**Class:** Final Year (Computer Science and Engineering)

**Year:** 2024-25 **Semester:** 1

**Course:** High Performance Computing Lab

**Practical No. 2**

**Exam Seat No: 21510036**

**Title of practical: Study and implementation of basic OpenMP clauses**

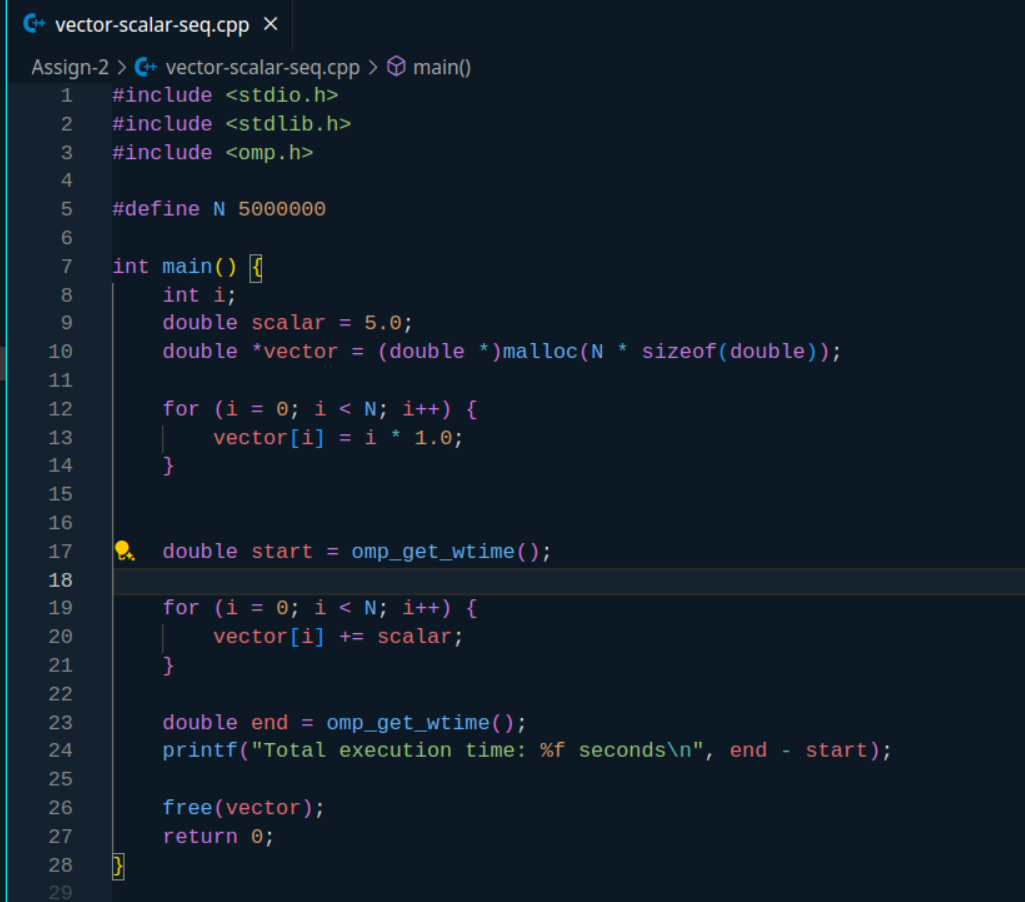
Implement following Programs using OpenMP with C:

1. Vector Scalar Addition
2. Calculation of value of Pi

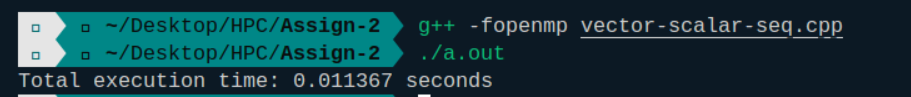
Analyse the performance of your programs for different number of threads and Data size.

**Problem Statement 1:**

**Screenshots: 1 - Sequential**



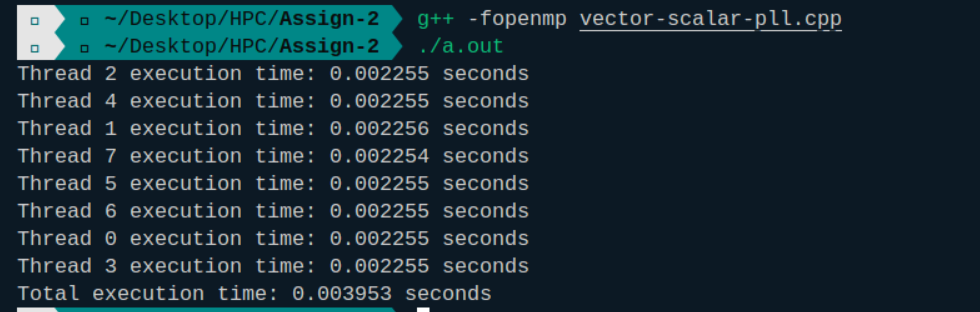
Output -



2 – Parallel



Output -



**Information:**

**In sequential method of vector scalar addition we took the size of vector as 5 million, and 1 thread i.e sequential**

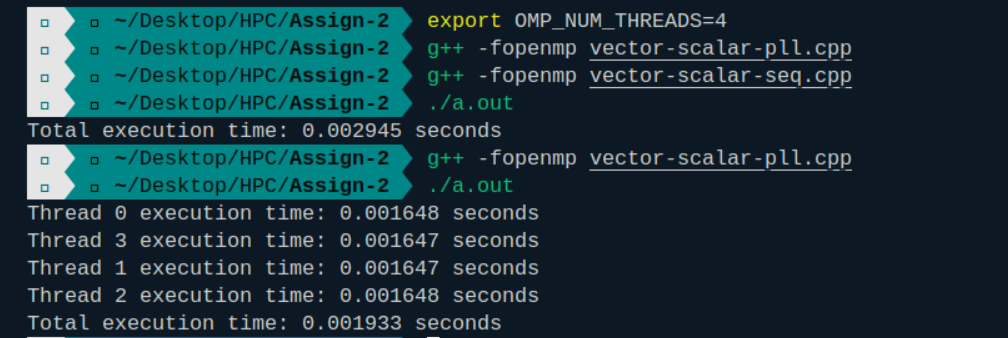
**In parallel method of vector scalar addition we took the size of vector as 5 million, and 8 threads i.e parallel**

**Analysis:**

**Analysis 1 – In sequential method with vector size of 1 million tends to be faster that parallel method with same vector size due to thread overheads like creation, deletion, context switching of OS, managing atomic operation mechnisms, scheduling, cache coherency, coordination. Used 8 threads here**

**Analysis 2 – When increased vector size to 5 million our parallel method tend to be lot faster than sequential one.**

**Analysis 3- data = 1 million but threads = 4**



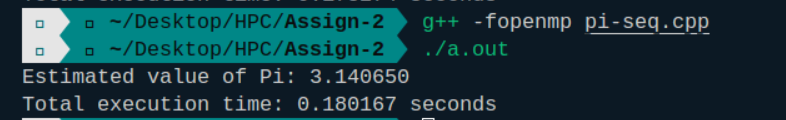
**Problem Statement 2:**

**Screenshots:**

**Sequential -**



Output -



Parallel -

#include <omp.h>

#include <stdio.h>

#include <stdlib.h>

#include <ctime>

#define NUM\_POINTS 6000000

int main() {

int i, count = 0;

double x, y, z, pi;

srand(time(NULL));

double start\_total = omp\_get\_wtime();

#pragma omp parallel

{

int local\_count = 0;

double start\_thread = omp\_get\_wtime();

#pragma omp for private(x, y, z)

for (i = 0; i < NUM\_POINTS; i++) {

x = (double)rand() / RAND\_MAX;

y = (double)rand() / RAND\_MAX;

z = x \* x + y \* y;

if (z <= 1) {

local\_count++;

}

}

#pragma omp atomic

count += local\_count;

double end\_thread = omp\_get\_wtime();

printf("Thread %d execution time: %f seconds\n", omp\_get\_thread\_num(), end\_thread - start\_thread);

}

double end\_total = omp\_get\_wtime();

pi = (double)count / NUM\_POINTS \* 4.0;

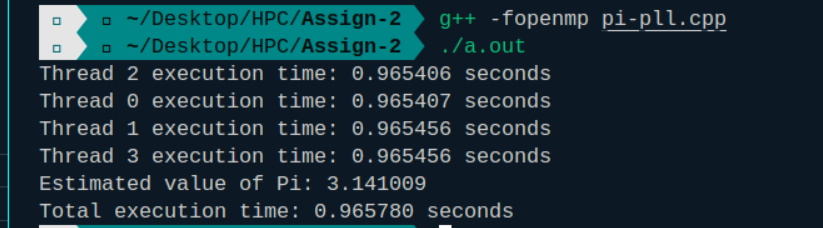
printf("Estimated value of Pi: %f\n", pi);

printf("Total execution time: %f seconds\n", end\_total - start\_total);

return 0;

}

Output -



**Information:**

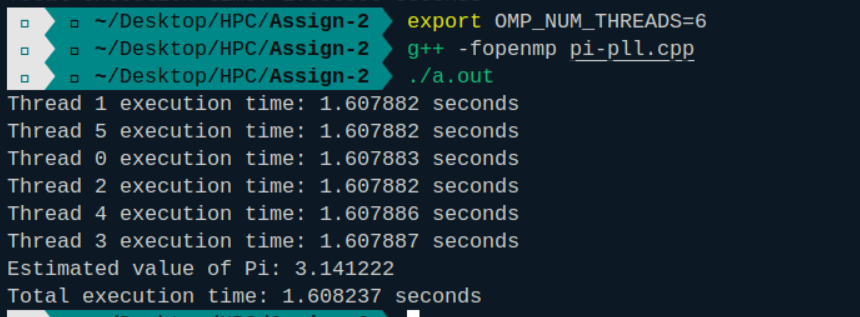
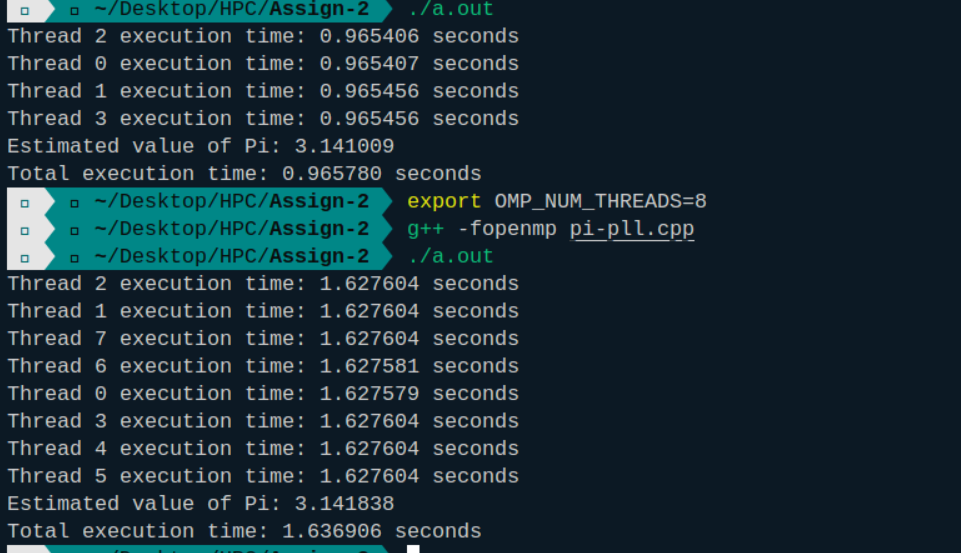
**Used monte carlo method to estimate calculation of value of pi**

**Analysis:**

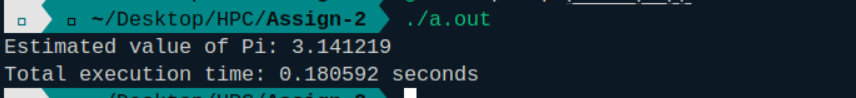
**Analysis 1 -**

**For 6 million points for parallel method we can see that total execution time is increasing with number of threads. While sequential tend to be faster but very inaccurate in terms of pi value due rand seed of a single thread doesnt provide complete randomness whereas with increasing threads the pi values tend to be closer to actual pi. It doesn’t necesarily always hold True.**

**Parallel output -**



**Sequential Output -**



**Github Link:**