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

**Year:** 2023-24 **Semester:** 1

**Course:** High Performance Computing Lab

**Practical No. 2**

**Exam Seat No: 2020BTECS00033**

**Name: Prathamesh Santosh Raje**

**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: Vector Scalar Addition**

**Screenshots:**

**Vector Scalar Addition Sequential Code:**

#include <omp.h>

#include <stdio.h>

#include <pthread.h>

int main()

{

    int N = 100;

    int A[N];

    for(int i=0;i<N;i++)A[i] = i + 1;

    int S = 2000;

    double itime, ftime, exec\_time;

    itime = omp\_get\_wtime();

    for (int i = 0; i < N; i++)

    {

        A[i] += S;

        printf("Thread: %d Index: %d\n", omp\_get\_thread\_num(),i);

    }

    ftime = omp\_get\_wtime();

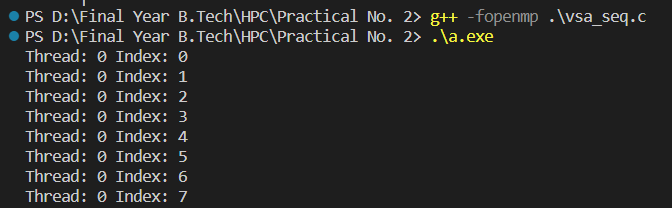
    exec\_time = ftime - itime;

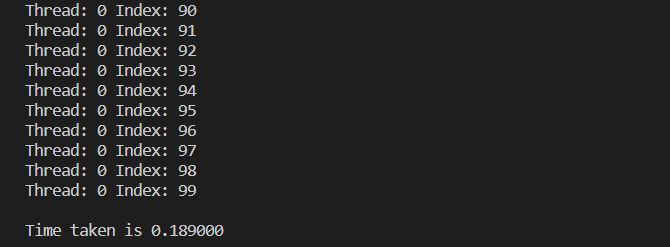
    printf("\nTime taken is %f\n", exec\_time);

    printf("\n");

}

**Vector Scalar Addition Sequential Output:**

****

****

**Vector Scalar Addition Parallel Code:**

#include <omp.h>

#include <stdio.h>

#include <pthread.h>

int main()

{

    int N = 100;

    int A[N];

    for(int i=0;i<N;i++)A[i] = i + 1;

    int S = 212354454;

    omp\_set\_num\_threads(6);

    double itime, ftime, exec\_time;

    itime = omp\_get\_wtime();

    #pragma omp parallel

    for (int i = 0; i < N; i++)

    {

        A[i] += S;

        printf("Thread: %d Index: %d\n", omp\_get\_thread\_num(),i);

    }

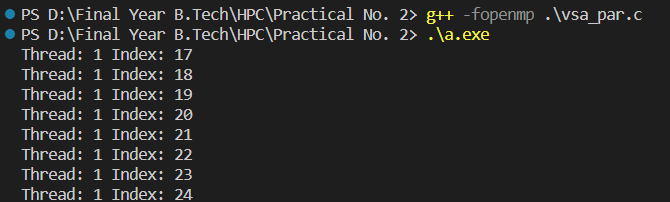
    ftime = omp\_get\_wtime();

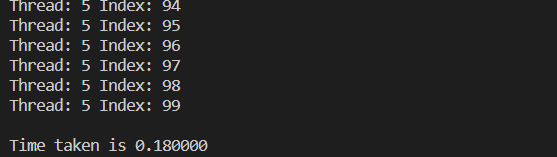
    exec\_time = ftime - itime;

    printf("\nTime taken is %f\n", exec\_time);

}

**Vector Scalar Addition Parallel Output:**

****

****

**Information:**

Execution time for sequential processing is: **0.189000**

Execution time for parallel processing is: **0.180000**

**Analysis:**

|  |  |  |
| --- | --- | --- |
| **No. of Threads** | **Execution Time** | |
| **Size = 100** | **Size = 1000** |
| **2** | **0.165000** | **1.105000** |
| **4** | **0.110000** | **1.100000** |
| **6** | **0.117000** | **1.088000** |
| **8** | **0.107000** | **1.133000** |

As the number of threads increasing, the performance is increased. Increasing the thread count beyond the number of CPU cores can potentially reduce execution time up to a point. Beyond that point, excessive threads may introduce overhead. Changing the thread count won't directly affect execution time since it's fixed at 6 threads. However, execution time can still vary depending on hardware and workload characteristics.

**Problem Statement 2: Calculation of value of Pi**

**Screenshots:**

**Calculation of value of Pi Sequential Code:**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define NUM\_POINTS 1000000

int main() {

    srand(time(NULL));

    clock\_t start\_time = clock();

    int inside\_circle = 0;

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

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

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

        double distance = x \* x + y \* y;

        if (distance <= 1.0) {

            inside\_circle++;

        }

    }

    double pi = 4.0 \* inside\_circle / NUM\_POINTS;

    printf("Estimated Pi value (sequential): %lf\n", pi);

    clock\_t end\_time = clock();

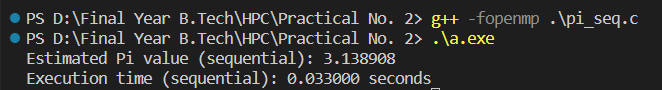
    double execution\_time = (double)(end\_time - start\_time) / CLOCKS\_PER\_SEC;

    printf("Execution time (sequential): %lf seconds\n", execution\_time);

    return 0;

}

**Calculation of value of Pi Sequential Output:**

****

**Calculation of value of Pi Parallel Code:**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <omp.h>

#define NUM\_POINTS 1000000

int main() {

    srand(time(NULL));

    clock\_t start\_time = clock();

    int inside\_circle = 0;

    #pragma omp parallel for reduction(+:inside\_circle)

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

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

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

        double distance = x \* x + y \* y;

        if (distance <= 1.0) {

            inside\_circle++;

        }

    }

    double pi = 4.0 \* inside\_circle / NUM\_POINTS;

    printf("Estimated Pi value (parallel): %lf\n", pi);

    clock\_t end\_time = clock();

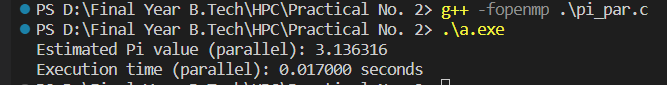
    double execution\_time = (double)(end\_time - start\_time) / CLOCKS\_PER\_SEC;

    printf("Execution time (parallel): %lf seconds\n", execution\_time);

    return 0;

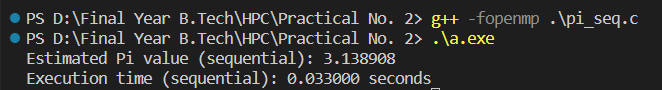
}

**Calculation of value of Pi Parallel Output:**

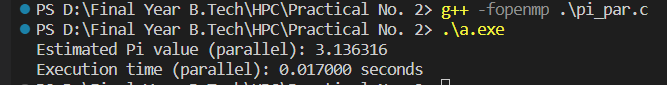
****

**Information:**

Execution time for sequential processing is:

****

Execution time for parallel processing is:

****

**Analysis:**

|  |  |  |
| --- | --- | --- |
| **No. of Threads** | **Execution Time** | |
| **Num\_points = 1000000** | **Num\_points = 100000000** |
| **2** | **0.022000** | **0.658000** |
| **4** | **0.014000** | **0.924000** |
| **6** | **0.021000** | **0.659000** |
| **8** | **0.013000** | **0.614000** |

As the number of threads increasing, the performance is increased. Increasing the thread count beyond the number of CPU cores can potentially reduce execution time up to a point. Beyond that point, excessive threads may introduce overhead. Changing the thread count won't directly affect execution time since it's fixed at 6 threads. However, execution time can still vary depending on hardware and workload characteristics.

**Github Link:**