Course: High Performance Computing Lab

Practical No 1

PRN: 21510013

Name: Bhupendra Baliram Jambhale

Batch: B1

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*

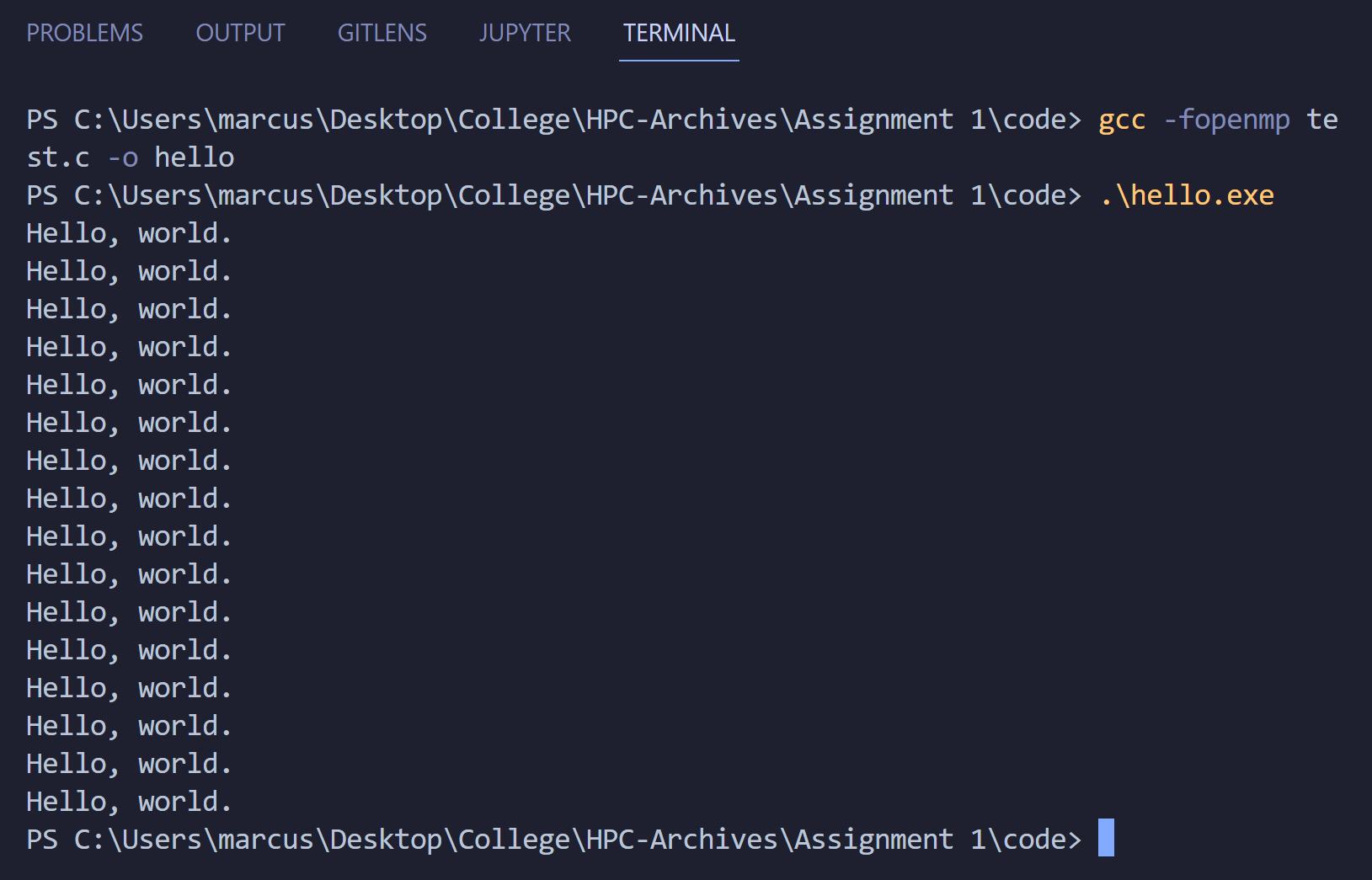
    printf("Hello, world.\n");

*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 :

**#include** <stdio.h>

**#include** <omp.h>

**int** main()

{

**int** num\_threads;

**double** start\_time, end\_time, sequential\_time, parallel\_time;

       printf("Enter the number of threads: ");

    scanf("%d", **&**num\_threads);

*// Sequential Execution*

    printf("Sequential execution:\n");

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

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

    {

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

    }

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

    sequential\_time **=** end\_time **-** start\_time;

    printf("Time taken for sequential execution: %f seconds\n", sequential\_time);

*// Parallel Execution*

    printf("\nParallel execution:\n");

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

**#pragma** **omp** **parallel** **num\_threads**(**num\_threads**)

    {

**int** thread\_id **=** omp\_get\_thread\_num();

        printf("Hello, World from thread %d\n", thread\_id);

    }

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

    parallel\_time **=** end\_time **-** start\_time;

    printf("Time taken for parallel execution: %f seconds\n", parallel\_time);

*// Analysis*

**double** speedup **=** sequential\_time **/** parallel\_time;

**double** efficiency **=** speedup **/** num\_threads;

    printf("\nAnalysis:\n");

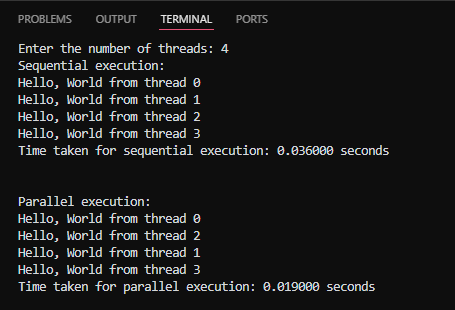
    printf("Speedup: %f\n", speedup);

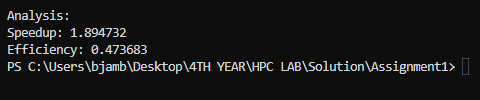
    printf("Efficiency: %f\n", efficiency);

**return** 0;

}

Output snapshot:





Analysis:

**1. Execution Time:**

* **Sequential Time:** 0.036 seconds
* **Parallel Time:** 0.019 seconds

The parallel version is faster because it utilizes multiple threads, allowing tasks to be executed simultaneously. This reduces the overall execution time compared to the sequential version, where each task runs one after the other.

**2. Speedup:**

* **Calculated Speedup:** 1.89

The speedup shows that the parallel execution is nearly twice as fast as the sequential one. However, it's not exactly double due to the overhead involved in managing threads.

**3. Efficiency:**

* **Calculated Efficiency:** 0.47

With an efficiency of 47%, the parallel execution makes good use of the available threads, though it's not fully optimized. This is expected in parallel computing, where not all tasks can be perfectly parallelized, and some overhead is inevitable.

**4. Execution Order:**

* In the parallel output, the order of "Hello, World" prints isn't sequential. This is typical in parallel processing, where thread execution order can vary depending on how the operating system schedules them.

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

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

Elaborate the parameters and show calculation.

CPU specifications:

* CPU Clock Speed: 2.00 GHz (2.00 x 10^9 cycles per second)
* Number of Cores: 4
* Number of Logical Processors (Threads): 8

To calculate the theoretical FLOPS, we'll assume:

* FLOPs per Cycle per Core: 2 (for single precision, which is typical for modern CPUs)
* Instructions per Cycle (IPC): Typically 1 (we'll assume 1 unless the architecture supports multiple instructions per cycle, which some modern CPUs do).

**Calculation Steps:**

1. **Determine FLOPS per Core:** FLOPS per Core = Clock Speed × FLOPs per Cycle per Core  
   FLOPS per Core = 2.00 x 10^9 × 2 = 4.00 x 10^9 FLOPS (or 4 GFLOPS)
2. **Calculate FLOPS for All Cores:** Total FLOPS = FLOPS per Core × Number of Cores  
   Total FLOPS = 4.00 x 10^9 × 4 = 16.00 x 10^9 FLOPS (or 16 GFLOPS)