AY 2025-26 Odd Sem

Course: Cutting Edge Technologies Lab

Course Code: 7CS352

Practical No 1

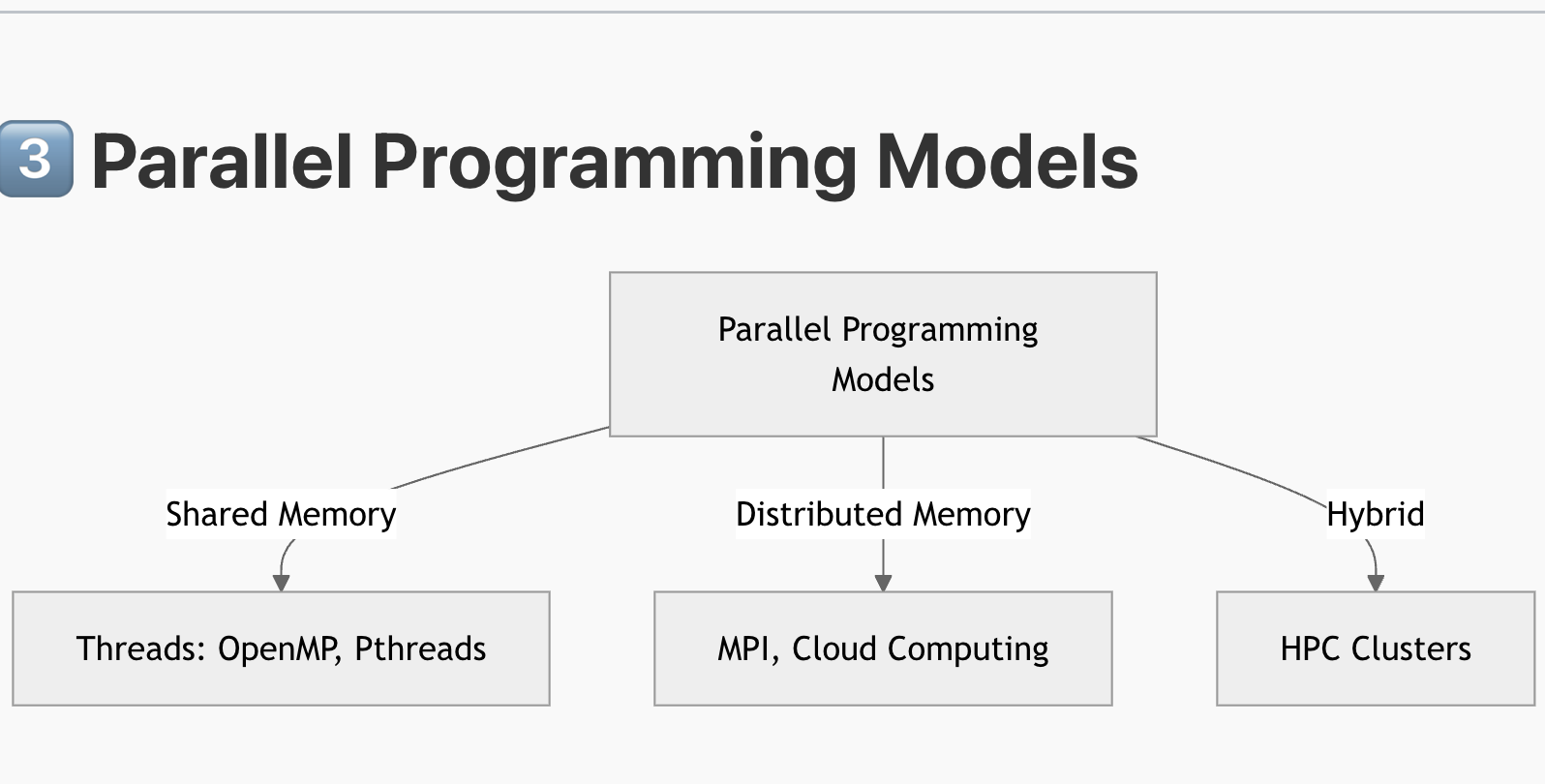
PRN: 23610067

Name: Aditi Mittal

Batch: T1

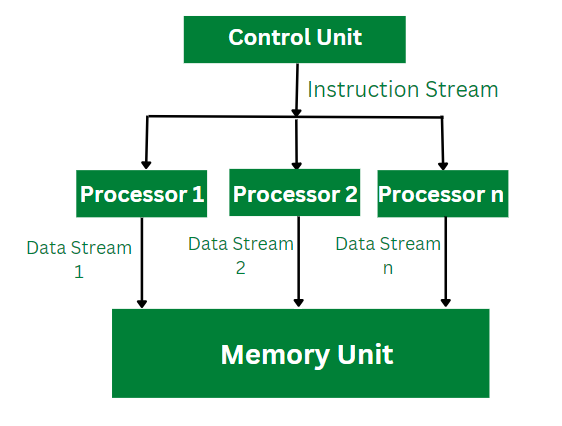
Title: Introduction to parallel programming, OpenMP installation, simple “Hello World” programs.

Problem Statement 1: Illustrate different types of Parallel Programming Models(Shared memory, distributed memory, accelerated computing)



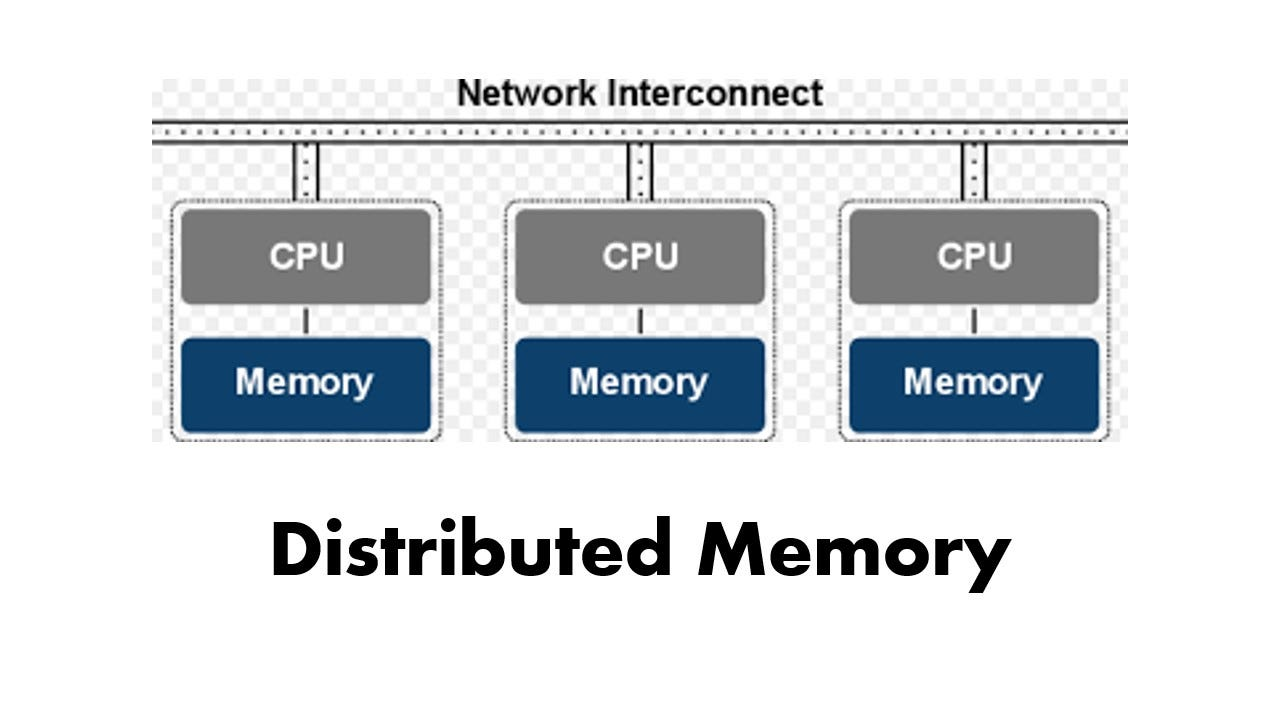
### 1. Shared Memory

* All processors share a single global address space.
* Threads can directly read/write shared variables without explicit message passing. Synchronization tools like locks, barriers, or atomic operations manage coordination.
* Easier programming and data communication.
* Limited scalability; requires careful handling of data races.
* OpenMP, POSIX threads.



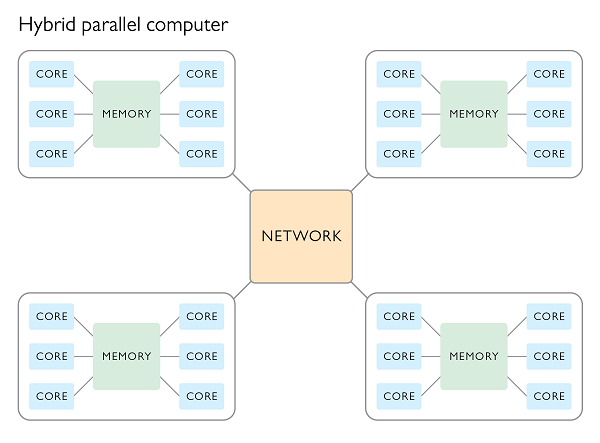
### 2. Distributed Memory

* Each processor operates with its **own local memory**. Data sharing happens explicitly through message passing over a network.
* Highly scalable (suited to clusters and supercomputers).
* Programming complexity increases with explicit communication management.
* MPI (Message Passing Interface).



### 3. Accelerated Computing (Hybrid / GPU-based)

* Combines traditional CPU-based shared memory or distributed memory models with accelerators like GPUs.
* Offload compute-intensive tasks to powerful GPUs while CPUs manage control flow and I/O.
* Massive parallel throughput for data-parallel jobs.
* **Cons**: Requires specialized knowledge and careful memory management across CPU–GPU boundaries.
* CUDA, OpenMP target directives, hybrid OpenMP + MPI.



Problem Statement 1 – Demonstrate Installation and Running of OpenMP code in C

Recommended Linux based System:

Following steps are for windows:

OpenMP – Open Multi-Processing is an API that supports multi-platform shared-memory multiprocessing programming in C, C++ and Fortran on multiple OS. OpenMP uses a portable, scalable model that gives programmers a simple and flexible interface for developing parallel applications for platforms ranging from the standard desktop computer to the supercomputer.

To set up OpenMP,

We need to first install C, C++ compiler if not already done. This is possible through the MinGW Installer.  
Reference: Article on GCC and G++ installer ([Link](https://www.scaler.com/topics/c/c-compiler-for-windows/))

Note: Also install `mingw32-pthreads-w32` package.

Then, to run a program in OpenMP, we have to pass a flag `-fopenmp`.

Example:

To run a basic Hello World,

*#include* <stdio.h>

*#include* <omp.h>

*int* main(*void*)

{

*#pragma* *omp* *parallel*

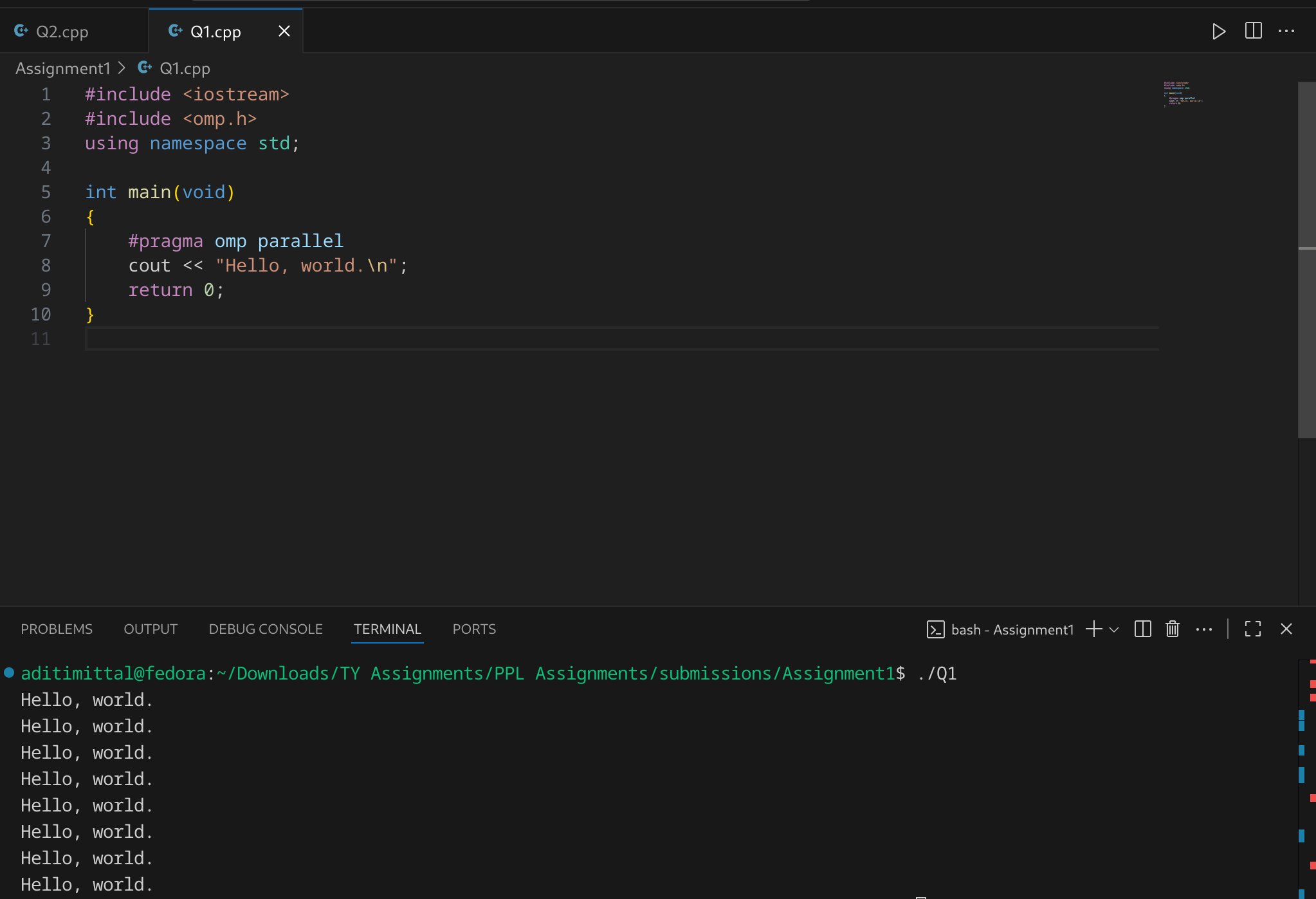
    printf("Hello, world.\n");

*return* 0;

}

gcc -fopenmp test.c -o hello

.\hello.exe



Problem Statement 2 – Print ‘Hello, World’ in Sequential and Parallel in OpenMP

We first ask the user for number of threads – OpenMP allows to set the threads at runtime. Then, we print the Hello, World in sequential – number of times of threads count and then run the code in parallel in each thread.

Code snapshot:

#include <iostream>

#include <omp.h>

using namespace std;

int main(){

int num\_of\_threads;

cout<< "Enter number of threads: ";

cin >> num\_of\_threads;

cout << "------------------Sequential--------------------------";

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

cout<<"Hello World" << endl;

}

cout << "------------------Parallel--------------------------";

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

#pragma omp parallel

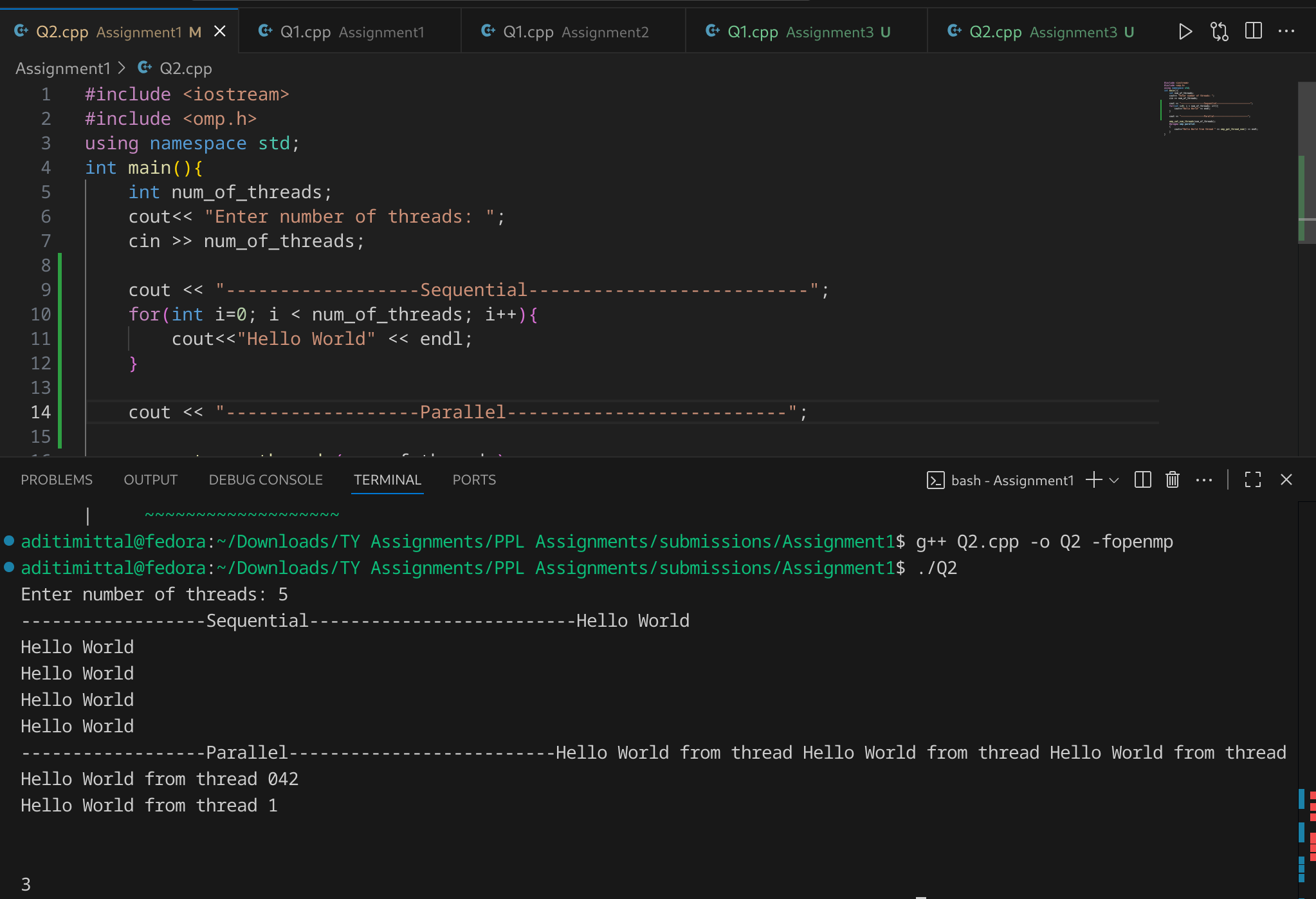
{

cout<<"Hello World from thread " << omp\_get\_thread\_num() << endl;

}

}

Output snapshot:



Analysis:

GitHub Link: make a public repository upload code of an assignment and paste its link here.

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

Problem statement 3: Calculate theoretical FLOPS of your system on which you are running the above codes.

// nproc = 8

// lscpu | grep "Mhz"

// CPU(s) scaling MHz: 22%

// CPU max MHz: 4400.0000

// CPU min MHz: 400.0000

#include <iostream>

#include <omp.h>

#include <vector>

using namespace std;

int main(){

int N = 1e8;

vector <double> A(N,1.5);

vector <double> B(N,2.5);

vector <double> C(N,1.6);

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

#pragma parallel for

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

C[i] = (A[i] + B[i]) \*B[i];

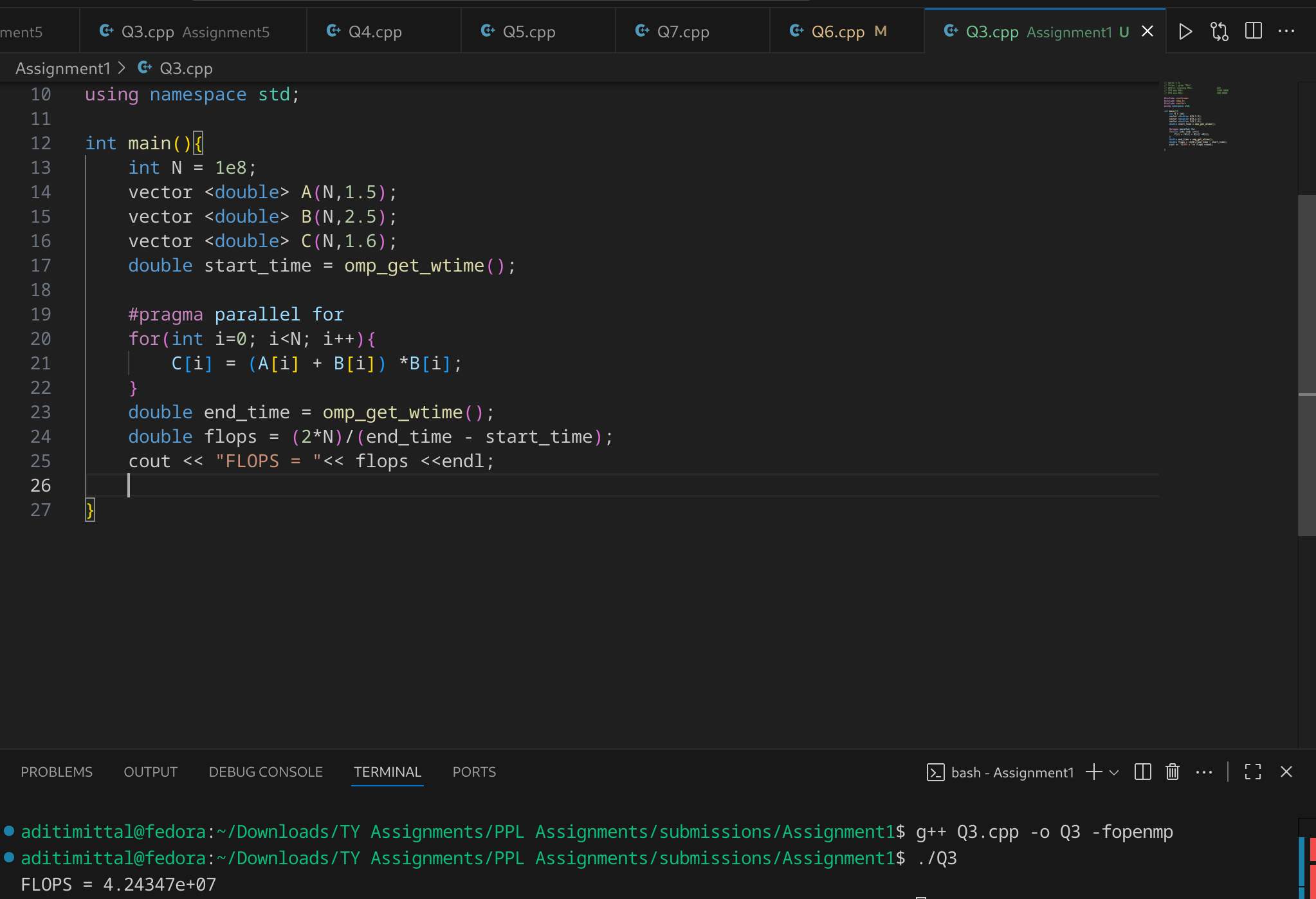
}

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

double flops = (2\*N)/(end\_time - start\_time);

cout << "FLOPS = "<< flops <<endl;

}



Elaborate the parameters and show calculation.

Flops: Floating Point Operations per Second.

So we perform inside a parallel for loop, 3 floating point operations per thread ( 1 multiplication, 2 additions). We calculate the time needed for 10^8 such events, i.e. 2\*10^8 operations and then calculate the flops as 3\*10^8/time taken to complete these flops.

We thus get the answer as 9.13 x 10^8 i.e. it performs 91 million floating point operations per second