Course: High Performance Computing Lab

Practical No 1

**PRN**: 21510067

**Name**: Vardhan Vivek Kulkarni

**Batch**: B-3

Title: Introduction to OpenMP

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

Recommended Linux based System:

* Install GCC: Ensure GCC is installed (includes OpenMP support).
* Write Code: Create a simple OpenMP C program.
* Compile: Use gcc -fopenmp to compile.
* Run: Execute the compiled program.

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*

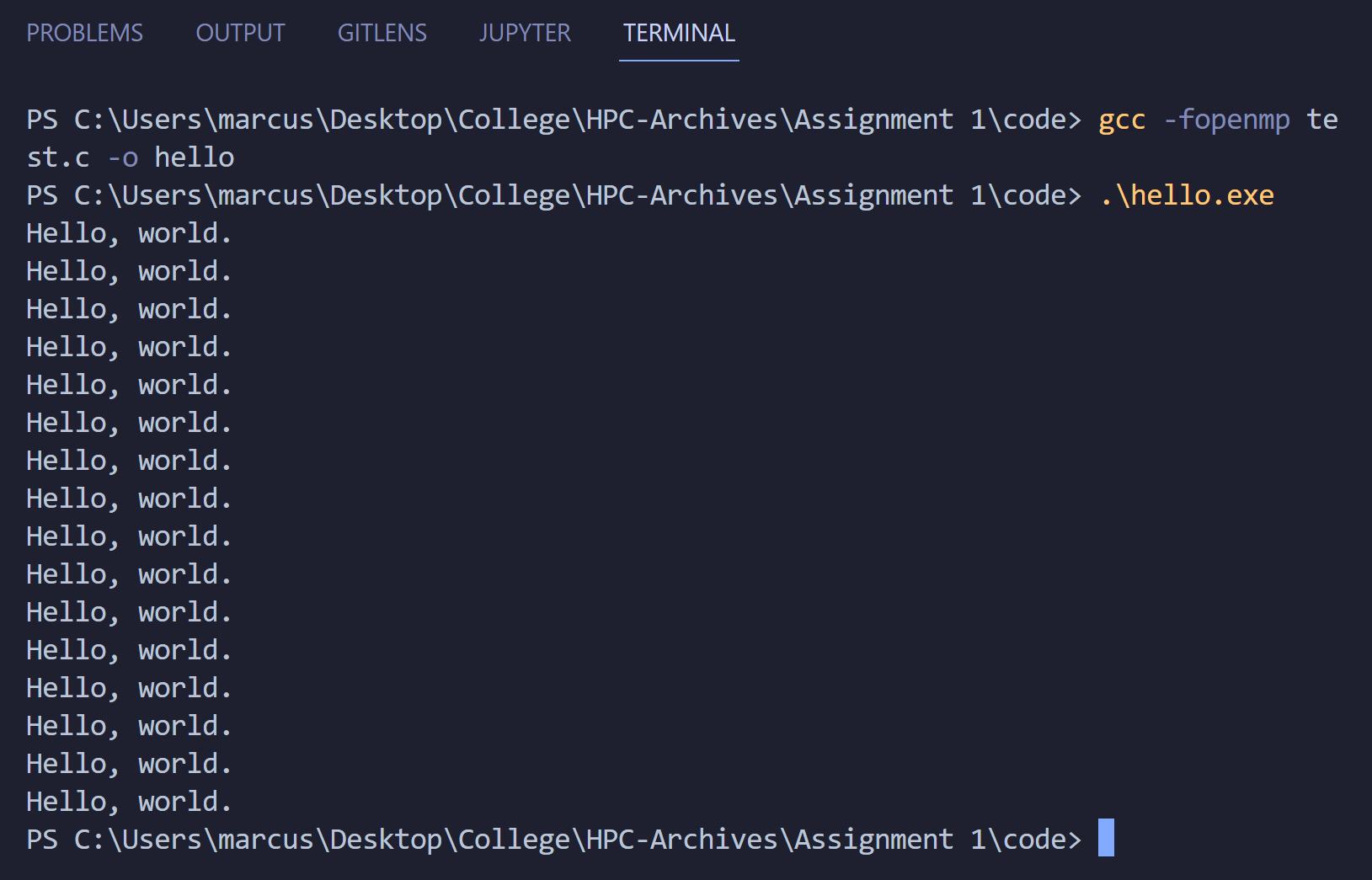
    printf("Hello, world.\n");

*return* 0;

}

gcc -fopenmp test.c -o hello

.\hello.exe



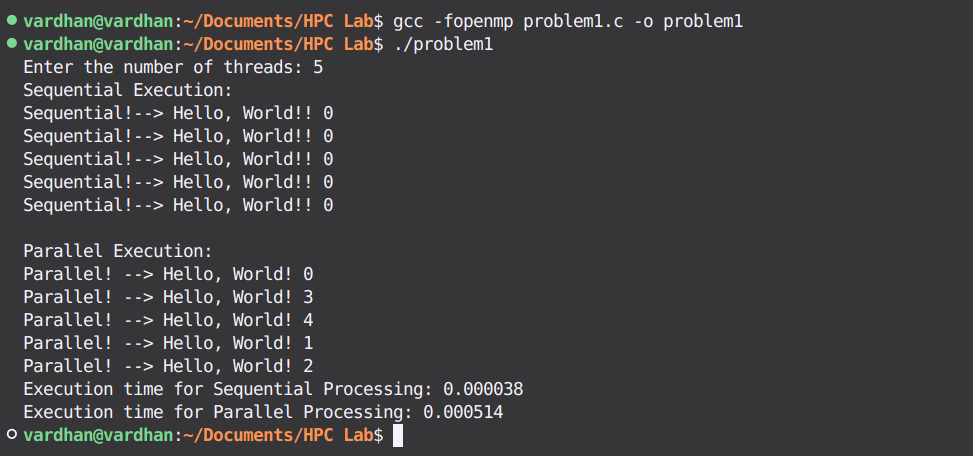
Problem Statement 2 – Print ‘Hello, World’ in Sequential and Parallel in OpenMP

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:

1. Sequential Execution:

* The code begins by asking the user to input the number of threads. However, in the sequential section, the code runs in a single thread, and the thread number remains constant (0). This section simulates a task running without any parallelism, showcasing how each iteration is executed one after the other.

2. Parallel Execution:

* The program then sets the number of threads as specified by the user and enters a parallel region. Here, the task of printing "Hello, World!" is distributed among multiple threads. Each thread executes its portion of the work simultaneously, potentially reducing the overall execution time.
* omp\_get\_thread\_num() is used to identify the thread number, demonstrating how OpenMP assigns different portions of work to different threads.

3. Performance Measurement:

* The code measures the execution time for both the sequential and parallel sections using omp\_get\_wtime(). This allows for a direct comparison of the performance difference between the two approaches.
* Typically, parallel execution should be faster for a large number of threads, but this also depends on the overhead of managing multiple threads and the nature of the task.

GitHub Link: [Practical 1 Repository](https://github.com/git8vardhan/HPCL/tree/master/practical1)

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

**We need the following to calculate FLOPs of our system:**

* CPU Clock Speed: The frequency at which the CPU operates (in GHz or MHz).
* Number of Cores: The number of processing cores in your CPU.
* FLOP per Cycle per Core: The number of floating-point operations a core can perform per clock cycle. This depends on the CPU architecture.

### Calculation:

* Clock Speed: 4.2 GHz (that is 4.2×109 Hz)
* Number of Cores: 8
* FLOP per Cycle per Core: 4 (double precision)

### Formula for Theoretical FLOPS:

The theoretical FLOPS can be calculated using the formula:

**FLOPS = Clock Speed \* Number of Cores \* FLOPs per cycle per core**

**= 4.2** ×109 Hz × 8 Cores × 4 FLOP per Cycle per Core

= **134.4 GFLOPS**