Course: High Performance Computing Lab

Practical No 1

PRN: 21510074

Name: Yash Rajesh Nawale

Batch: B4

Title: Introduction to OpenMP

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

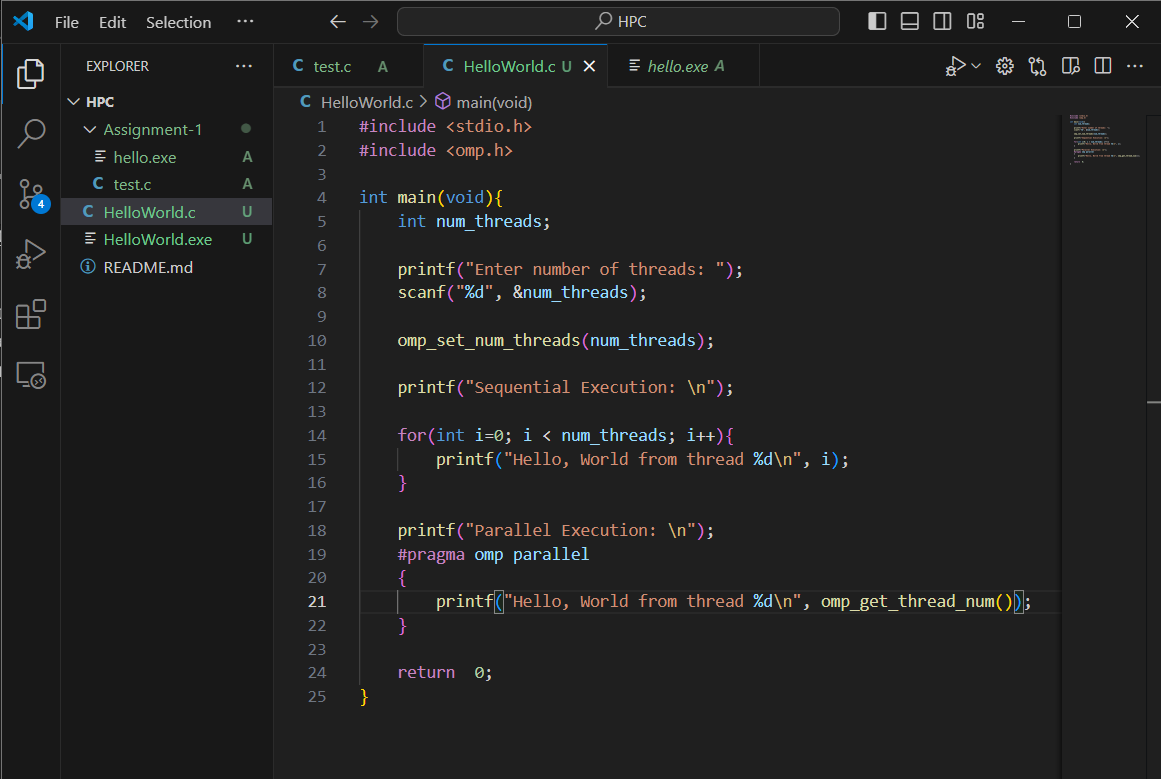
.\hello.exe



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:



Output snapshot:

A screenshot of a computer

Description automatically generated

Analysis:

This code demonstrates the use of OpenMP to compare sequential and parallel execution in C. It asks the user for the number of threads, prints "Hello, World" sequentially for each thread, and then executes a parallel block where each thread prints a message concurrently. You can see in the parallel execution the threads are executes in a random order on the basis of whichever threads gets executed first. This highlights basic thread management and the non-deterministic nature of parallel output.

GitHub Link: <https://github.com/YashNawale26/High-Performance-Computing>

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

Elaborate the parameters and show calculation.

FLOPS : Floating Point Operations Per Second

FLOPS=Clock Speed (Hz)×Number of Cores×FLOPs per Cycle per FPU×Number of FPUs per Core

Where,

**Clock Speed (GHz)**: The speed at which the CPU operates, typically measured in gigahertz (GHz).

**Number of Cores**: The total number of CPU cores in your system.

**Number of Floating Point Units (FPUs) per Core**: Most modern CPUs have at least one FPU per core, but high-performance processors might have more.

**FLOPs per Cycle per FPU**: This is the number of floating-point operations that can be performed per cycle by a single FPU. For example, a typical FPU might handle 2 floating-point operations per cycle (e.g., one addition and one multiplication).

Clock Speed: 3.2 GHz (Base Clock) = 3.2×109Hz

Number of Cores: 8

Number of FPUs per Core: 1

FLOPs per Cycle per FPU: 2

**FLOPS per Core:** Clock Speed × FLOPs per Cycle per FPU FLOPS per Core

= 3.2 x (10^9) Hz x 2

= 6.4 x (10^9) FLOPS

Totals Flops = Flops per Core x Number of Cores

= 6.4 x (10^9) x 8

= 51.2 GFLOPS