Course: High Performance Computing Lab

Practical No 1

PRN: 22510078

Name: Piyush Krishna Jadhav

Batch: B5

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

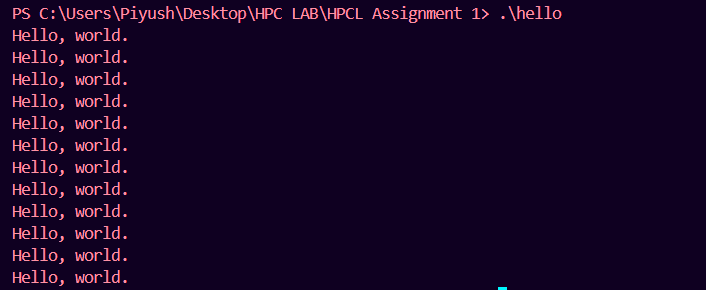
    printf("Hello, world.\n");

    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:

#include <stdio.h>

#include <omp.h>

int main() {

    int n\_threads;

    printf("Enter number of threads: ");

    scanf("%d", &n\_threads);

    printf("\nSequential Hello World:\n");

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

        printf("Hello from thread %d (Sequential)\n", i);

    }

    printf("\nParallel Hello World using OpenMP:\n");

    omp\_set\_num\_threads(n\_threads);

    #pragma omp parallel

    {

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

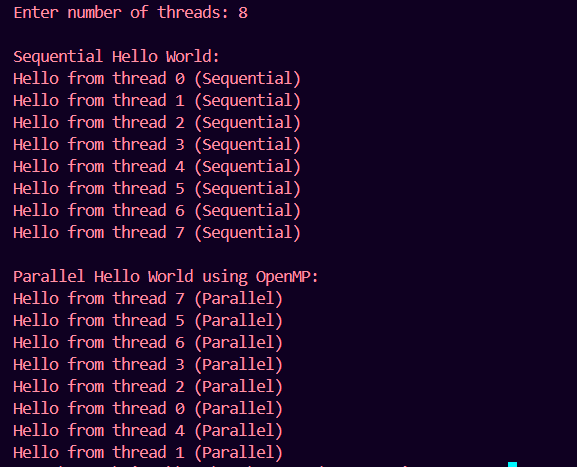
        printf("Hello from thread %d (Parallel)\n", tid);

    }

    return 0;

}

Output snapshot:



Analysis:

* **Objective:** Demonstrates difference between **sequential** and **parallel** execution using OpenMP.
* **Sequential Part:** Prints "Hello" messages in order (single thread loop).
* **Parallel Part:** Uses #pragma omp parallel with user-defined number of threads. Each thread prints its ID independently.
* **Key Point:** Shows how OpenMP distributes work across threads, where outputs may appear unordered due to parallel execution.

GitHub Link: <https://github.com/PiyushJadhav06044556/HPC-LAB-7th-Sem>

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

The formula for theoretical peak FLOPS is: Theoretical FLOPS=ClockSpeed(Hz)×Cores×OperationsPerCycle×InstructionLevelParallelism

* **Clock Speed (Hz)**: The processor's clock speed, typically measured in gigahertz (GHz), which is the number of cycles per second.
* **Cores**: The number of individual processing cores on the CPU or GPU.
* **Operations Per Cycle**: The number of floating-point operations a single core can perform in one clock cycle. This is determined by the processor's instruction set architecture (like AVX or AVX-512) and the data type (e.g., FP32 or FP64). For instance, an AVX-512 instruction can perform 16 single-precision (FP32) floating-point operations at once.
* **Instruction Level Parallelism**: The number of parallel operations (e.g., a fused multiply-add, or FMA, which combines a multiplication and an addition) that can be executed in a single cycle. A single FMA instruction is counted as two FLOPS.

FLOPS=6cores×(4.5×10^9Hz)×32operations/cycle FLOPS=864×109FLOPS FLOPS=864GFLOPS