Practical No 1

PRN: 22520005

Name: Aftab Imtiyaj Bhadgaonkar

Batch: B6

Course: High Performance Computing Lab

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.

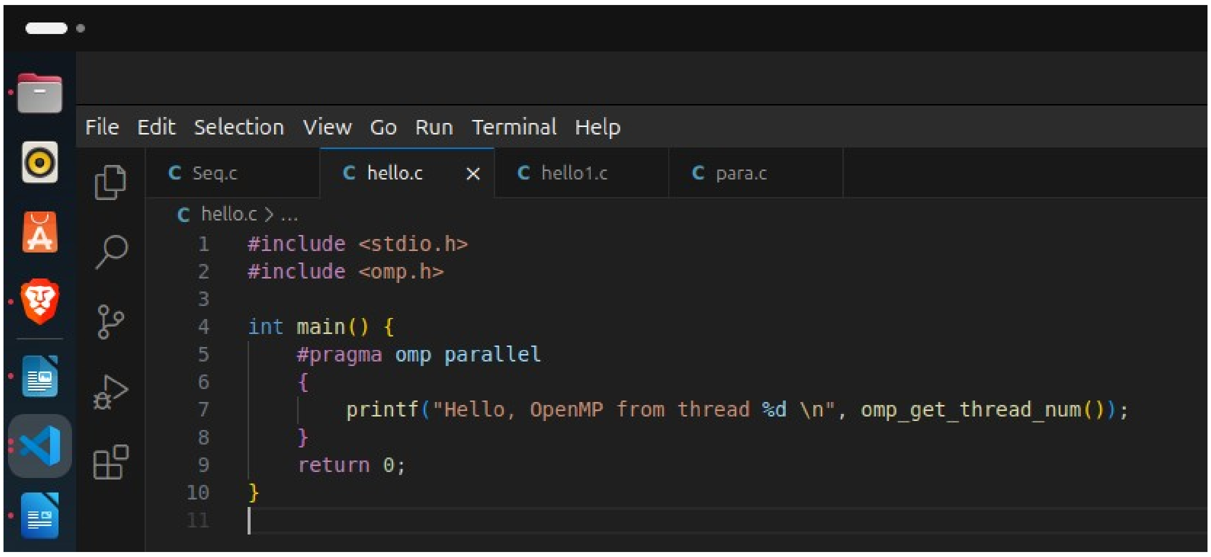
To set up OpenMP,

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.

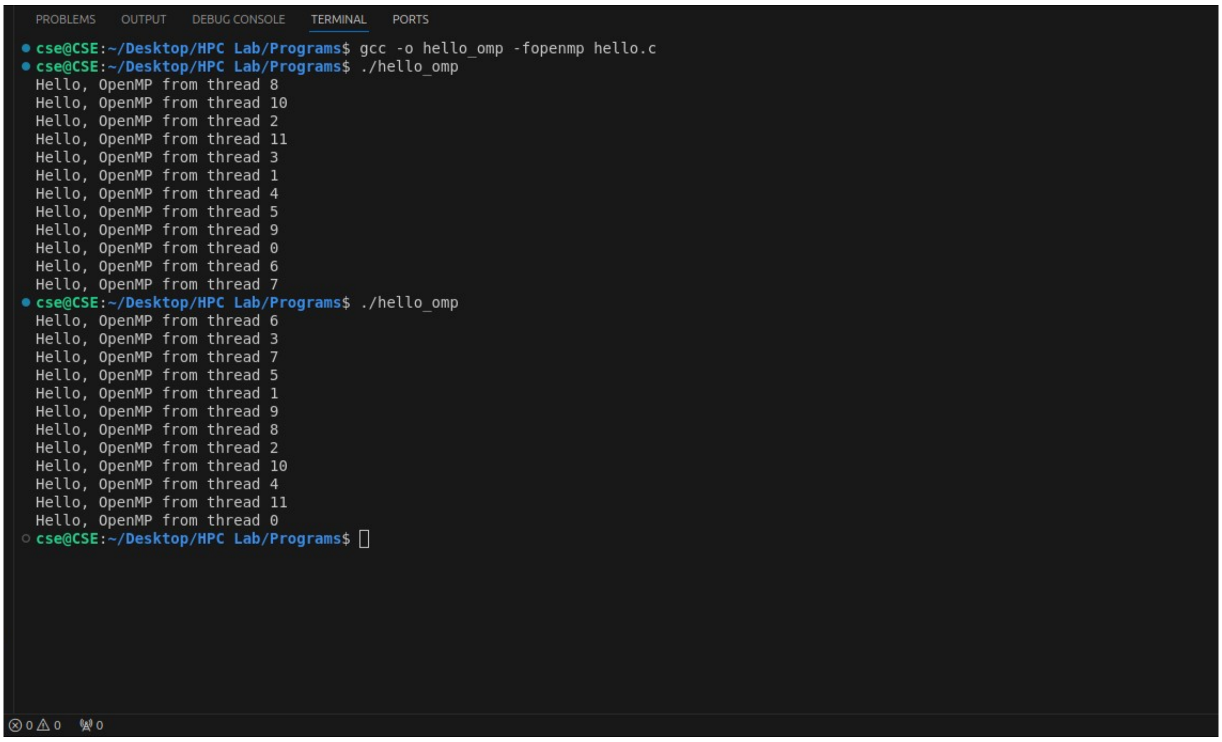
Then, to run a program in OpenMP, we have to pass a flag **`-fopenmp`.**

Code 1:



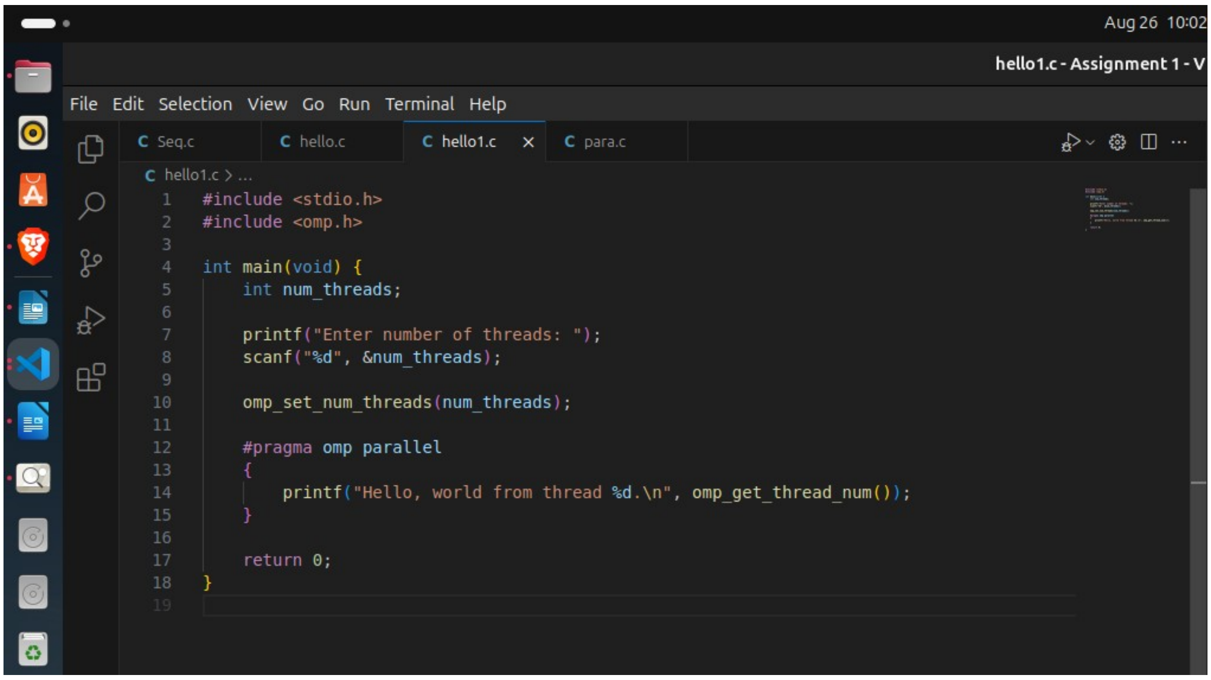
```c
#include <stdio.h>
#include <omp.h>

int main() {
    #pragma omp parallel
    {
        printf("Hello, OpenMP from thread %d \n", omp_get_thread_num());
    }
    return 0;
}
```

```
gcc -o hello_omp -fopenmp hello.c

.\hello_omp.exe
```

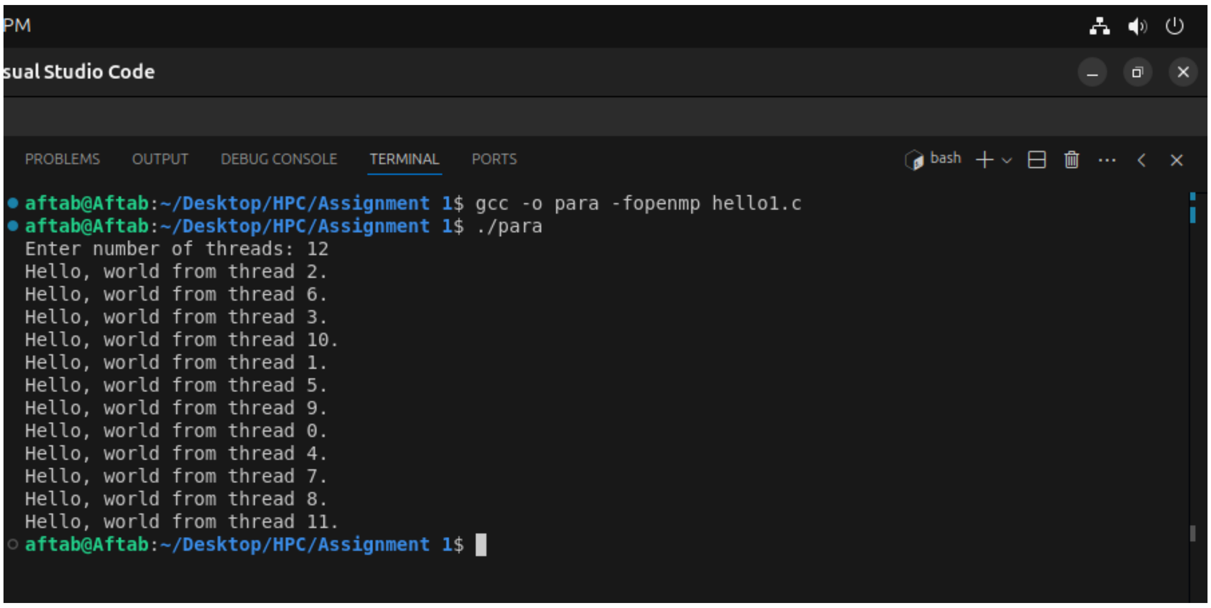


```
cse@CSE:~/Desktop/HPC Lab/Programs$ gcc -o hello_omp -fopenmp hello.c
cse@CSE:~/Desktop/HPC Lab/Programs$ ./hello_omp
Hello, OpenMP from thread 8
Hello, OpenMP from thread 10
Hello, OpenMP from thread 2
Hello, OpenMP from thread 11
Hello, OpenMP from thread 3
Hello, OpenMP from thread 1
Hello, OpenMP from thread 4
Hello, OpenMP from thread 5
Hello, OpenMP from thread 9
Hello, OpenMP from thread 0
Hello, OpenMP from thread 6
Hello, OpenMP from thread 7
cse@CSE:~/Desktop/HPC Lab/Programs$ ./hello_omp
Hello, OpenMP from thread 6
Hello, OpenMP from thread 3
Hello, OpenMP from thread 7
Hello, OpenMP from thread 5
Hello, OpenMP from thread 1
Hello, OpenMP from thread 9
Hello, OpenMP from thread 8
Hello, OpenMP from thread 2
Hello, OpenMP from thread 10
Hello, OpenMP from thread 4
Hello, OpenMP from thread 11
Hello, OpenMP from thread 0
cse@CSE:~/Desktop/HPC Lab/Programs$ 
```

Code 2:



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

int main(void) {
    int num_threads;

    printf("Enter number of threads: ");
    scanf("%d", &num_threads);

    omp_set_num_threads(num_threads);

    #pragma omp parallel
    {
        printf("Hello, world from thread %d.\n", omp_get_thread_num());
    }

    return 0;
}
```



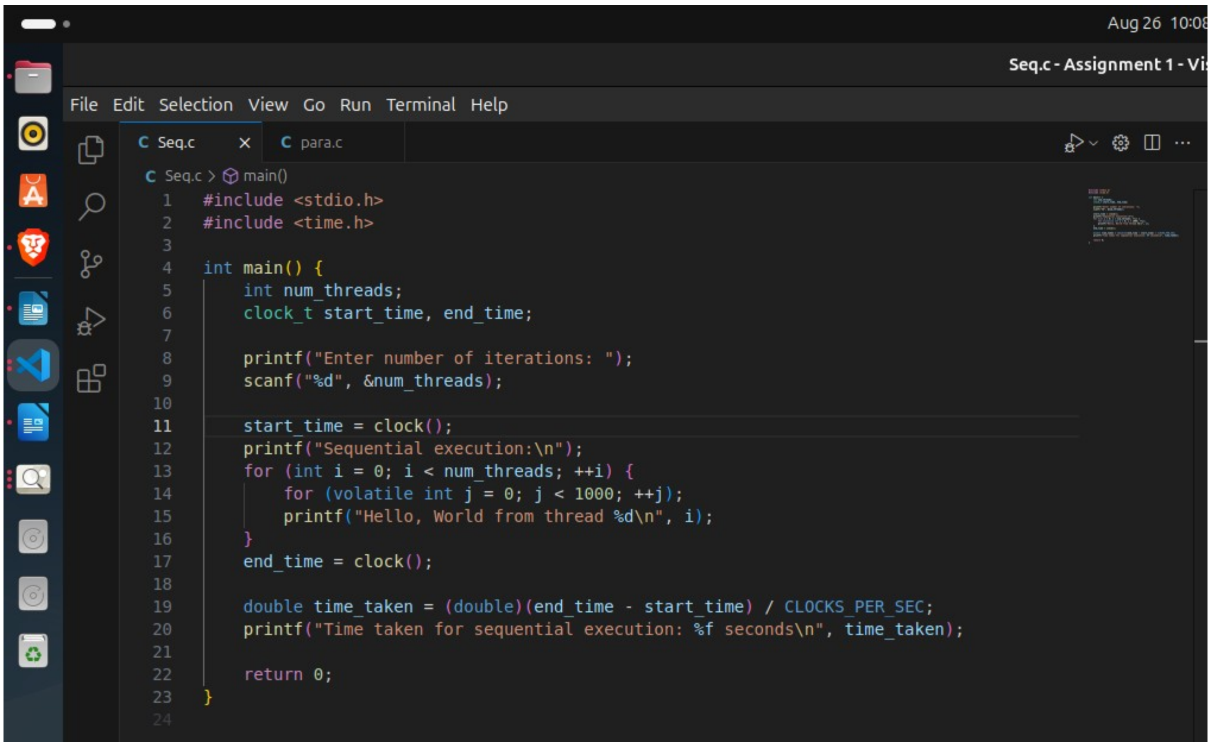
```
aftab@Aftab:~/Desktop/HPC/Assignment 1$ gcc -o para -fopenmp hello1.c
aftab@Aftab:~/Desktop/HPC/Assignment 1$ ./para
Enter number of threads: 12
Hello, world from thread 2.
Hello, world from thread 6.
Hello, world from thread 3.
Hello, world from thread 10.
Hello, world from thread 1.
Hello, world from thread 5.
Hello, world from thread 9.
Hello, world from thread 0.
Hello, world from thread 4.
Hello, world from thread 7.
Hello, world from thread 8.
Hello, world from thread 11.
aftab@Aftab:~/Desktop/HPC/Assignment 1$
```

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

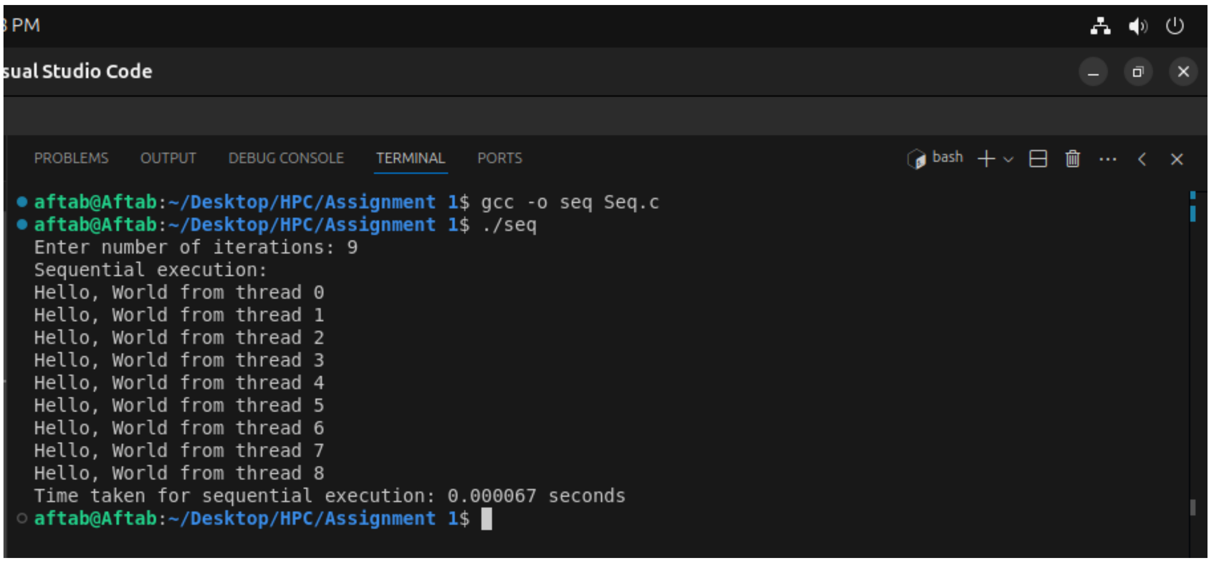
We first ask the user for number of threads – OpenMP allows to set the threads at runtime. Then, we print the Hello, World in sequential – number of times of threads count and then run the code in parallel in each thread.

Code:

Sequential Code :

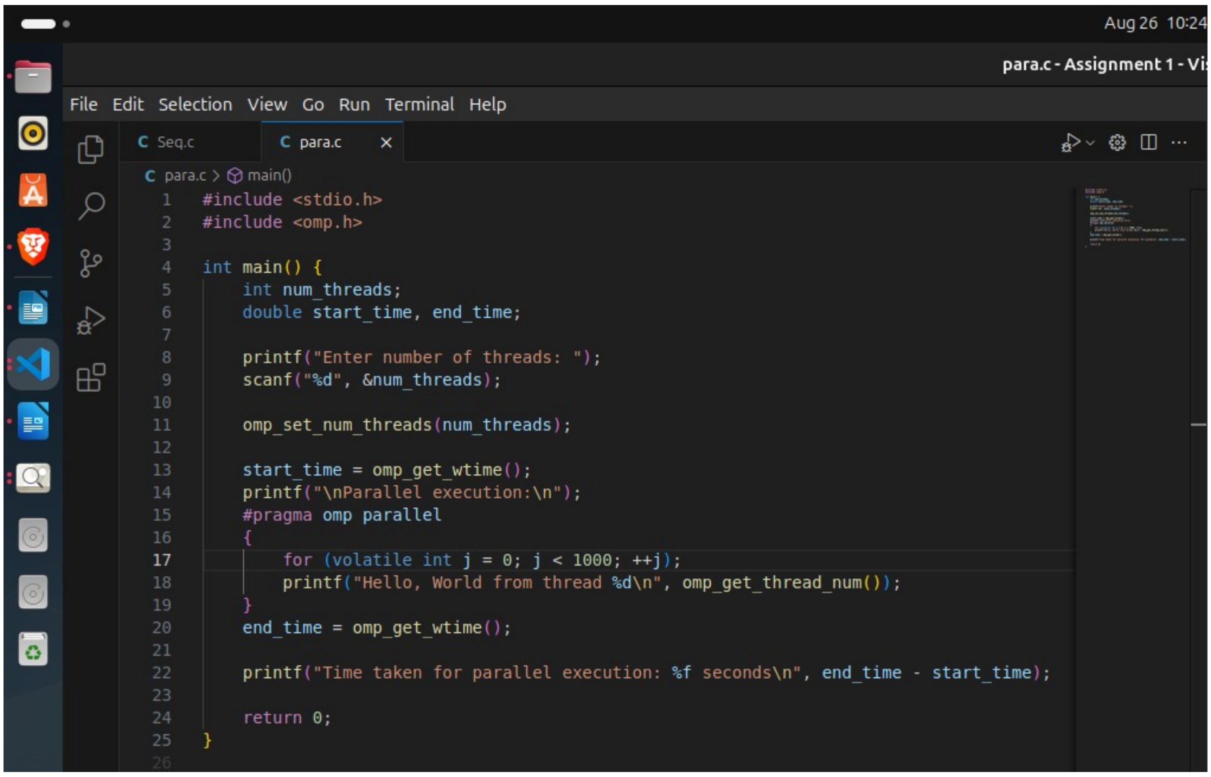


```c
#include <stdio.h>
#include <time.h>

int main() {
    int num_threads;
    clock_t start_time, end_time;

    printf("Enter number of iterations: ");
    scanf("%d", &num_threads);

    start_time = clock();
    printf("Sequential execution:\n");
    for (int i = 0; i < num_threads; ++i) {
        for (volatile int j = 0; j < 1000; ++j);
        printf("Hello, World from thread %d\n", i);
    }
    end_time = clock();

    double time_taken = (double)(end_time - start_time) / CLOCKS_PER_SEC;
    printf("Time taken for sequential execution: %f seconds\n", time_taken);

    return 0;
}
```

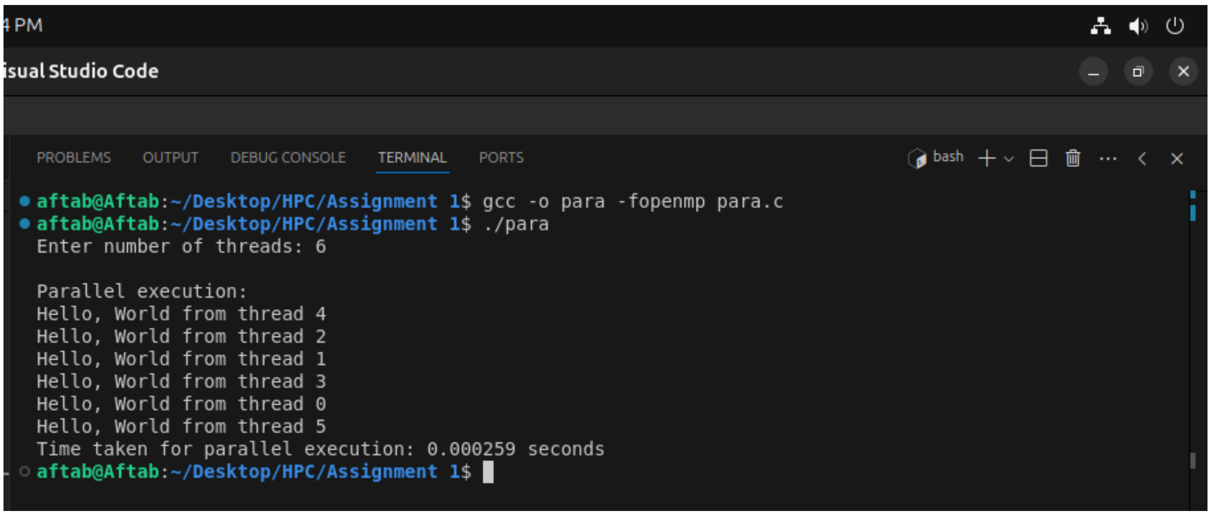


```
aftab@Aftab:~/Desktop/HPC/Assignment 1$ gcc -o seq Seq.c
aftab@Aftab:~/Desktop/HPC/Assignment 1$ ./seq
Enter number of iterations: 9
Sequential execution:
Hello, World from thread 0
Hello, World from thread 1
Hello, World from thread 2
Hello, World from thread 3
Hello, World from thread 4
Hello, World from thread 5
Hello, World from thread 6
Hello, World from thread 7
Hello, World from thread 8
Time taken for sequential execution: 0.000067 seconds
aftab@Aftab:~/Desktop/HPC/Assignment 1$
```

Parallel code :


Aug 26 10:24
para.c - Assignment 1 - Vi
File Edit Selection View Go Run Terminal Help
Seq.c
para.c
para.c > main()

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

int main() {
    int num_threads;
    double start_time, end_time;

    printf("Enter number of threads: ");
    scanf("%d", &num_threads);

    omp_set_num_threads(num_threads);

    start_time = omp_get_wtime();
    printf("\nParallel execution:\n");
    #pragma omp parallel
    {
        for (volatile int j = 0; j < 1000; ++j);
        printf("Hello, World from thread %d\n", omp_get_thread_num());
    }
    end_time = omp_get_wtime();

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

    return 0;
}
```



```
aftab@Aftab:~/Desktop/HPC/Assignment 1$ gcc -o para -fopenmp para.c
aftab@Aftab:~/Desktop/HPC/Assignment 1$ ./para
Enter number of threads: 6

Parallel execution:
Hello, World from thread 4
Hello, World from thread 2
Hello, World from thread 1
Hello, World from thread 3
Hello, World from thread 0
Hello, World from thread 5
Time taken for parallel execution: 0.000259 seconds
aftab@Aftab:~/Desktop/HPC/Assignment 1$ 
```

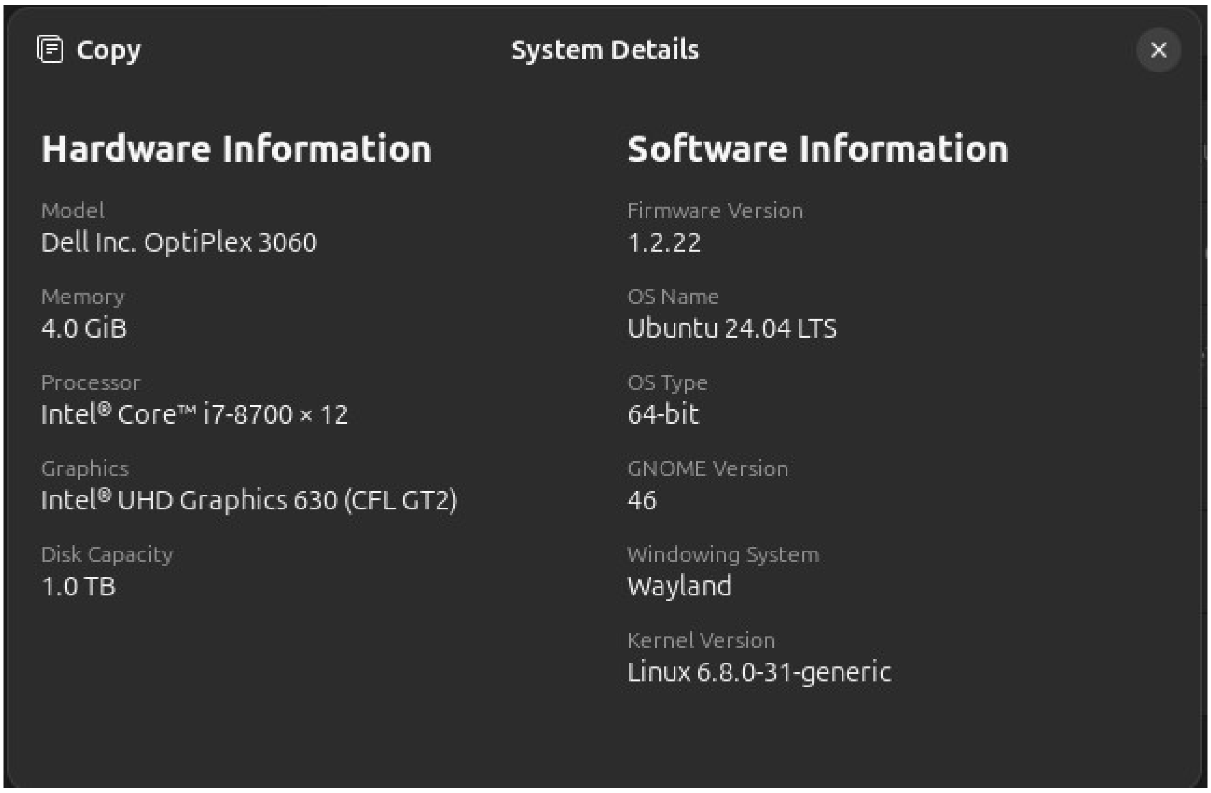
Analysis:

- In the parallel code, the user can dynamically set the number of threads, which run simultaneously.
- The machine's hardware determines the maximum thread count.
- Threads are executed in random order, with scheduling handled by the system.
- The order in which the threads are executed is not fixed; threads are scheduled for execution by the system in a random or unpredictable sequence.

GitHub Link:

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

System Information:



Copy System Details

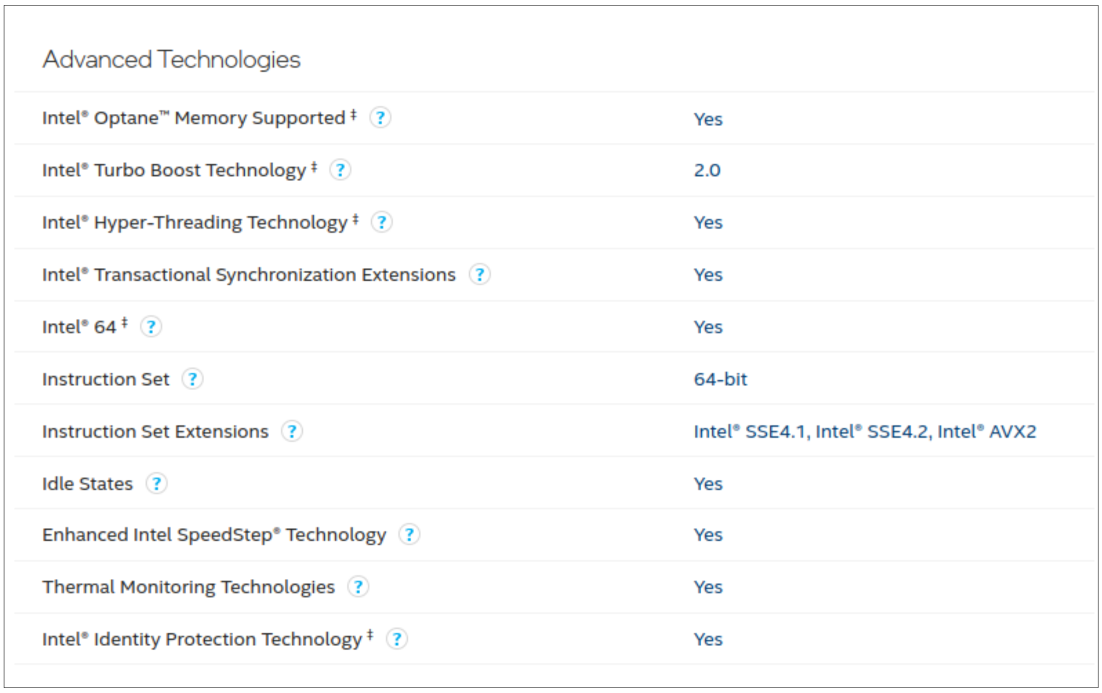
Hardware Information

Model	Dell Inc. OptiPlex 3060
Memory	4.0 GiB
Processor	Intel® Core™ i7-8700 × 12
Graphics	Intel® UHD Graphics 630 (CFL GT2)
Disk Capacity	1.0 TB

Software Information

Firmware Version	1.2.22
OS Name	Ubuntu 24.04 LTS
OS Type	64-bit
GNOME Version	46
Windowing System	Wayland
Kernel Version	Linux 6.8.0-31-generic

Instruction set Architecture:



Advanced Technologies

Intel® Optane™ Memory Supported ‡	Yes
Intel® Turbo Boost Technology ‡	2.0
Intel® Hyper-Threading Technology ‡	Yes
Intel® Transactional Synchronization Extensions	Yes
Intel® 64 ‡	Yes
Instruction Set	64-bit
Instruction Set Extensions	Intel® SSE4.1, Intel® SSE4.2, Intel® AVX2
Idle States	Yes
Enhanced Intel SpeedStep® Technology	Yes
Thermal Monitoring Technologies	Yes
Intel® Identity Protection Technology ‡	Yes

Calculating the theoretical FLOPS (Floating Point Operations Per Second) of a system involves understanding several key parameters related to the hardware architecture, such as the number of processor cores, clock speed, and the type of floating-point operations supported by the processor.

Parameters for Calculation:

1. **Processor Clock Speed (Ghz):**
    - This represents the speed at which the processor operates, measured in gigahertz (GHz).
2. **Number of Cores:**
    - Modern processors typically have multiple cores, each capable of executing instructions independently.
3. **SIMD (Single Instruction, Multiple Data) Units:**
    - SIMD units allow processors to perform operations on multiple data elements simultaneously, increasing throughput for certain types of calculations.
4. **FLOPS per Core:**
    - This depends on the architecture and capabilities of the processor, including factors like vector width and instruction set extensions (e.g., SSE, AVX).

Steps to Calculate Theoretical FLOPS:

To calculate the theoretical FLOPS per core, you typically consider the maximum number of floating-point operations that can be performed per cycle per core. This depends on the SIMD capabilities and instruction sets supported by your CPU.

1) Modern CPUs (e.g., Intel and AMD processors):

- Intel processors with AVX-512 support can perform up to 32 floating-point operations per cycle per core (FMA operations).
- **Intel processors with AVX-2 support can perform up to 16 floating-point operations per cycle per core (FMA operations).**
- AMD processors with AVX-2 support can perform up to 8 floating-point operations per cycle per core (FMA operations).

2) Calculate Theoretical FLOPS for the Entire System

- Once you have the FLOPS per core, you can calculate the theoretical FLOPS for the entire system by multiplying the FLOPS per core by the number of cores and the processor clock speed:
- FLOPS per Core of Cores Clock SpeedTheoretical FLOPS=FLOPS per Core×Number of Cores×Processor Clock Speed

Actual Calculation:

- **Processor**: Intel Core i7-8700 (6 cores, 12 threads)
- **Processor Clock Speed**: 3.2 GHz (base clock)
- **FLOPS per Core**: 16 FLOPS/cycle (AVX-2 support)

Calculation:

- **FLOPS per Core:** 16 FLOPS/cycle (AVX-2)
- **Number of Cores:** 6 cores
- **Processor Clock Speed:** 3.2 GHz (3.2 billion cycles per second)

FLOPS/cycle cores Ghz Theoretical FLOPS = **16 FLOPS/cycle × 6 cores × 3.2 Ghz**

= **307.2 GFLOPS (GigaFLOPS)**