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

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

**Course:** Cutting Edge Technologies Lab

**Course code:** 7CS352

**Practical No. 4**

**Exam Seat No:** 23610067

**Title of practical:**

Study and Implementation of Use of private, shared, firstprivate, lastprivate clauses in OpenMP.

**Problems:**

Q1. Write an OpenMP program such that, it should print the name of your family members, such that the names should come from different threads/cores. Also print the respective job id.

Q2. Write an OpenMP program such that, it should print the sum of square of the thread id’s. Also make sure that, each thread should print the square value of their thread id.

Q3. Consider a variable called “Aryabhatta” declared as 10 (i.e int Arbhatta=10).Write an OpenMP program which should print the result of multiplication of thread id and value of the above variable.

Note\*: The variable “Aryabhatta” should be declared as private

Q4. Write an OpenMP program that calculates the partial sum of the first 20 natural numbers using parallelism. Each thread should compute a portion of the sum by iterating through a loop. Implement the program using the lastprivate clause to ensure that the final total sum is correctly computed and printed outside the parallel region.

Hint:

1.Utilize OpenMP directives to parallelize the summation process.

2.Ensure that each thread has its private copy of partial sum.

3.Use the lastprivate clause to assign the value of the last thread's partial sum to the final total sum after the parallel region.

Q5. Consider a scenario where you have to parallelize a program that performs matrix multiplication using OpenMP. Your task is to implement parallelization using both static and dynamic scheduling, and compare the execution time of each approach.

Note\*:

* Implement a serial version of matrix multiplication in C/C++.
* Parallelize the matrix multiplication using OpenMP with static scheduling.
* Parallelize the matrix multiplication using OpenMP with dynamic scheduling.
* Measure the execution time of each parallelized version for various matrix sizes.
* Compare the execution times and discuss the advantages and disadvantages of static and dynamic scheduling in this context.

Q6. Write a Parallel C program which should print the series of 2  and 4. Make sure both should be executed by different threads !

Q7. Consider a scenario where you have a shared variable total\_sum that needs to be updated concurrently by multiple threads in a parallel program. However, concurrent updates to this variable can result in data races and incorrect results. Your task is to modify the program to ensure correct synchronization using OpenMP's critical and atomic constructs.

Note\*:

* Implement a simple parallel program in C that initializes an array of integers and calculates the sum of its elements concurrently using OpenMP.
* Identify potential issues with concurrent updates to the total\_sum variable in the parallelized version of the program.
* Modify the program to use OpenMP's critical/atomic directive to ensure synchronized access to the total\_sum variable.
* Measure and compare the performance of synchronized versions against the unsynchronized implementation.

Q8. Consider a scenario where you have a large array of integers, and you need to find the sum of all its elements in parallel using OpenMP. The array is shared among multiple threads, and parallelism is needed to expedite the computation process. Your task is to write a parallel program that calculates the sum of all elements in the array using OpenMP's reduction clause.

Note\*:

* Implement a sequential version of the program that calculates the sum of all elements in the array without using any parallelism.
* Identify potential bottlenecks and limitations of the sequential implementation in handling large arrays efficiently.
* Modify the program to utilize OpenMP's reduction clause to parallelize the summation process across multiple threads.
* Test the program with different array sizes and thread counts to evaluate its scalability and performance.
* Discuss the advantages of using the reduction clause for parallel summation and its impact on program efficiency.

Q9: Implementation of calculation of Pi using OpenMP.

**Problem Statement 1:**

Write an OpenMP program such that, it should print the name of your family members, such that the names should come from different threads/cores. Also print the respective job id.

**Screenshots:**

// Q1. Write an OpenMP program such that, it should print the name of your family members, such that the names should come from different threads/cores. Also print the respective job id.

#include <iostream>

#include <omp.h>

#include <iomanip>

using namespace std;

int main() {

string names[] = {"Rakesh", "Ritu", "Tushar", "Me"};

int n = sizeof(names) / sizeof(names[0]);

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

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

cout << "Seq -> " << names[i] << " (JobID: " << i << ")" << endl;

}

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

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

#pragma omp parallel num\_threads(4)

{

int tid = omp\_get\_thread\_num();

if (tid < n) {

cout << "Thread " << tid << " -> " << names[tid] << " (JobID: " << tid << ")" << endl;

}

}

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

double seq\_time = end\_seq - start\_seq;

double par\_time = end\_par - start\_par;

cout << fixed << setprecision(9);

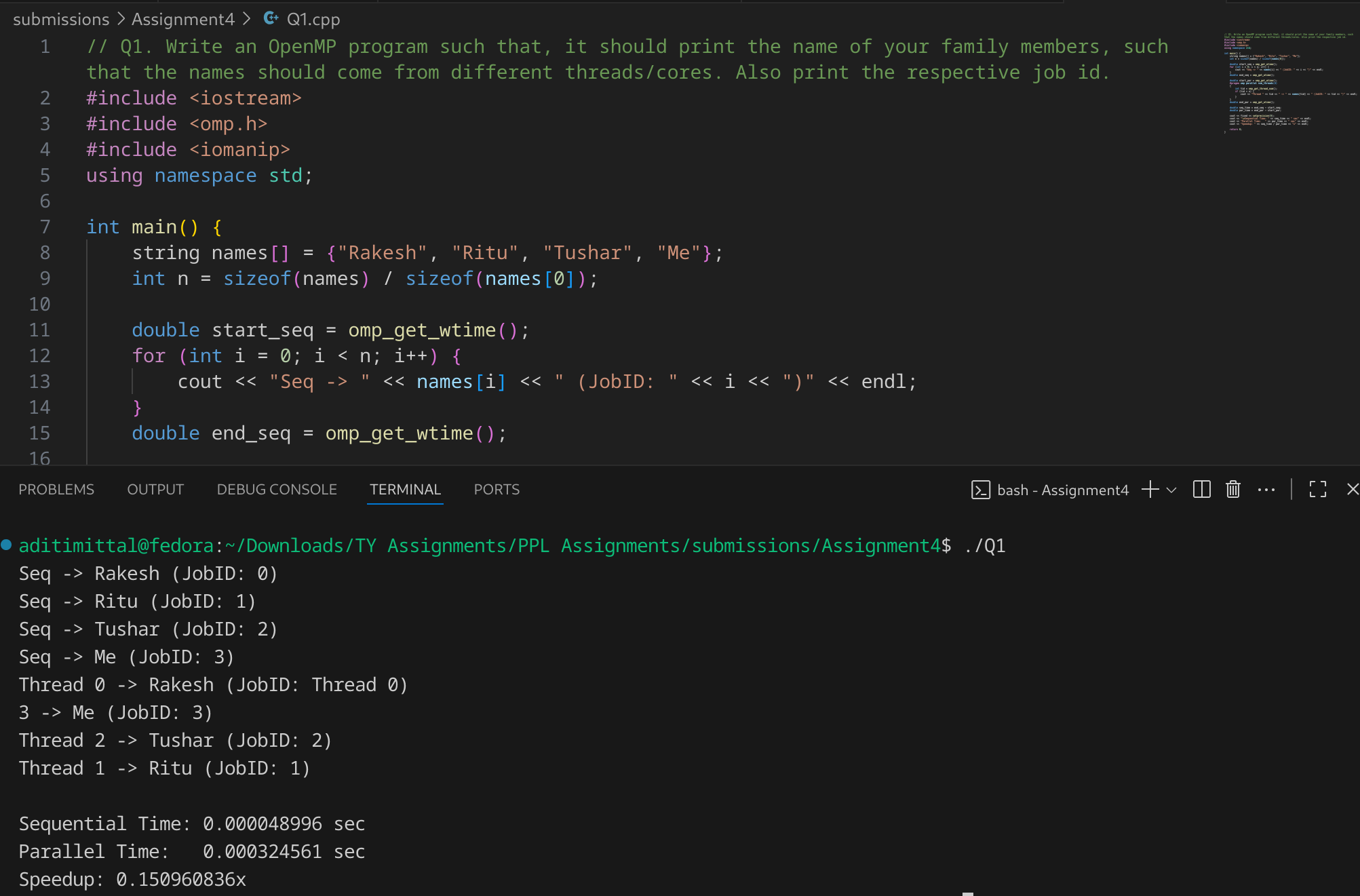
cout << "\nSequential Time: " << seq\_time << " sec" << endl;

cout << "Parallel Time: " << par\_time << " sec" << endl;

cout << "Speedup: " << seq\_time / par\_time << "x" << endl;

return 0;

}

****

**Speedup is low** because:

* Printing is **not computationally heavy**.
* Parallelization adds **thread management overhead**.

Sometimes, **parallel execution may even be slower** (if par\_time > seq\_time) because of synchronization costs in cout.

For **I/O tasks** → Parallelism doesn’t help much.  
 For **CPU-heavy tasks** (factorial, sum of squares, matrix multiplication) → Parallelism gives **significant speedup**.

**Problem Statement 2:**

Write an OpenMP program such that, it should print the sum of square of the thread id’s. Also make sure that, each thread should print the square value of their thread id

**Screenshots:**

// Write an OpenMP program such that, it should print the sum of square of the thread id’s. Also make sure that, each thread should print the square value of their thread id

#include <iostream>

#include <omp.h>

#include <iomanip>

using namespace std;

int main() {

int n = 8;

int seq\_sum = 0;

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

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

int sq = i \* i;

cout << "Sequential -> Thread " << i << " square = " << sq << endl;

seq\_sum += sq;

}

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

int par\_sum = 0;

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

#pragma omp parallel num\_threads(n) reduction(+:par\_sum)

{

int tid = omp\_get\_thread\_num();

int sq = tid \* tid;

cout << "Parallel -> Thread " << tid << " square = " << sq << endl;

par\_sum += sq;

}

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

double seq\_time = end\_seq - start\_seq;

double par\_time = end\_par - start\_par;

cout << fixed << setprecision(9);

cout << "\nSequential Sum = " << seq\_sum

<< " | Time = " << seq\_time << " sec" << endl;

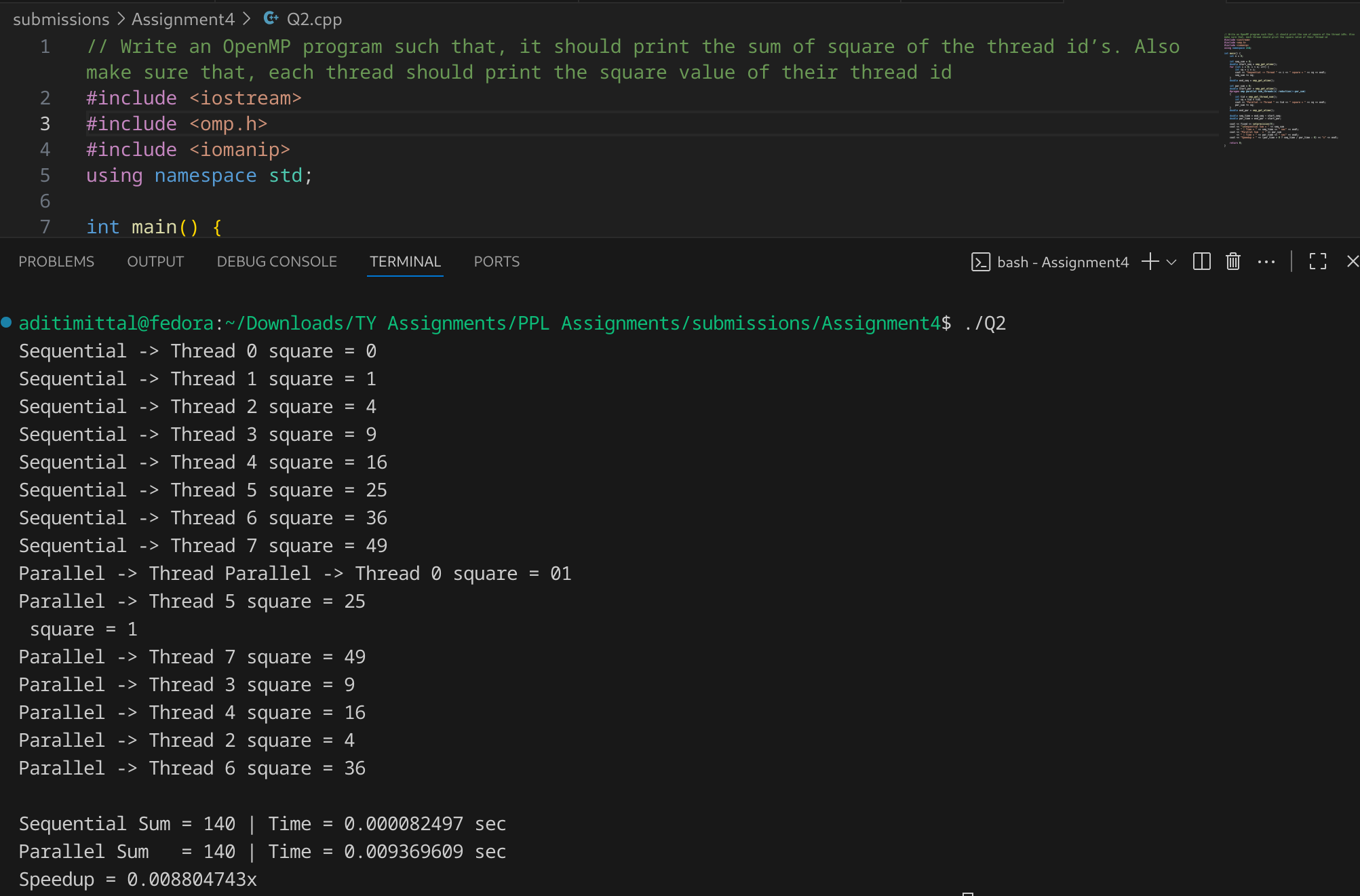
cout << "Parallel Sum = " << par\_sum

<< " | Time = " << par\_time << " sec" << endl;

cout << "Speedup = " << (par\_time > 0 ? seq\_time / par\_time : 0) << "x" << endl;

return 0;

}



**Information and Analysis:**

* Instead of speeding up, the parallel version is **~113× slower** than the sequential one.
* This happened because:  
  1. **Printing (cout) inside parallel region** → Console I/O is **sequentially synchronized** and very slow compared to computation. Threads end up waiting on each other to print.
  2. The computation (just squaring a number) is trivial, so overhead of thread creation + synchronization far outweighs the work.

**Expected behavior in heavier tasks:**

* If instead of just 8 squares, we computed sum of squares of, say, **10⁷ numbers**, the parallel execution would reduce time drastically and show a **real speedup (~3–6x depending on cores)**.

**Problem Statement 3:**

Consider a variable called “Aryabhatta” declared as 10 (i.e int Arbhatta=10).Write an OpenMP program which should print the result of multiplication of thread id and value of the above variable.

Note\*: The variable “Aryabhatta” should be declared as private

**Screenshots:**

// Consider a variable called “Aryabhatta” declared as 10 (i.e int Arbhatta=10).Write an OpenMP program which should print the result of multiplication of thread id and value of the above variable.

// Note\*: The variable “Aryabhatta” should be declared as private

#include <iostream>

#include <omp.h>

using namespace std;

int main() {

int Aryabhatta = 10;

#pragma omp parallel private(Aryabhatta)

{

Aryabhatta = 10;

int tid = omp\_get\_thread\_num();

int result = tid \* Aryabhatta;

cout << "Thread " << tid

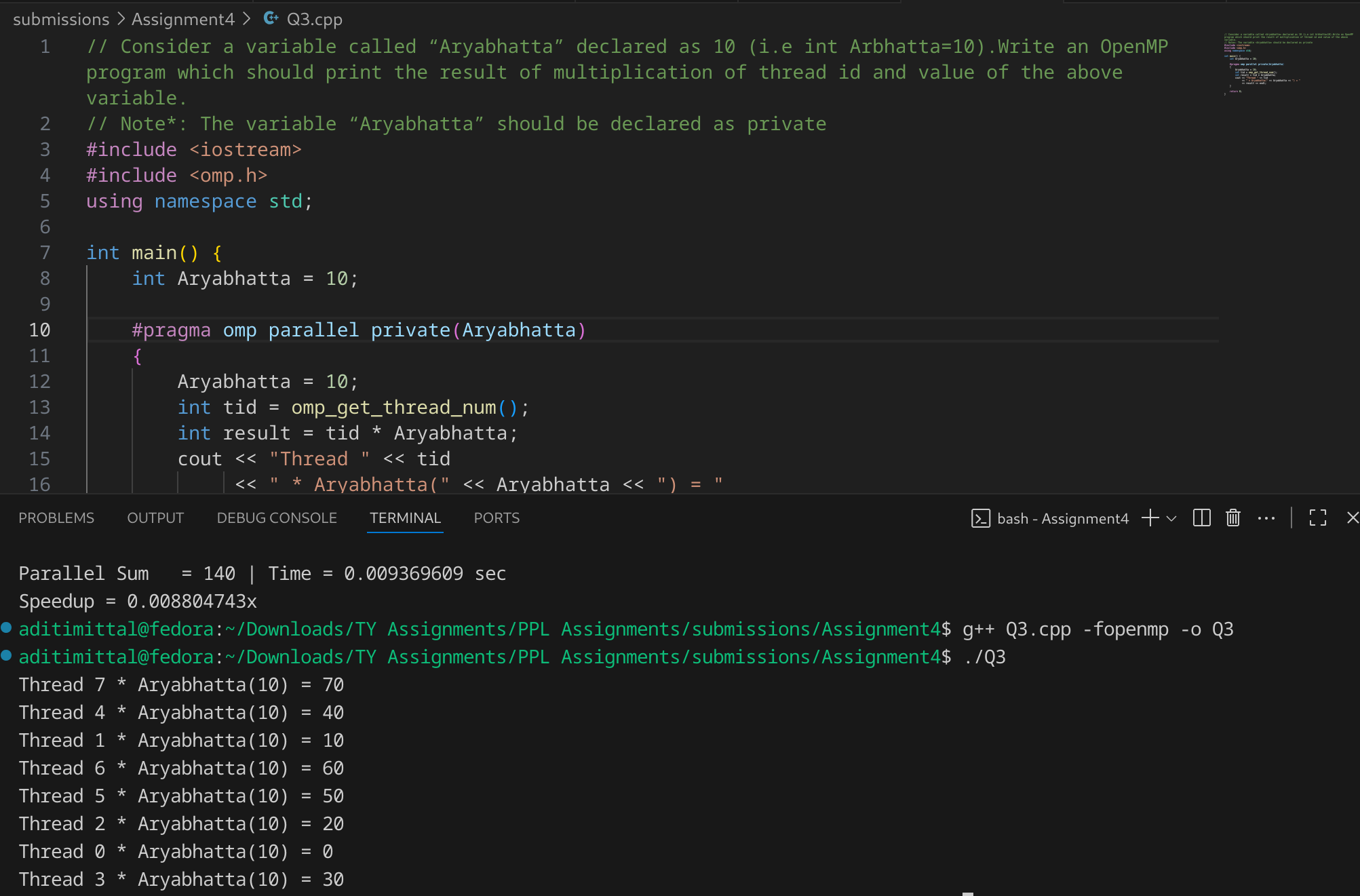
<< " \* Aryabhatta(" << Aryabhatta << ") = "

<< result << endl;

}

return 0;

}



**Information:**

* private(Aryabhatta) → ensures each thread has its own **independent copy**.
* Thread IDs help **differentiate work distribution**.
* The program demonstrates **parallel execution** with **per-thread variables**.

**Problem Statement 4:**

Write an OpenMP program that calculates the partial sum of the first 20 natural numbers using parallelism. Each thread should compute a portion of the sum by iterating through a loop. Implement the program using the lastprivate clause to ensure that the final total sum is correctly computed and printed outside the parallel region.

Hint:

1.Utilize OpenMP directives to parallelize the summation process.

2.Ensure that each thread has its private copy of partial sum.

3.Use the lastprivate clause to assign the value of the last thread's partial sum to the final total sum after the parallel region.

**Screenshots:**

// Write an OpenMP program that calculates the partial sum of the first 20 natural numbers using parallelism. Each thread should compute a portion of the sum by iterating through a loop. Implement the program using the lastprivate clause to ensure that the final total sum is correctly computed and printed outside the parallel region.

// Hint:

// 1.Utilize OpenMP directives to parallelize the summation process.

// 2.Ensure that each thread has its private copy of partial sum.

// 3.Use the lastprivate clause to assign the value of the last thread's partial sum to the final total sum after the parallel region.

#include <iostream>

#include <omp.h>

using namespace std;

int main() {

int total\_sum = 0;

#pragma omp parallel for lastprivate(total\_sum)

for (int i = 1; i <= 20; i++) {

int partial\_sum = 0;

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

partial\_sum += j;

}

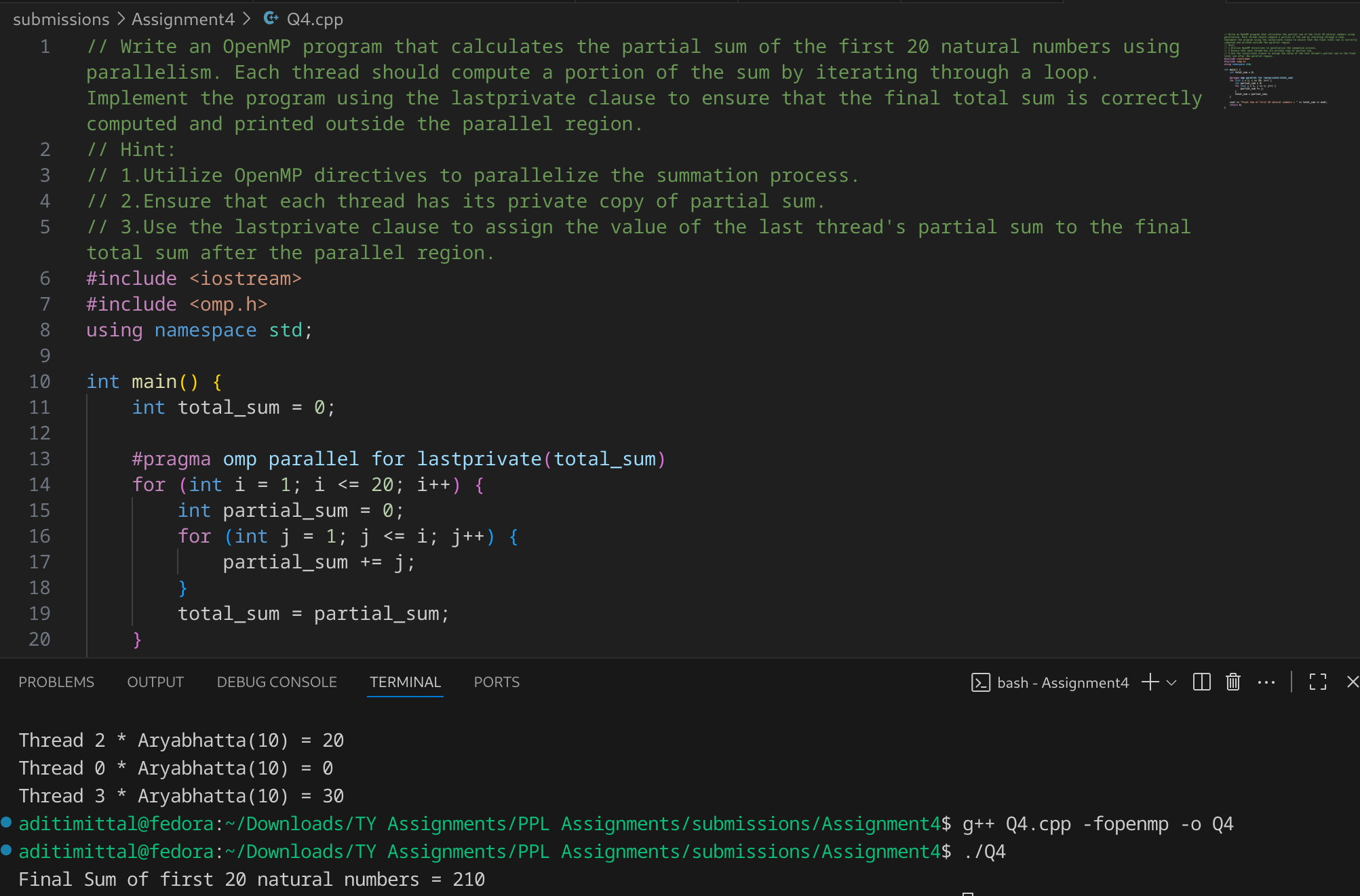
total\_sum = partial\_sum;

}

cout << "Final Sum of first 20 natural numbers = " << total\_sum << endl;

return 0;

}



**Information:**

### 1. What is lastprivate in OpenMP?

* The lastprivate clause in OpenMP ensures that **the value of a private variable from the last iteration of the loop** is copied to the variable outside the parallel region.

Syntax:  
  
 #pragma omp parallel for lastprivate(variable)

### 2. Why do we need it here?

* Inside the loop, we calculate a **partial sum** up to i.
* Without lastprivate, the variable total\_sum outside the loop won’t know the **final value from the last iteration** (i=20).
* Using lastprivate(total\_sum) ensures that after the loop finishes:  
  + The **value from the final iteration (i=20)** is stored in total\_sum.
  + Hence, we correctly get the sum = 210.

### ✅ Key Point

* private → Each thread has its **own copy**, discarded after use.
* lastprivate → Same as private, but the **last iteration’s value is saved** to the outer variabl

**Problem Statement 5:**

Consider a scenario where you have to parallelize a program that performs matrix multiplication using OpenMP. Your task is to implement parallelization using both static and dynamic scheduling, and compare the execution time of each approach.

Note\*:

* Implement a serial version of matrix multiplication in C/C++.
* Parallelize the matrix multiplication using OpenMP with static scheduling.
* Parallelize the matrix multiplication using OpenMP with dynamic scheduling.
* Measure the execution time of each parallelized version for various matrix sizes.
* Compare the execution times and discuss the advantages and disadvantages of static and dynamic scheduling in this context.

**Screenshots:**

// Consider a scenario where you have to parallelize a program that performs matrix multiplication using OpenMP. Your task is to implement parallelization using both static and dynamic scheduling, and compare the execution time of each approach.

// Note\*:

// Implement a serial version of matrix multiplication in C/C++.

// Parallelize the matrix multiplication using OpenMP with static scheduling.

// Parallelize the matrix multiplication using OpenMP with dynamic scheduling.

// Measure the execution time of each parallelized version for various matrix sizes.

// Compare the execution times and discuss the advantages and disadvantages of static and dynamic scheduling in this context.

#include <iostream>

#include <omp.h>

#include <vector>

#include <iomanip>

using namespace std;

int main() {

int N;

cout << "Enter matrix size (N x N): ";

cin >> N;

vector<vector<int>> A(N, vector<int>(N, 1));

vector<vector<int>> B(N, vector<int>(N, 2));

vector<vector<int>> C(N, vector<int>(N, 0));

double start, end;

// ---------------- SERIAL ----------------

start = omp\_get\_wtime();

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

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

int sum = 0;

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

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

}

C[i][j] = sum;

}

}

end = omp\_get\_wtime();

double serial\_time = end - start;

cout << "Serial Time: " << serial\_time << " sec" << endl;

// ---------------- STATIC SCHEDULING ----------------

C.assign(N, vector<int>(N, 0));

start = omp\_get\_wtime();

#pragma omp parallel for schedule(static)

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

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

int sum = 0;

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

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

}

C[i][j] = sum;

}

}

end = omp\_get\_wtime();

double static\_time = end - start;

cout << "Static Scheduling Time: " << static\_time << " sec" << endl;

cout << "Speedup (Static): " << serial\_time / static\_time << "x" << endl;

// ---------------- DYNAMIC SCHEDULING ----------------

C.assign(N, vector<int>(N, 0));

start = omp\_get\_wtime();

#pragma omp parallel for schedule(dynamic)

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

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

int sum = 0;

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

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

}

C[i][j] = sum;

}

}

end = omp\_get\_wtime();

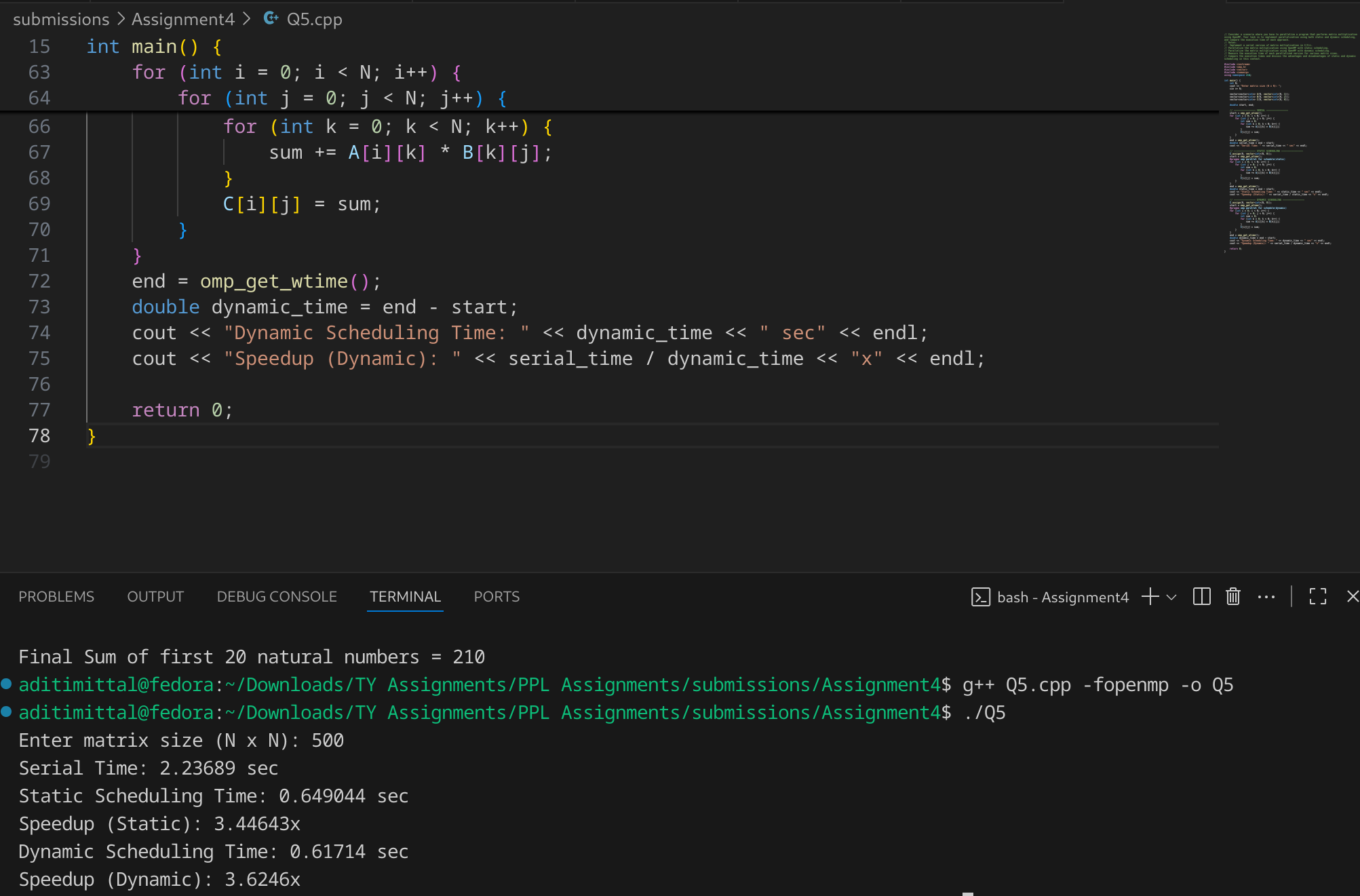
double dynamic\_time = end - start;

cout << "Dynamic Scheduling Time: " << dynamic\_time << " sec" << endl;

cout << "Speedup (Dynamic): " << serial\_time / dynamic\_time << "x" << endl;

return 0;

}



**Information:**

**Static Scheduling**

* Threads were assigned equal numbers of rows beforehand.
* Since matrix multiplication is *fairly uniform*, static works well.
* But still, some rows may take slightly longer (cache effects, memory alignment), causing **minor imbalance**.

**Dynamic Scheduling**

* Here, OpenMP handed out rows *on demand*.
* This improved **load balancing**, so the total execution time was a bit less than static.
* But the difference is **small**, because the workload is already quite uniform.

**Problem Statement 6:**

Write a Parallel C program which should print the series of 2  and 4. Make sure both should be executed by different threads !

**Screenshots:**

// Write a Parallel C program which should print the series of 2 and 4. Make sure both should be executed by different threads !

#include <iostream>

#include <omp.h>

using namespace std;

int main() {

int n = 10;

#pragma omp parallel num\_threads(2)

{

int tid = omp\_get\_thread\_num();

if (tid == 0) {

cout << "Thread " << tid << " printing multiples of 2:" << endl;

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

cout << "2 \* " << i << " = " << 2 \* i << endl;

}

}

if (tid == 1) {

cout << "Thread " << tid << " printing multiples of 4:" << endl;

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

cout << "4 \* " << i << " = " << 4 \* i << endl;

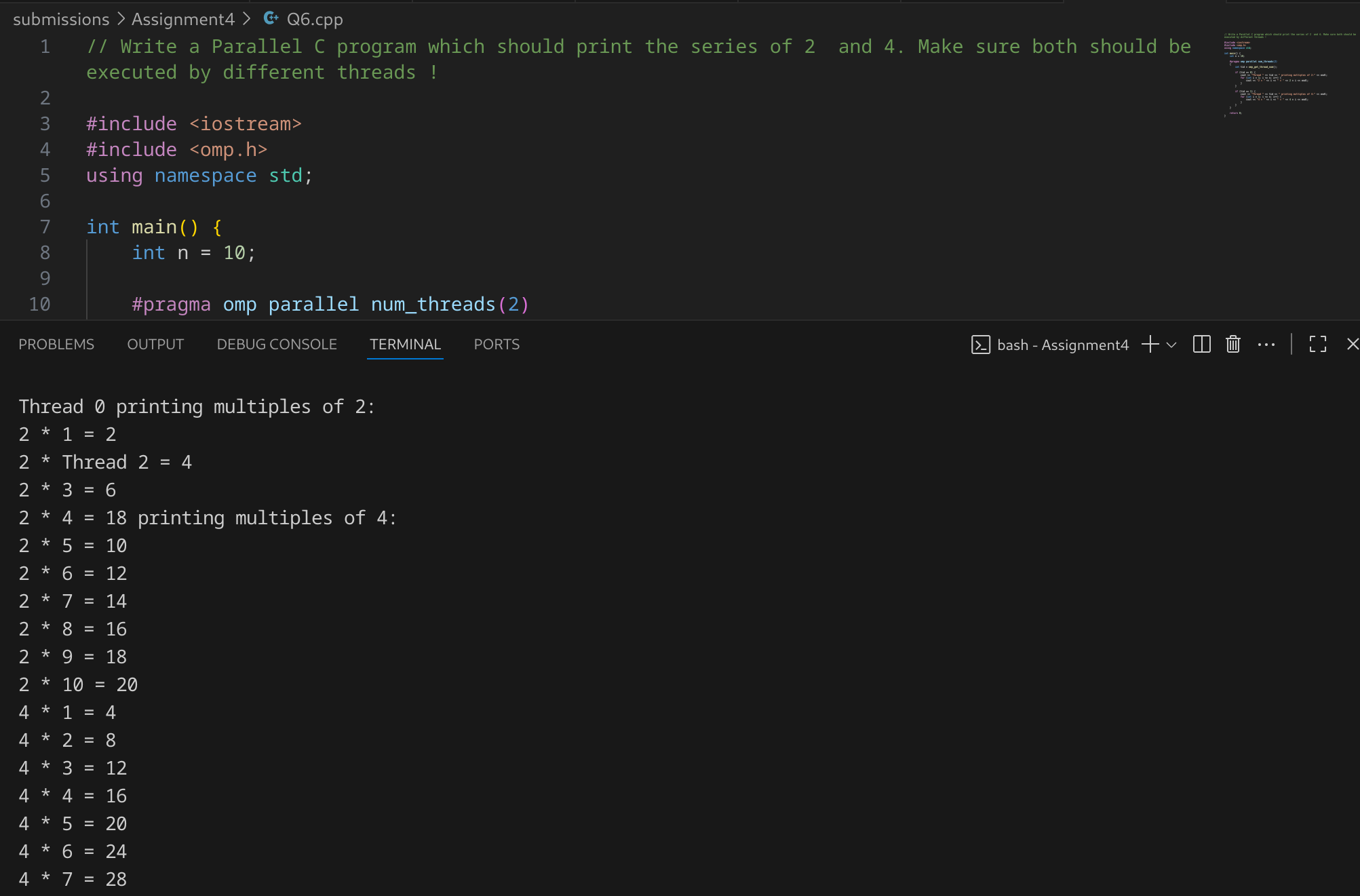
}

}

}

return 0;

}



**Information:**

**Threads Handling Different Tasks**

* Thread 0 → Series of 2
* Thread 1 → Series of 4  
   This ensures both sequences are **independent** and parallelized.

**OpenMP Parallel Region**

* #pragma omp parallel num\_threads(2) creates **two threads**.
* Each thread checks its **thread id (tid)** and executes the right loop.

**Performance**

* The workload here is very light (simple loops), so **speedup is negligible** compared to serial execution.
* However, this program demonstrates **task parallelism** (different tasks by different threads), not just **data parallelism**.

**Use Case**

* This approach is useful in real problems where **different computations** (not just parts of the same loop) must run in parallel.

**Problem Statement 7:**

Consider a scenario where you have a shared variable total\_sum that needs to be updated concurrently by multiple threads in a parallel program. However, concurrent updates to this variable can result in data races and incorrect results. Your task is to modify the program to ensure correct synchronization using OpenMP's critical and atomic constructs.

Note\*:

* Implement a simple parallel program in C that initializes an array of integers and calculates the sum of its elements concurrently using OpenMP.
* Identify potential issues with concurrent updates to the total\_sum variable in the parallelized version of the program.
* Modify the program to use OpenMP's critical/atomic directive to ensure synchronized access to the total\_sum variable.
* Measure and compare the performance of synchronized versions against the unsynchronized implementation.

**Screenshots:**

// Consider a scenario where you have a shared variable total\_sum that needs to be updated concurrently by multiple threads in a parallel program. However, concurrent updates to this variable can result in data races and incorrect results. Your task is to modify the program to ensure correct synchronization using OpenMP's critical and atomic constructs.

// Note\*:

// Implement a simple parallel program in C that initializes an array of integers and calculates the sum of its elements concurrently using OpenMP.

// Identify potential issues with concurrent updates to the total\_sum variable in the parallelized version of the program.

// Modify the program to use OpenMP's critical/atomic directive to ensure synchronized access to the total\_sum variable.

// Measure and compare the performance of synchronized versions against the unsynchronized implementation.

#include <iostream>

#include <omp.h>

#include <vector>

using namespace std;

int main() {

const int N = 1000000;

vector<int> arr(N, 1);

long long total\_sum = 0;

double start, end;

// ---------------- UNSYNCHRONIZED ----------------

total\_sum = 0;

start = omp\_get\_wtime();

#pragma omp parallel for

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

total\_sum += arr[i];

}

end = omp\_get\_wtime();

cout << "Unsynchronized Sum = " << total\_sum

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

// ---------------- CRITICAL ----------------

total\_sum = 0;

start = omp\_get\_wtime();

#pragma omp parallel for

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

#pragma omp critical

{

total\_sum += arr[i];

}

}

end = omp\_get\_wtime();

cout << "Critical Section Sum = " << total\_sum

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

// ---------------- ATOMIC ----------------

total\_sum = 0;

start = omp\_get\_wtime();

#pragma omp parallel for

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

#pragma omp atomic

total\_sum += arr[i];

}

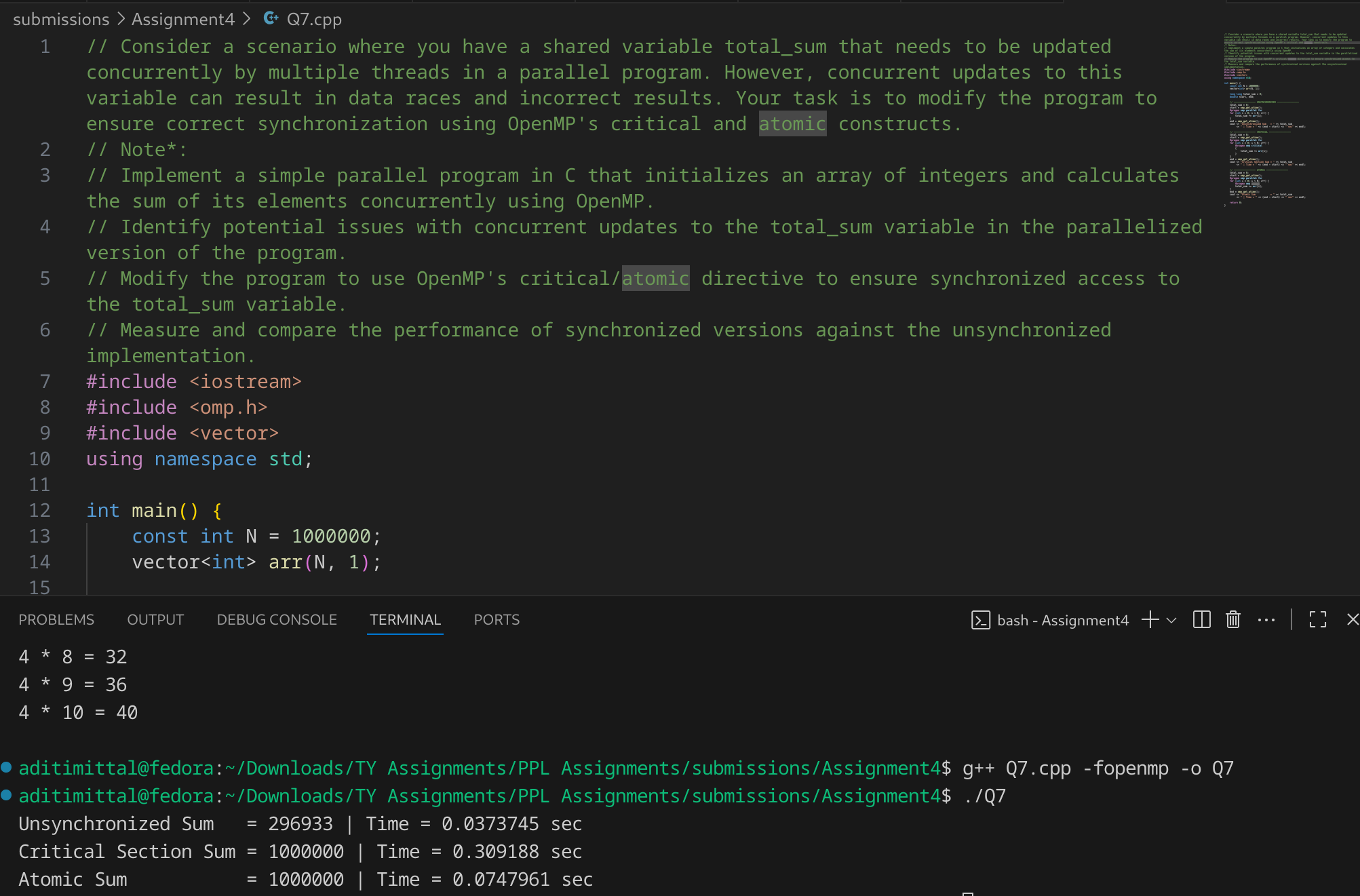
end = omp\_get\_wtime();

cout << "Atomic Sum = " << total\_sum

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

return 0;

}



**Information:**

**Race Condition (Unsynchronized)**

* Multiple threads update total\_sum **at the same time**.
* Leads to incorrect (smaller) values, since updates are lost.
* Fastest version, but wrong.

**Critical Section**

* #pragma omp critical ensures only **one thread at a time** updates total\_sum.
* Correct result, but slower due to higher synchronization overhead.

**Atomic Operation**

* #pragma omp atomic applies to simple operations like +=.
* Lighter weight than critical.
* Gives correct result and usually performs better than critical.

**Analysis:**

unsynchronized: wrong result, since many threads write at the same time.

critical: locks access → correct, but slower.

atomic: lighter than critical, correct, usually faster.

**Problem Statement 8:**

Consider a scenario where you have a large array of integers, and you need to find the sum of all its elements in parallel using OpenMP. The array is shared among multiple threads, and parallelism is needed to expedite the computation process. Your task is to write a parallel program that calculates the sum of all elements in the array using OpenMP's reduction clause.

Note\*:

* Implement a sequential version of the program that calculates the sum of all elements in the array without using any parallelism.
* Identify potential bottlenecks and limitations of the sequential implementation in handling large arrays efficiently.
* Modify the program to utilize OpenMP's reduction clause to parallelize the summation process across multiple threads.
* Test the program with different array sizes and thread counts to evaluate its scalability and performance.
* Discuss the advantages of using the reduction clause for parallel summation and its impact on program efficiency.

**Screenshots:**

// Problem Statement 8:

// Consider a scenario where you have a large array of integers, and you need to find the sum of all its elements in parallel using OpenMP. The array is shared among multiple threads, and parallelism is needed to expedite the computation process. Your task is to write a parallel program that calculates the sum of all elements in the array using OpenMP's reduction clause.

// Note\*:

// Implement a sequential version of the program that calculates the sum of all elements in the array without using any parallelism.

// Identify potential bottlenecks and limitations of the sequential implementation in handling large arrays efficiently.

// Modify the program to utilize OpenMP's reduction clause to parallelize the summation process across multiple threads.

// Test the program with different array sizes and thread counts to evaluate its scalability and performance.

// Discuss the advantages of using the reduction clause for parallel summation and its impact on program efficiency.

#include <iostream>

#include <omp.h>

#include <vector>

using namespace std;

int main() {

const int N = 100000000;

vector<int> arr(N, 1);

long long sum\_seq = 0, sum\_par = 0;

double start, end;

// ---------------- SEQUENTIAL ----------------

start = omp\_get\_wtime();

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

sum\_seq += arr[i];

}

end = omp\_get\_wtime();

double time\_seq = end - start;

cout << "Sequential Sum = " << sum\_seq

<< " | Time = " << time\_seq << " sec" << endl;

// ---------------- PARALLEL (reduction) ----------------

start = omp\_get\_wtime();

#pragma omp parallel for reduction(+:sum\_par)

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

sum\_par += arr[i];

}

end = omp\_get\_wtime();

double time\_par = end - start;

cout << "Parallel Sum = " << sum\_par

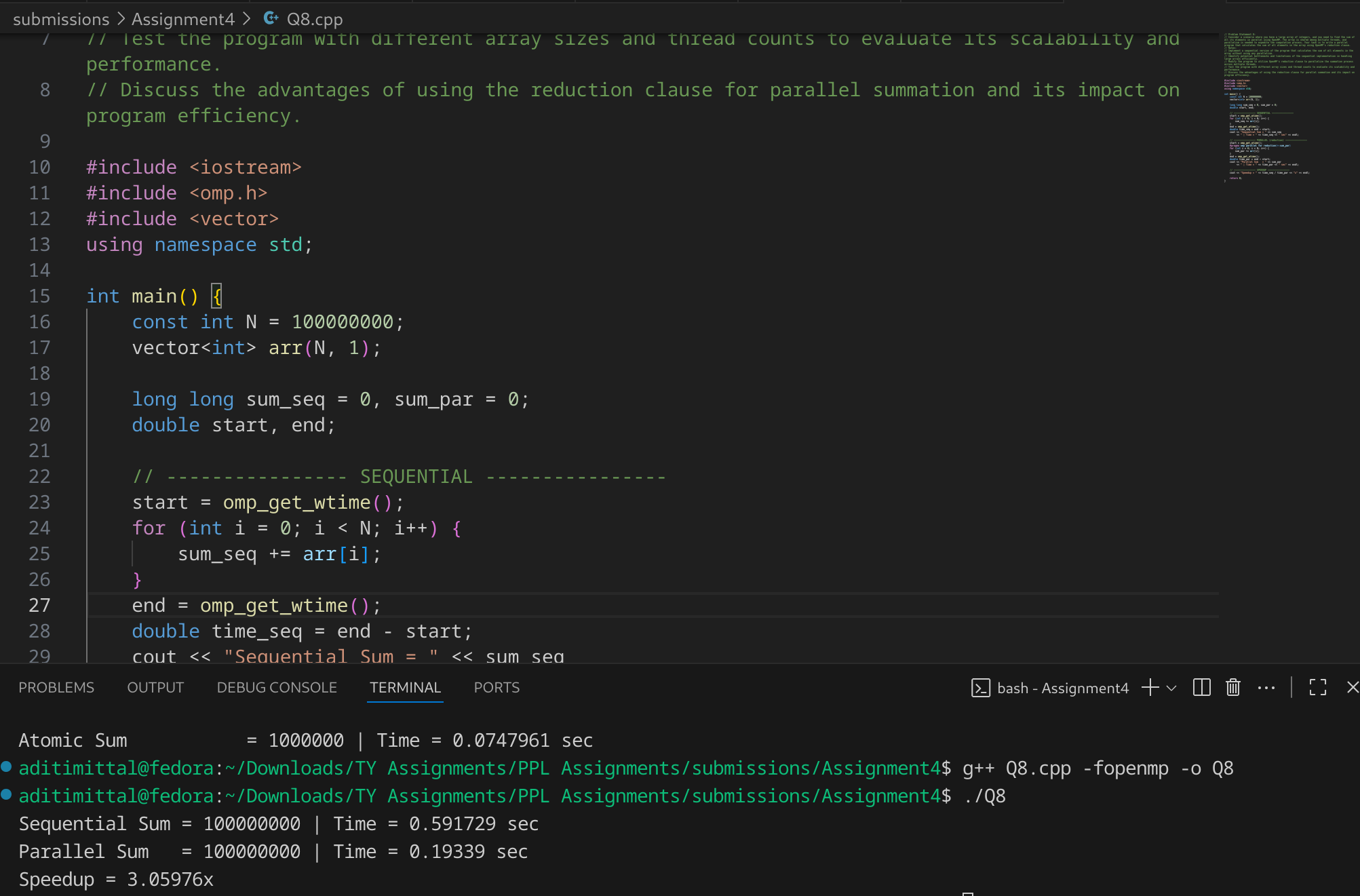
<< " | Time = " << time\_par << " sec" << endl;

// ---------------- SPEEDUP ----------------

cout << "Speedup = " << time\_seq / time\_par << "x" << endl;

return 0;

}



**Information:**

### Bottlenecks in Sequential

* Only **one CPU core** does all work.
* For very large arrays, memory bandwidth becomes limiting.

### Why Reduction Works Well

* Each thread keeps a **private local sum** → avoids race conditions.
* At the end, OpenMP combines results **efficiently** (tree reduction).
* Eliminates overhead of critical or atomic.

### Scalability

* Works well up to the number of physical cores.
* Diminishing returns after memory bandwidth is saturated.

**Analysis:**

## Advantages of reduction

1. **Correctness** → No race conditions.
2. **Performance** → Faster than critical/atomic.
3. **Simplicity** → One directive handles synchronization and summation.
4. **Scalability** → Efficient across many threads.

**Problem Statement 9:**

Implementation of calculation of Pi using OpenMP.

**Screenshots:**

// Implementation of calculation of Pi using OpenMP.

#include <iostream>

#include <omp.h>

#include <cstdlib>

#include <ctime>

using namespace std;

int main() {

long long N = 100000000;

long long inside\_seq = 0, inside\_par = 0;

double pi\_seq, pi\_par;

double start, end;

srand(time(0));

// ---------------- SEQUENTIAL ----------------

start = omp\_get\_wtime();

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

double x = (double)rand() / RAND\_MAX;

double y = (double)rand() / RAND\_MAX;

if (x \* x + y \* y <= 1.0) inside\_seq++;

}

pi\_seq = 4.0 \* inside\_seq / (double)N;

end = omp\_get\_wtime();

double time\_seq = end - start;

cout << "Sequential Pi = " << pi\_seq

<< " | Time = " << time\_seq << " sec" << endl;

// ---------------- PARALLEL ----------------

start = omp\_get\_wtime();

#pragma omp parallel

{

unsigned int seed = omp\_get\_thread\_num() + time(0); // thread-private seed

long long local\_count = 0;

#pragma omp for

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

double x = (double)rand\_r(&seed) / RAND\_MAX;

double y = (double)rand\_r(&seed) / RAND\_MAX;

if (x \* x + y \* y <= 1.0) local\_count++;

}

#pragma omp atomic

inside\_par += local\_count;

}

pi\_par = 4.0 \* inside\_par / (double)N;

end = omp\_get\_wtime();

double time\_par = end - start;

cout << "Parallel Pi = " << pi\_par

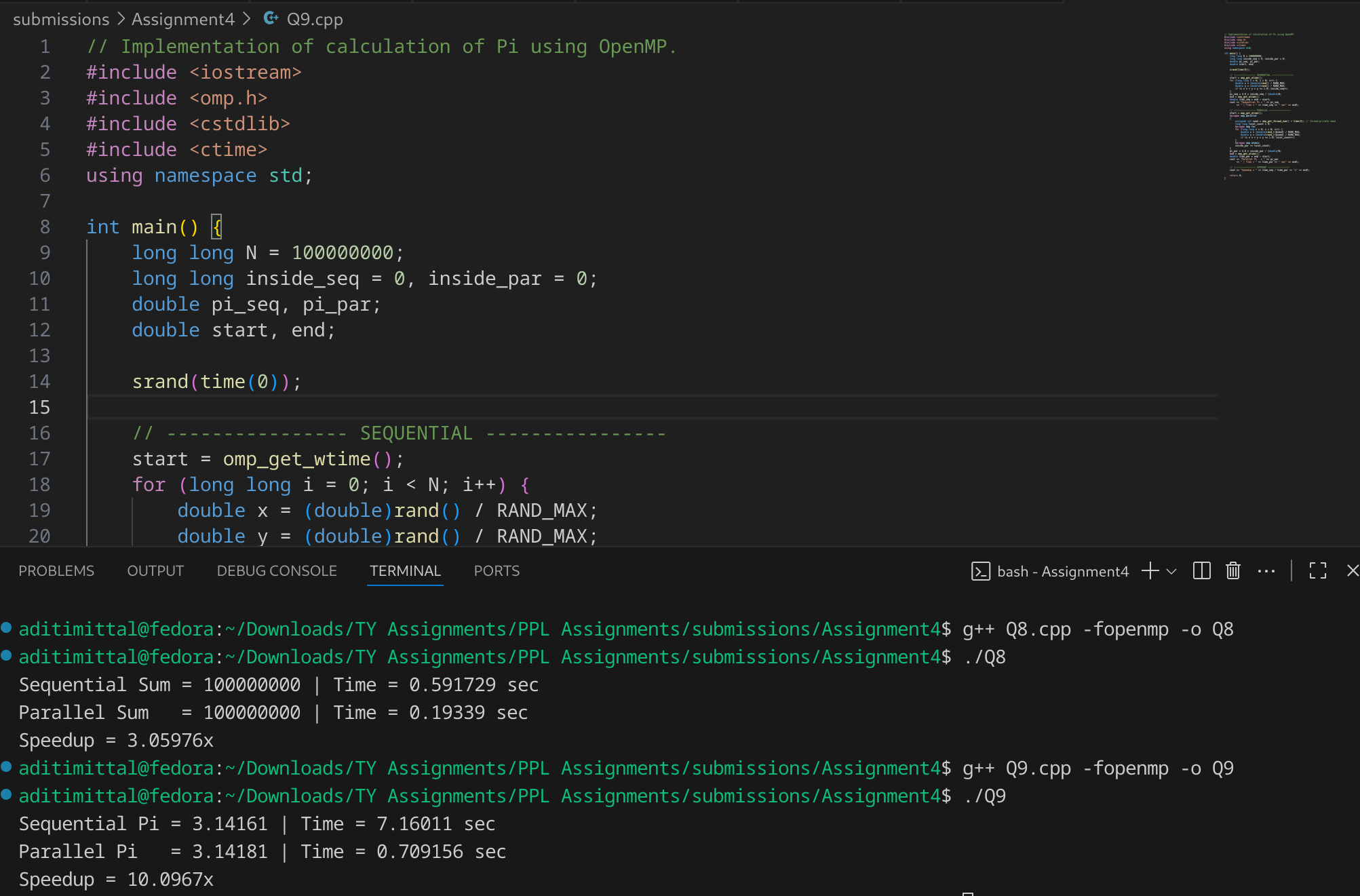
<< " | Time = " << time\_par << " sec" << endl;

// ---------------- SPEEDUP ----------------

cout << "Speedup = " << time\_seq / time\_par << "x" << endl;

return 0;

}



**Information:**

**Monte Carlos Method**

### Pros

* Simple to implement.
* Parallelizes well since each thread independently generates points.

### Cons

* Less **accurate** than numerical integration for the same number of iterations.
* Needs **huge N** (hundreds of millions) to converge near 3.14159.
* Random number generation overhead may limit scaling.

**Analysis:**

### 1. Sequential Version

* Each random point is generated and checked inside a single loop.
* Only **one CPU core** is used.
* Runtime grows linearly with number of iterations NNN.

### 2. Parallel Version (OpenMP)

* Workload is divided among multiple threads using #pragma omp parallel for.
* Each thread:  
  + Generates its own random points.
  + Keeps a private counter.
* Final counts are combined using #pragma omp atomic.
* Uses **all available CPU cores**.

**Performance**:

* Sequential: Slower, CPU-bound.
* Parallel: Significant improvement with multiple cores.

**Scalability**: Nearly linear speedup, but limited by:

* Random number generation overhead.
* Synchronization (atomic) when updating shared counter.

Sequential Monte Carlo is straightforward but inefficient for large NNN.

Parallel OpenMP implementation reduces runtime dramatically.

Speedup depends on number of threads and system architecture.

Monte Carlo converges slowly to π (needs huge NNN), but is an excellent example of embarrassingly parallel computation.

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