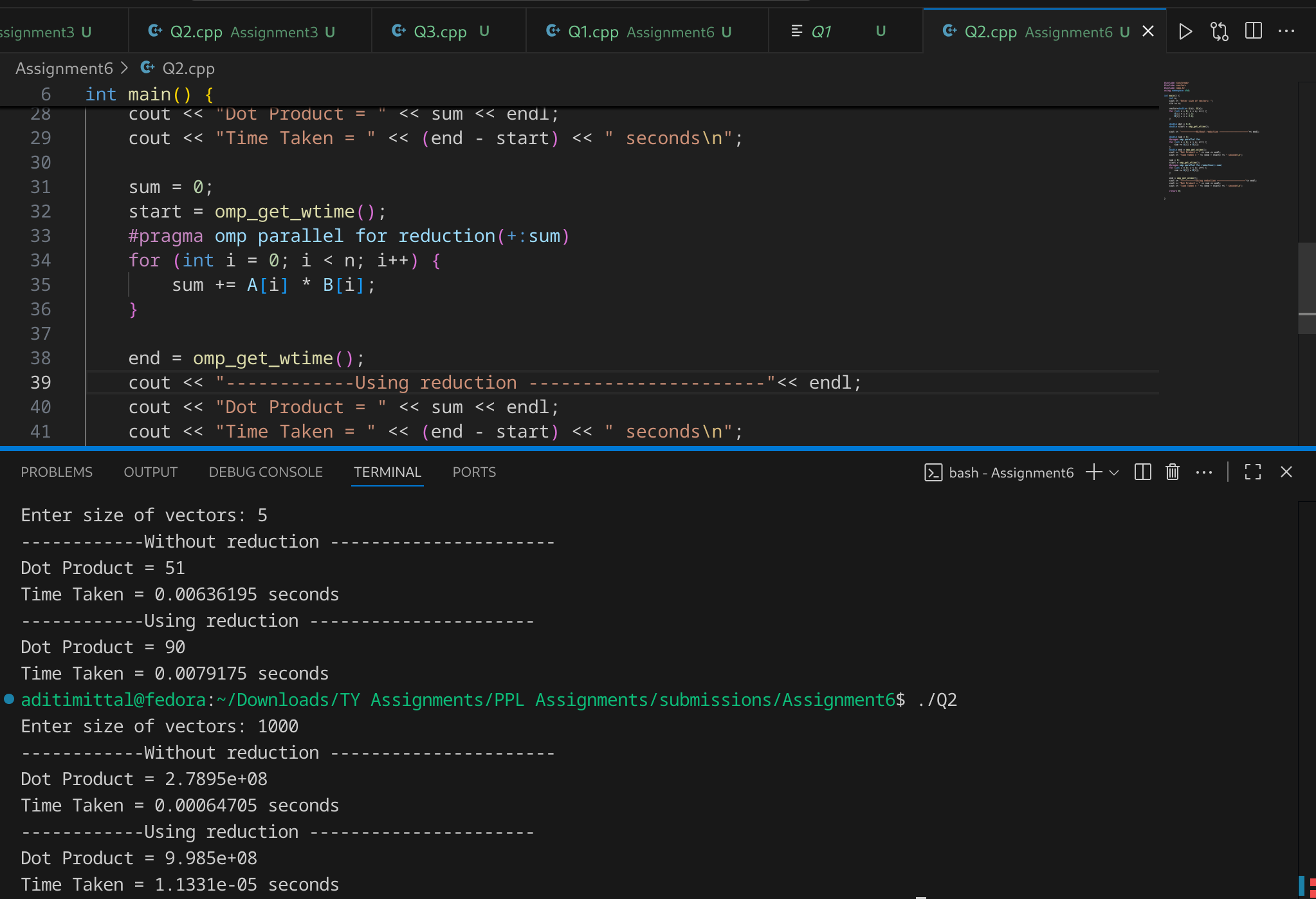
**cout << "------------Using reduction ----------------------"<< endl;**

**cout << "Dot Product = " << sum << endl;**

**cout << "Time Taken = " << (end - start) << " seconds\n";**

**return 0;**

**}**

****

**Information and analysis:**

**Without Reduction**

* If you just write sum += A[i]\*B[i]; inside a parallel loop:  
  + **Race condition occurs** (multiple threads update sum at the same time).
  + Result may be wrong or inconsistent.

**With Reduction(+:sum)**

* OpenMP automatically creates **private copies** of sum for each thread.
* Each thread computes its local partial sum.
* At the end, OpenMP combines (reduces) all partial sums into the final result.
* Eliminates race conditions and ensures correctness.

**Performance Observation**

* Reduction adds **slight overhead**, but ensures correctness.
* Still much faster than sequential for large vectors.

**Github Link:**

<https://github.com/aditimittal38/Parallel-Programming-Lab.git>