**Class:** Third Year (Computer Science and Engineering)

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

**Course:** Cutting Edge Technologies Lab

**Course code:** 7CS352

**Practical No. 5**

**Exam Seat No:** 23610067

**Title of practical: Study and implementation of Synchronization constructs: critical, barrier, atomic, ordered, nowait.**

1. Write an OpenMP program where multiple threads increment a shared counter inside a loop.
   * Implement once without synchronization (observe wrong result).
   * Implement again using #pragma omp critical (correct result).
2. Write a program to compute the sum of an array of N=10^6 numbers in parallel.
   * First, use #pragma omp atomic to update the shared sum.
   * Compare runtime with #pragma omp critical.
3. Write a program with two parallel regions:
   * In the first part, each thread computes the square of its thread ID and stores it in an array.
   * Use a #pragma omp barrier to ensure all threads finish before the master thread prints the array.
4. Write a program to print numbers from 1 to 20 in ascending order using a parallel for loop.
   * Use #pragma omp ordered to maintain order.
   * Run the same code without ordered and compare the output.
5. Write a program with two parallel for loops inside the same parallel region:
   * First loop initializes an array.
   * Second loop immediately prints the array elements.
   * Use #pragma omp for nowait on the first loop and observe what happens (incorrect output).
   * Then remove nowait to fix the issue.
6. Write a program to calculate factorial of n in parallel using OpenMP.
   * Use critical or atomic for shared multiplication.
   * Then optimize using reduction.
7. Sum of squares of first 1 million numbers.

**Problem Statement 1:**

Write an OpenMP program where multiple threads increment a shared counter inside a loop.

* + Implement once without synchronization (observe wrong result).
  + Implement again using #pragma omp critical (correct result).

**Screenshots:**

**// Write an OpenMP program where multiple threads increment a shared counter inside a loop.**

**// Implement once without synchronization (observe wrong result).**

**// Implement again using #pragma omp critical (correct result).**

**#include <iostream>**

**#include <omp.h>**

**using namespace std;**

**int main(){**

**int counter = 0;**

**#pragma omp parallel**

**{**

**cout << counter ++ << "from thread: " << omp\_get\_thread\_num()<< endl;**

**}**

**cout << "Final Value: "<< counter<< endl;**

**cout << "------------------With critical--------------";**

**counter = 0;**

**#pragma omp parallel**

**{**

**#pragma omp critical**

**{**

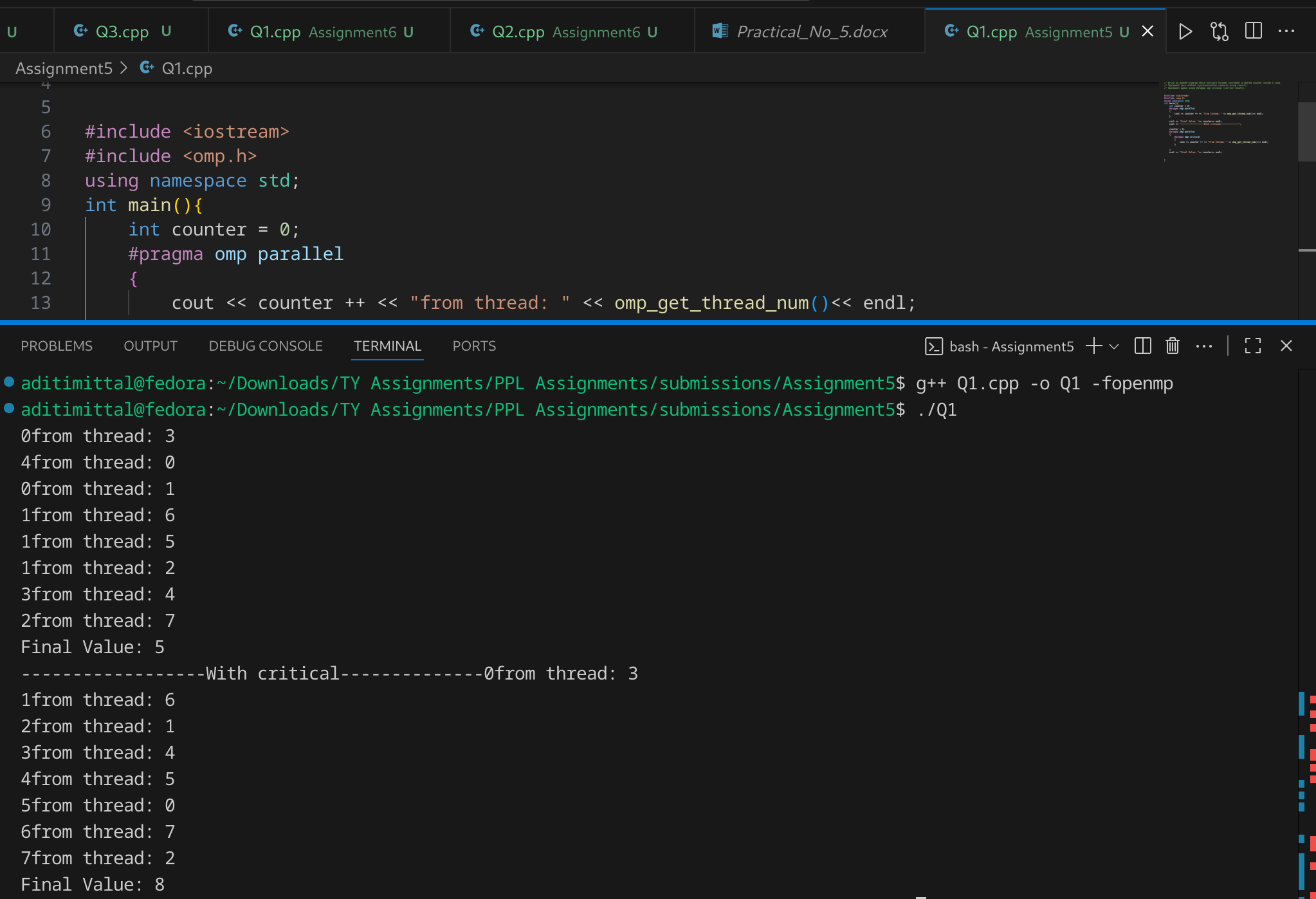
**cout << counter ++ << "from thread: " << omp\_get\_thread\_num()<< endl;**

**}**

**}**

**cout << "Final Value: "<< counter<< endl;**

**}**

****

### Information & Analysis:

* **Without Synchronization:** Threads update counter concurrently. Data races occur, so increments overlap and the final counter < number of threads. Wrong output.
* **With critical:** Each update is serialized inside a critical section. Threads increment one at a time → correct result but slower due to locking overhead.

**Problem Statement 2:**

Write a program to compute the sum of an array of N=10^6 numbers in parallel.

* + First, use #pragma omp atomic to update the shared sum.
  + Compare runtime with #pragma omp critical.

**Screenshots:**

**// Write a program to compute the sum of an array of N=10^6 numbers in parallel.**

**// First, use #pragma omp atomic to update the shared sum.**

**// Compare runtime with #pragma omp critical.**

**#include <iostream>**

**#include <omp.h>**

**#include <vector>**

**using namespace std;**

**int main(){**

**int n = 1000000;**

**vector<double> arr(n);**

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

**arr[i] = i\*1.26;**

**}**

**double sum = 0;**

**cout << "------------------Without critical and atomic--------------"<< endl;**

**#pragma omp parallel for**

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

**sum += arr[i];**

**}**

**cout << "Final Value: "<< sum << endl;**

**sum = 0;**

**cout << "------------------With atomic--------------"<< endl;**

**#pragma omp parallel for**

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

**#pragma omp atomic**

**sum += arr[i];**

**}**

**cout << "Final Value: "<< sum << endl;**

**sum = 0;**

**cout << "------------------With critical--------------"<< endl;**

**#pragma omp parallel for**

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

**#pragma omp critical**

**{**

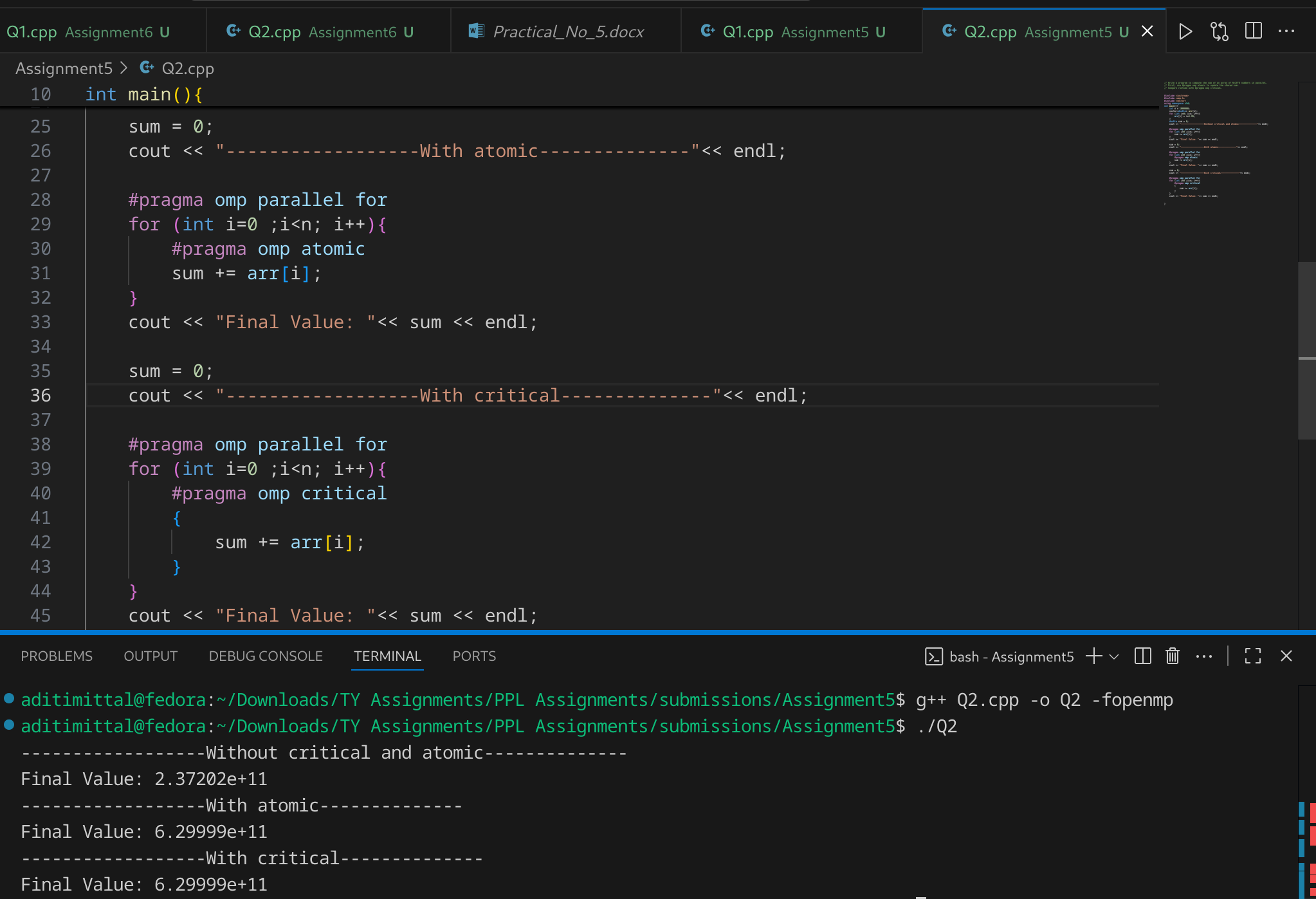
**sum += arr[i];**

**}**

**}**

**cout << "Final Value: "<< sum << endl;**

**}**

****

### Information & Analysis:

* **Without Synchronization:** Wrong result due to race conditions.
* **With atomic:** Lightweight synchronization. Ensures correctness with minimal overhead. Faster than critical.
* **With critical:** Ensures correctness, but slower than atomic because every thread waits longer to enter the section.

**Observation:** atomic is preferred for simple updates like sum += value.

**Problem Statement 3:**

Write a program with two parallel regions:

* + In the first part, each thread computes the square of its thread ID and stores it in an array.
  + Use a #pragma omp barrier to ensure all threads finish before the master thread prints the array.

**Screenshots:**

**// Write a program with two parallel regions:**

**// In the first part, each thread computes the square of its thread ID and stores it in an array.**

**// Use a #pragma omp barrier to ensure all threads finish before the master thread prints the array.**

**#include <iostream>**

**#include <omp.h>**

**#include <vector>**

**using namespace std;**

**int main() {**

**int num\_threads = 4;**

**vector<int> arr(num\_threads);**

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

**#pragma omp parallel**

**{**

**int thread\_id = omp\_get\_thread\_num();**

**arr[thread\_id] = thread\_id \* thread\_id;**

**#pragma omp barrier**

**#pragma omp master**

**{**

**cout << "Squares of thread IDs: ";**

**for (int i = 0; i < num\_threads; i++) {**

**cout << arr[i] << " ";**

**}**

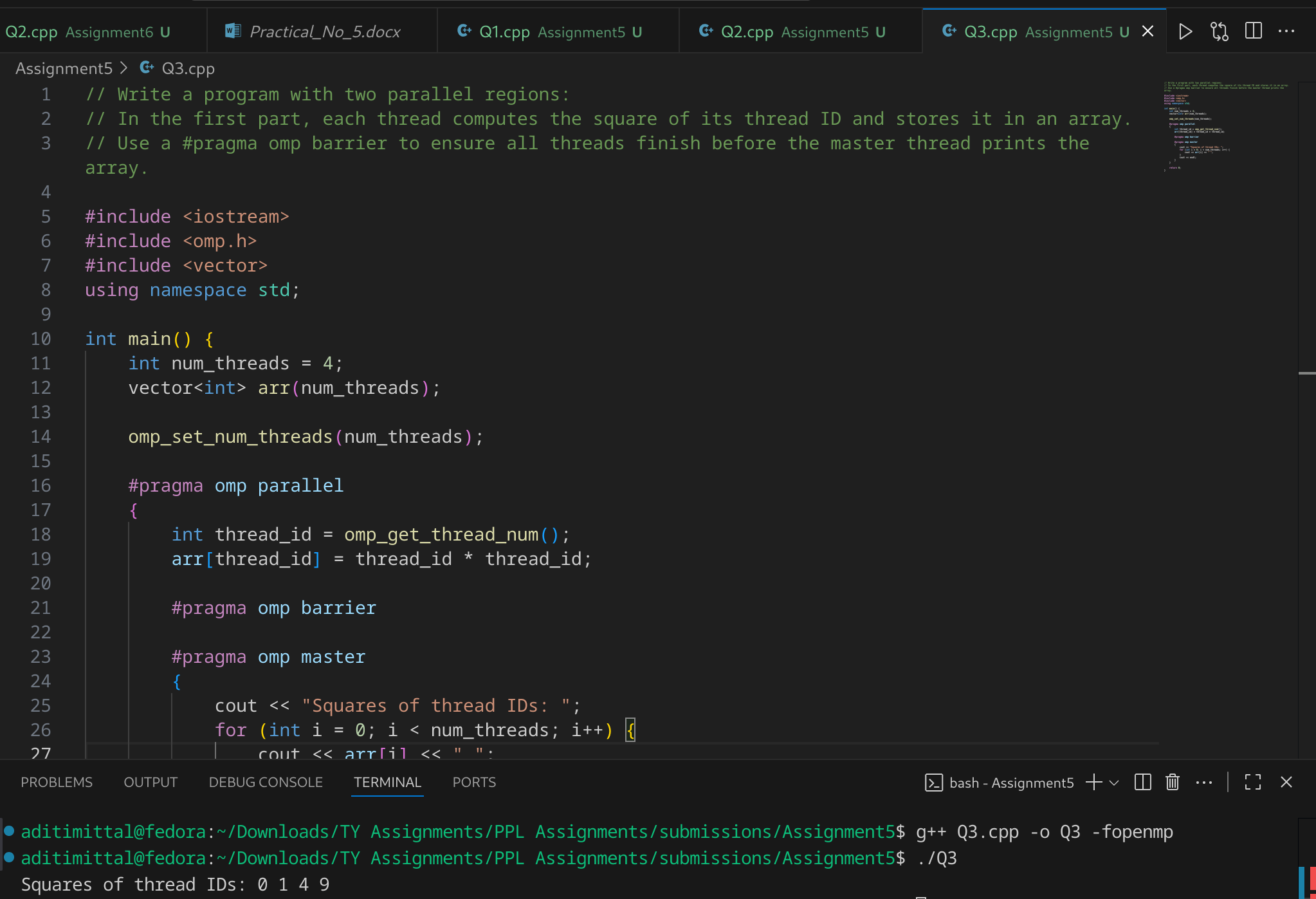
**cout << endl;**

**}**

**}**

**return 0;**

**}**

****

### Information & Analysis:

* Each thread computes square = tid² and stores in arr.
* **Barrier:** Ensures all threads complete storing before master thread prints.
* Without barrier, master may print partial/garbage data.
* Shows how #pragma omp barrier enforces order of execution across threads.

**Problem Statement 4:**

Write a program to print numbers from 1 to 20 in ascending order using a parallel for loop.

* + Use #pragma omp ordered to maintain order.
  + Run the same code without ordered and compare the output.

**Screenshots:**

**// Write a program to print numbers from 1 to 20 in ascending order using a parallel for loop.**

**// Use #pragma omp ordered to maintain order.**

**// Run the same code without ordered and compare the output.**

**#include <iostream>**

**#include <omp.h>**

**using namespace std;**

**int main() {**

**int N = 20;**

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

**#pragma omp parallel for**

**for (int i = 1; i <= N; i++) {**

**cout << i << " ";**

**}**

**cout << endl;**

**cout << "---------With ordered ---------" << endl;**

**#pragma omp parallel for ordered**

**for (int i = 1; i <= N; i++) {**

**#pragma omp ordered**

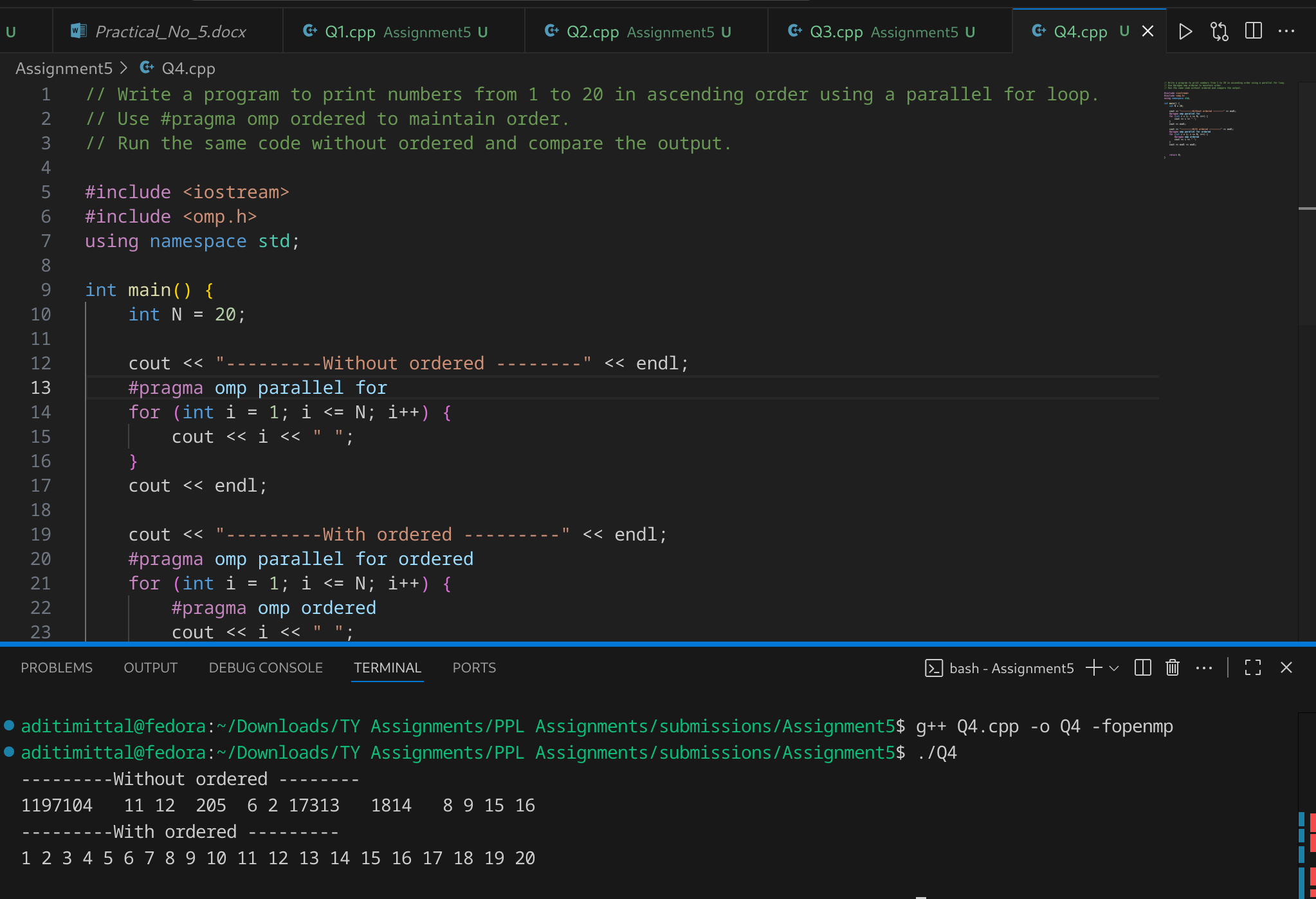
**cout << i << " ";**

**}**

**cout << endl << endl;**

**return 0;**

**}**

****

### Information & Analysis:

* **Without ordered:** Threads print numbers in arbitrary order → jumbled output.
* **With ordered:** #pragma omp ordered enforces ascending order printing.
* Demonstrates deterministic vs nondeterministic execution.

**Problem Statement 5:**

Write a program with two parallel for loops inside the same parallel region:

* + First loop initializes an array.
  + Second loop immediately prints the array elements.
  + Use #pragma omp for nowait on the first loop and observe what happens (incorrect output).
  + Then remove nowait to fix the issue.

**Screenshots:**

// Write a program with two parallel for loops inside the same parallel region:

// First loop initializes an array.

// Second loop immediately prints the array elements.

// Use #pragma omp for nowait on the first loop and observe what happens (incorrect output).

// Then remove nowait to fix the issue.

#include <iostream>

#include <omp.h>

#include <vector>

using namespace std;

int main() {

int N = 10;

vector<int> arr(N, 0);

cout << "---------Using nowait ---------" << endl;

#pragma omp parallel

{

#pragma omp for nowait

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

arr[i] = i \* 2;

}

#pragma omp for

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

cout << arr[i] << " ";

}

}

cout << endl << endl;

cout << "---------Without nowait ---------" << endl;

#pragma omp parallel

{

#pragma omp for

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

arr[i] = i \* 2;

}

#pragma omp for

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

cout << arr[i] << " ";

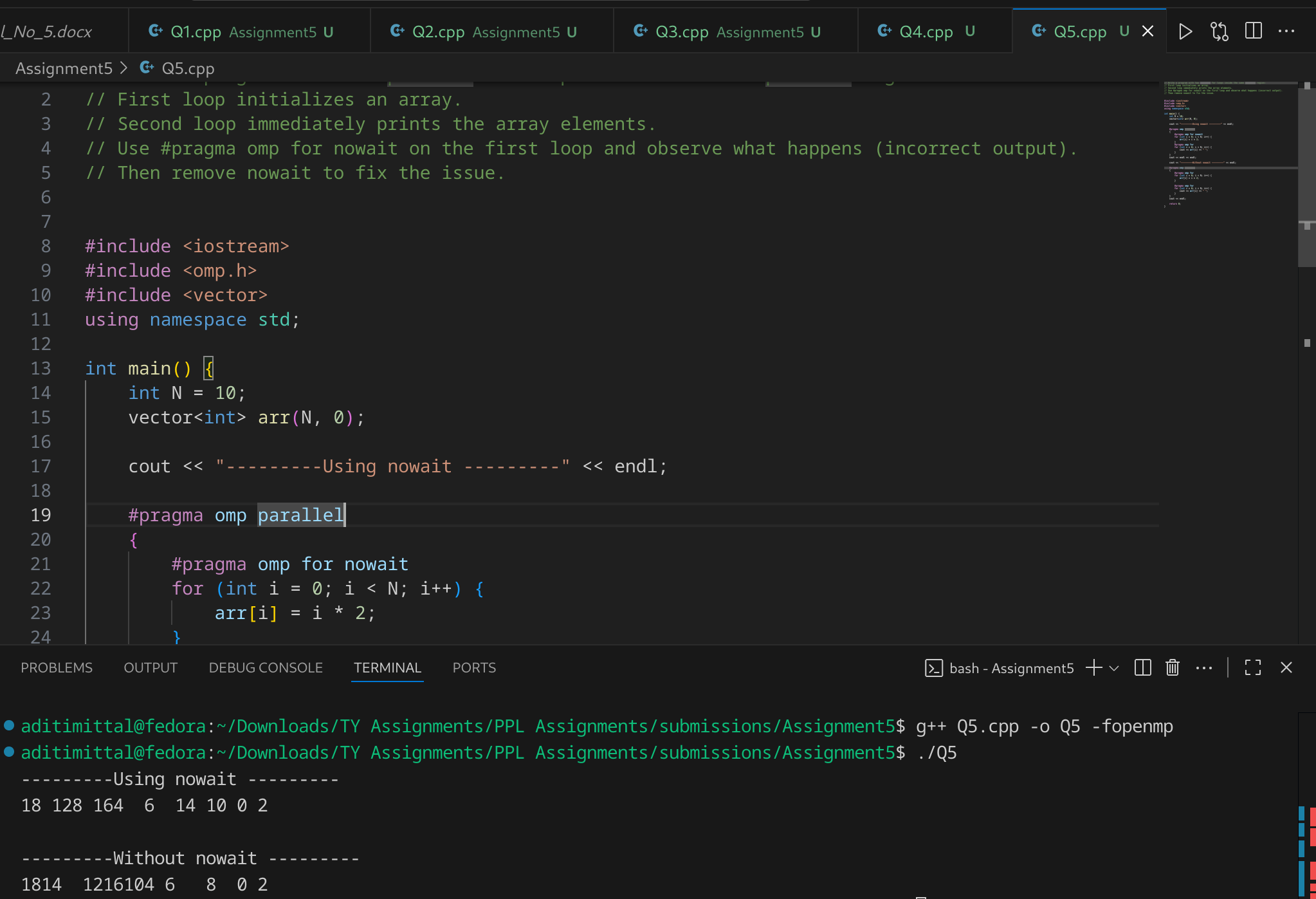
}

}

cout << endl;

return 0;

}



### Information & Analysis:

* **With nowait:** Second loop may start before first loop finishes → uninitialized values printed → incorrect output.
* **Without nowait:** Default barrier ensures first loop completes initialization before printing → correct output.
* **Lesson:** nowait must be used carefully; barriers are necessary when one loop depends on the previous.

**Problem Statement 6:**

Write a program to calculate factorial of n in parallel using OpenMP.

* + Use critical or atomic for shared multiplication.
  + Then optimize using reduction.

**Screenshots:**

// Write a program to calculate factorial of n in parallel using OpenMP.

// Use critical or atomic for shared multiplication.

// Then optimize using reduction.

#include <iostream>

#include <omp.h>

using namespace std;

int main() {

int n;

cout << "Enter n: ";

cin >> n;

//-----------------Parallel without sync-----------

long long fact\_parallel= 1;

#pragma omp parallel for

for (int i = 1; i <= n; i++) {

fact\_parallel \*= i;

}

cout << "Factorial without synchronisation: " << fact\_parallel << endl;

// ---------------- Using critical ----------------

long long fact\_critical = 1;

#pragma omp parallel for

for (int i = 1; i <= n; i++) {

#pragma omp critical

fact\_critical \*= i;

}

cout << "Factorial using critical: " << fact\_critical << endl;

// ---------------- Using atomic ----------------

long long fact\_atomic = 1;

#pragma omp parallel for

for (int i = 1; i <= n; i++) {

#pragma omp atomic

fact\_atomic \*= i;

}

cout << "Factorial using atomic: " << fact\_atomic << endl;

// ---------------- Using reduction ----------------

long long fact\_reduction = 1;

#pragma omp parallel for reduction(\*:fact\_reduction)

for (int i = 1; i <= n; i++) {

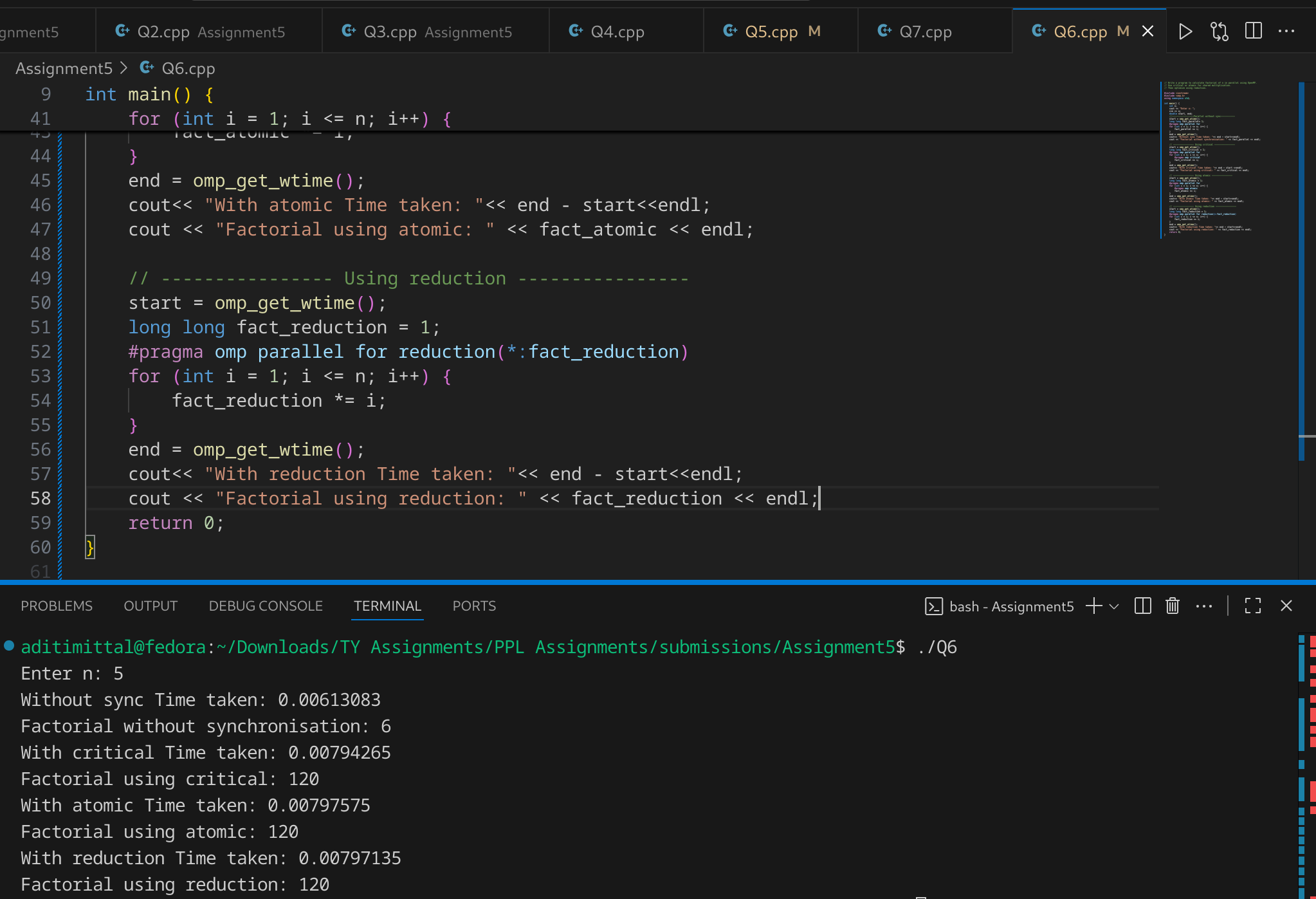
fact\_reduction \*= i;

}

cout << "Factorial using reduction: " << fact\_reduction << endl;

return 0;

}



### Information & Analysis:

* **Without sync:** Wrong (race condition).
* **With critical:** Correct but slower (serialization).
* **With atomic:** Works for addition but not ideal for multiplication (less efficient, may still serialize).
* **With reduction(\*:fact):** Best approach → parallel partial multiplications, then combine results. Correct + fastest.

Reduction gives scalability and avoids excessive synchronization.

**Problem Statement 7:**

Sum of squares of first 1 million numbers.

**Screenshots:**

**#include <iostream>**

**#include <omp.h>**

**using namespace std;**

**int main() {**

**long long N = 1000000;**

**long long sum\_seq = 0, sum\_parallel = 0, sum\_with\_reduction = 0;**

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

**for(long long i = 1; i <= N; i++) {**

**sum\_seq += i \* i;**

**}**

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

**#pragma omp parallel for**

**for(long long i = 1; i <= N; i++) {**

**sum\_parallel += i \* i;**

**}**

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

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

**for(long long i = 1; i <= N; i++) {**

**sum\_with\_reduction += i \* i;**

**}**

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

**cout << "Sum of squares (Sequential) = " << sum\_seq << endl;**

**cout << "Sequential Time = " << end\_seq - start\_seq << " seconds\n\n";**

**cout << "Sum of squares (Parallel) = " << sum\_parallel << endl;**

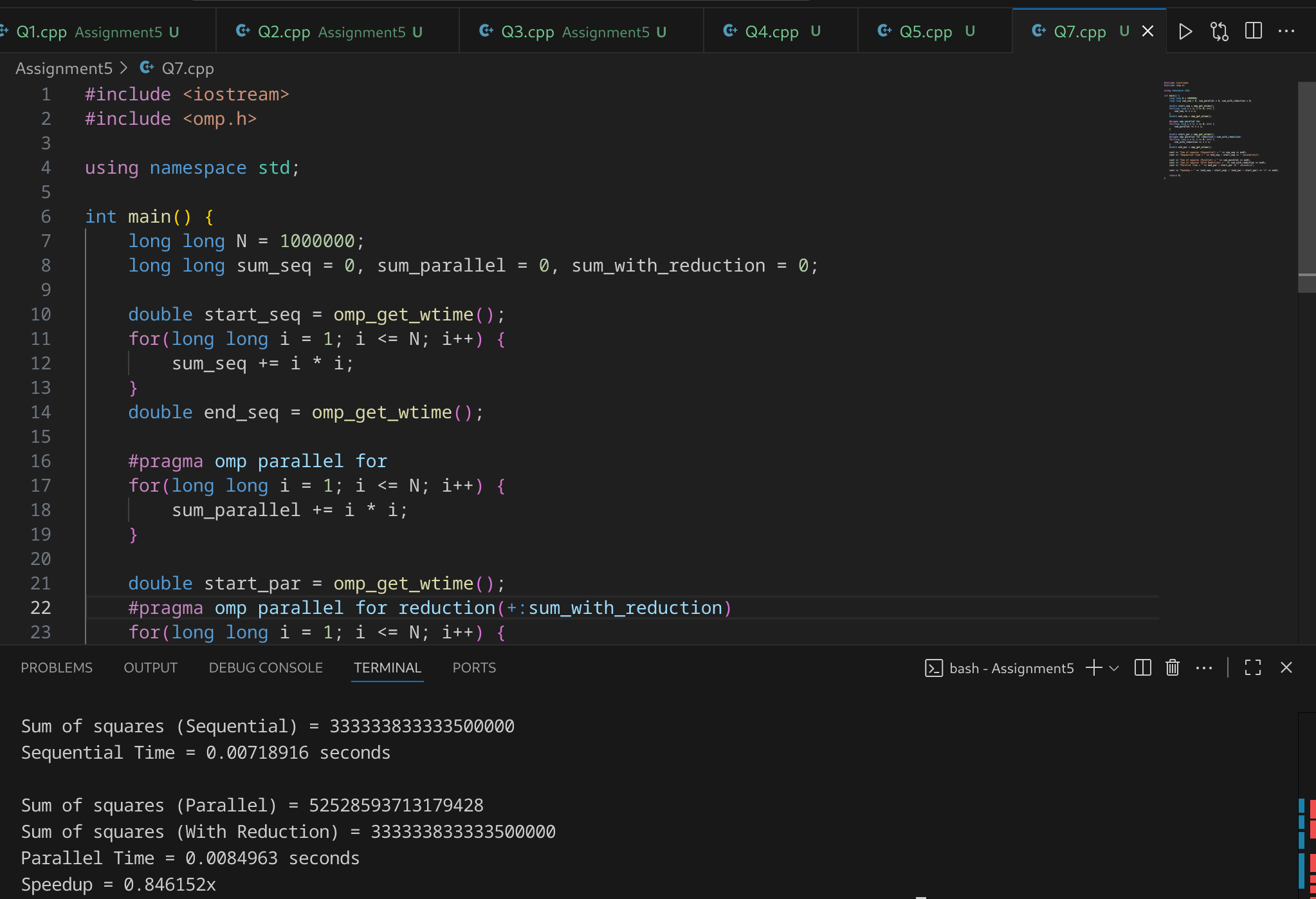
**cout << "Sum of squares (With Reduction) = " << sum\_with\_reduction << endl;**

**cout << "Parallel Time = " << end\_par - start\_par << " seconds\n";**

**cout << "Speedup = " << (end\_seq - start\_seq) / (end\_par - start\_par) << "x" << endl;**

**return 0;**

**}**

****

### Information & Analysis:

* **Sequential:** Simple loop, O(N).
* **Parallel without sync:** Wrong due to races.
* **Parallel with reduction:** Correct + efficient. Workload distributed across threads, then results combined.
* **Speedup:** Parallel reduction gives significant improvement over sequential for large N.

# Speedup (General Trends)

1. **Critical vs Atomic vs Reduction**
   * Critical → highest overhead.
   * Atomic → good for simple updates.
   * Reduction → best for associative operations (sum, product).
2. **Barrier**
   * Ensures correctness in dependent tasks.
   * Adds some overhead but necessary.
3. **Ordered**
   * Maintains sequential consistency.
   * Performance slightly slower than unordered but ensures correctness.
4. **Nowait**
   * Reduces barrier cost when tasks are independent.
   * Wrong output if used carelessly in dependent loops.
5. **Speedup**
   * For large N (10⁶ or more), reduction achieves the best parallel speedup (close to number of threads).
   * Synchronization constructs ensure correctness but trade-off with performance.

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