

Course: High Performance Computing Lab

## Practical No 1

PRN: 23520012

Name: Bhakti Sharad More

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](#))

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,

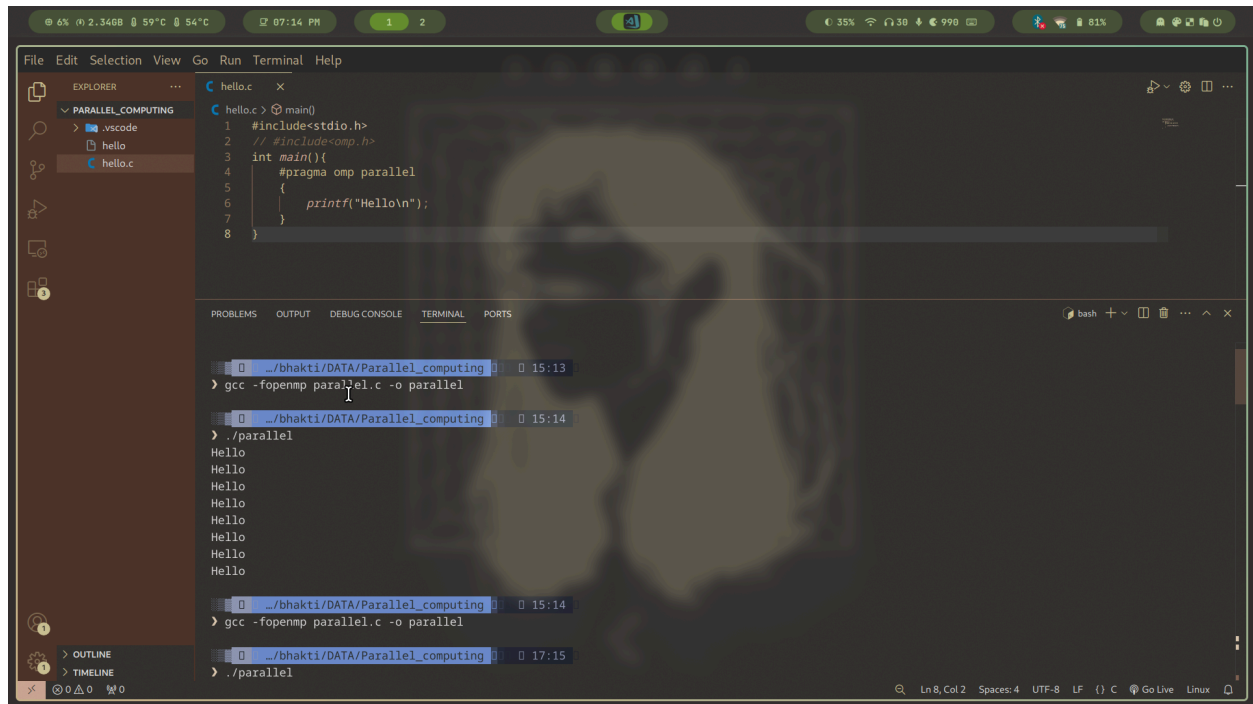
```
#include <stdio.h>
```

```
#include <omp.h>
```

```
int main(void)
{
    #pragma omp parallel
    printf("Hello, world.\n");
    return 0;
}
```

```
gcc -fopenmp test.c -o hello
.\hello.exe
```

Code/ Output snapshot:



The screenshot shows a Visual Studio Code editor window with a C file named `hello.c`. The code is as follows:

```
1 hello.c > main()
2 #include<stdio.h>
3 // #include<omp.h>
4 int main(){
5     #pragma omp parallel
6     {
7         printf("Hello\n");
8     }
9 }
```

The terminal at the bottom shows the compilation and execution of the program:

```
0 ./bhakti/DATA/Parallel_computing 0 15:13
> gcc -fopenmp parallel.c -o parallel
0 ./bhakti/DATA/Parallel_computing 0 15:14
> ./parallel
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
0 ./bhakti/DATA/Parallel_computing 0 15:14
> gcc -fopenmp parallel.c -o parallel
0 ./bhakti/DATA/Parallel_computing 0 17:15
> ./parallel
```

Analysis:

### Step-by-step Explanation:

#### 1. `#include <stdio.h>`

- Includes the standard input/output library so we can use `printf()`.

#### 2. `#include <omp.h>`

- Includes the OpenMP library, which allows you to write parallel programs using simple `#pragma` commands.

#### 3. `#pragma omp parallel`

- This is an OpenMP *directive*.

- It tells the compiler:  
“Run the next block of code **in parallel**, using multiple threads.”

4. **printf("Hello, world.\n");**

- Each thread **prints** this line.

5. **So what happens?**

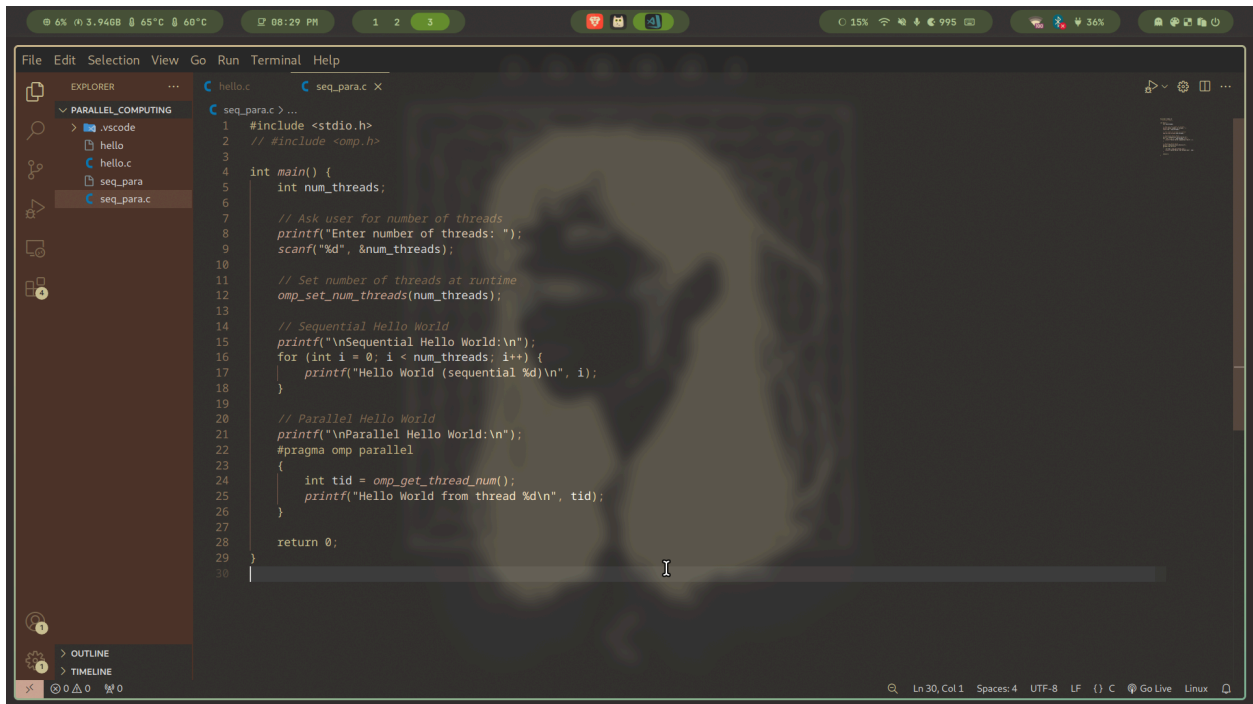
- If your system runs with 4 threads, you will see the line **printed 4 times** (one from each thread).
- The order of printing may look random (because threads run independently).

GitHub Link: [https://github.com/bhaktimore18/hpc\\_lab](https://github.com/bhaktimore18/hpc_lab)

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

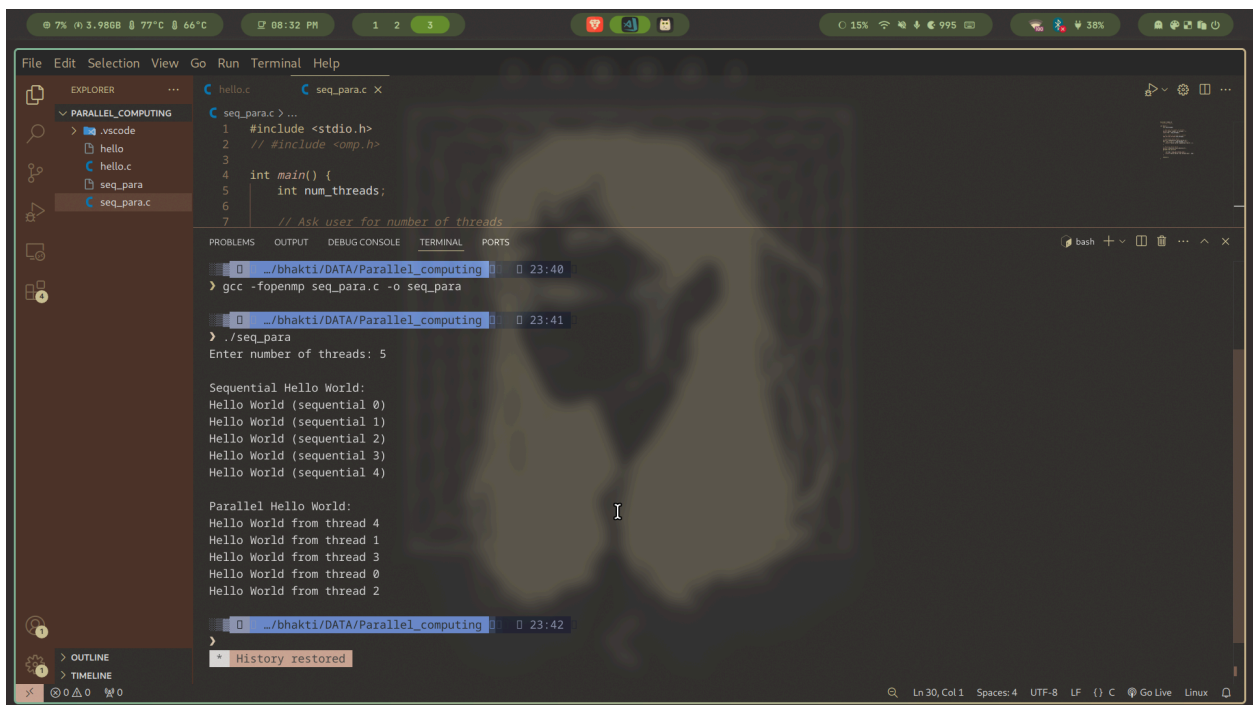
We first ask the user for number of threads – OpenMP allows to set the threads at runtime. Then, we print the Hello, World in sequential – number of times of threads count and then run the code in parallel in each thread.

Code Snapshots:



```
1 #include <stdio.h>
2 // #include <omp.h>
3
4 int main() {
5     int num_threads;
6
7     // Ask user for number of threads
8     printf("Enter number of threads: ");
9     scanf("%d", &num_threads);
10
11     // Set number of threads at runtime
12     omp_set_num_threads(num_threads);
13
14     // Sequential Hello World
15     printf("\nSequential Hello World:\n");
16     for (int i = 0; i < num_threads; i++) {
17         printf("Hello World (sequential %d)\n", i);
18     }
19
20     // Parallel Hello World
21     printf("\nParallel Hello World:\n");
22     #pragma omp parallel
23     {
24         int tid = omp_get_thread_num();
25         printf("Hello World from thread %d\n", tid);
26     }
27
28     return 0;
29 }
30
```

Output snapshots:



```
0 ./bhakti/DATA/Parallel_computing : 0 23:40
> gcc -fopenmp seq_para.c -o seq_para

0 ./bhakti/DATA/Parallel_computing : 0 23:41
> ./seq_para
Enter number of threads: 5

Sequential Hello World:
Hello World (sequential 0)
Hello World (sequential 1)
Hello World (sequential 2)
Hello World (sequential 3)
Hello World (sequential 4)

Parallel Hello World:
Hello World from thread 4
Hello World from thread 1
Hello World from thread 3
Hello World from thread 0
Hello World from thread 2

0 ./bhakti/DATA/Parallel_computing : 0 23:42
>
* History restored
```

Analysis:

### What this program does:

1. **It asks the user** how many threads they want to use.
  2. It tells OpenMP to use that many threads when running parallel code.
  3. Then it prints "Hello World" messages in two ways:
    - First, **sequentially** (one by one, using only the main thread).
    - Then, **in parallel** (many threads printing at the same time).
- 

### Explanation of Each Step:

- The program starts by asking for a number, for example 4.
  - It stores this number and tells OpenMP: "Use 4 threads."
  - It then prints "Hello World (sequential 0)", "Hello World (sequential 1)", and so on up to 3 — this part is **not parallel**, so the order is always correct.
  - Next, it enters a **parallel block**. OpenMP starts 4 threads.
  - Each thread runs the print statement and says "Hello World from thread X", where X is the thread's ID.
  - Since this runs in **parallel**, the order may change every time you run it.
- 

### Behind the scenes:

- `omp_set_num_threads()` sets how many threads OpenMP will use.
- `omp_get_thread_num()` gets the ID of each thread (like 0, 1, 2, etc).
- The parallel block tells all threads to run the same code.

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

**Theoretical FLOPS = Number of cores × Clock speed × FLOPs per cycle**

Assuming:

- Number of cores = 4
- Clock speed = 3.0 GHz =  $3 \times 10^9$  Hz
- FLOPs per cycle = 8

**FLOPS =  $4 \times 3 \times 10^9 \times 8 = 96 \times 10^9 = 96$  GFLOPS**

The theoretical peak performance of the system is **96 GFLOPS** (96 billion floating-point operations per second).

**Theoretical FLOPS** means how fast your CPU *can* do decimal (floating-point) math per second.

In this case:

- The CPU has **4 cores** (4 brains working together)
- Each core runs at **3.0 GHz** (3 billion steps per second)
- Each core can do **8 floating-point operations in one step**

So, in total:

**$4 \times 3 \text{ billion} \times 8 = 96 \text{ billion operations per second}$ ,**  
which is **96 GFLOPS**.