Course: High Performance Computing Lab

Practical No 1

PRN: 22510110

Name: Mohammed Hamza Ejaz Ahmed Shaikh

Batch: B2

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

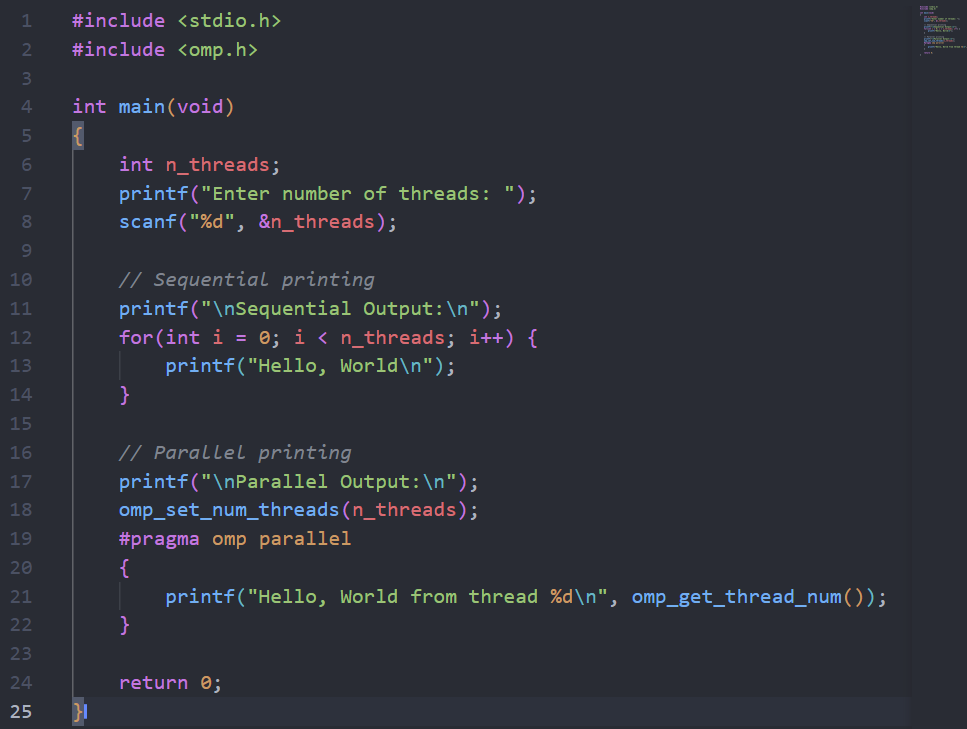
.\hello.exe



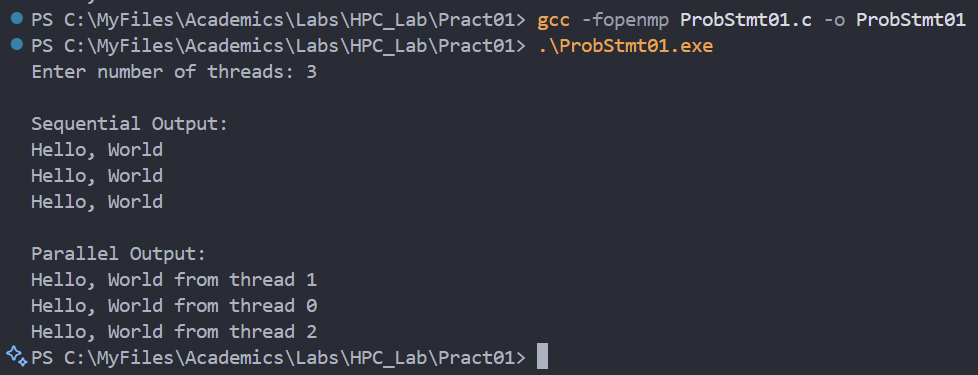
Problem Statement 2 – Print ‘Hello, World’ in Sequential and Parallel in OpenMP

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:



Analysis:

* **Sequential Section:**  
  The message is printed one after another, exactly n\_threads times, using a simple loop.
* **Parallel Section:**  
  OpenMP creates n\_threads threads. Each thread executes the code inside the parallel block and prints the message, including its thread number (using omp\_get\_thread\_num()).
* **OpenMP Features Used:**
  + omp\_set\_num\_threads(n\_threads): Sets the number of threads for parallel execution.
  + #pragma omp parallel: Creates a parallel region.
  + omp\_get\_thread\_num(): Returns the thread’s unique ID.
* **Comparison:**  
  Sequential printing is predictable and ordered. Parallel printing is faster for large workloads but the output order may not be sequential due to concurrent execution.

GitHub Link: https://github.com/hamzask018/HPC\_Lab/tree/main/Pract01

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

**Elaborate the parameters and show calculation**

**Parameters for Intel Core Ultra 7 155H**

* **Number of Performance Cores:** 6
* **Number of Efficient Cores:** 8
* **Number of Threads:** 22
* **Max Turbo Frequency:** ~4.8 GHz (Performance cores)
* **SIMD Width:** Supports AVX2 (256-bit), so each core can do **8 single-precision (float) FLOPs per cycle**.

FLOPS = Number of cores × Clock speed × FLOPs per cycle × 10^9

= 6 × 4.8 × 8 × 10^9d

= 230.4 × 10^9

= 230.4 GFLOPS (GigaFLOPS)

If Include Efficiency Core:

FLOPS (Efficient) = 8 × 3.8 × 1 × 10^9 = 30.4 GFLOPS

Total Theoretical FLOPS:

Total FLOPS = 230.4 GFLOPS + 30.4 GFLOPS = 260.8 GFLOPS