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

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

**Course:** High Performance Computing Lab

**Practical No. 2**

**Exam Seat No: 2020BTECS00006**

**Name: Samrat V Jadhav**

**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>

void 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;

    // for(int i=0;i<N;i++){

    //     printf("%d ", A[i]);

    // }

    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>

void 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

    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;

    // for(int i=0;i<N;i++){

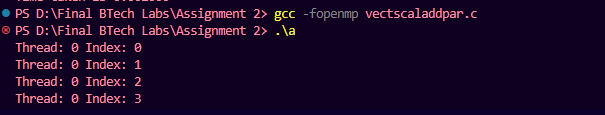
    //     printf("%d ", A[i]);

    // }

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

}

**Vector Scalar Addition Parallel Output:**





**Information:**

Execution time for sequential processing is:



Execution time for parallel processing is:

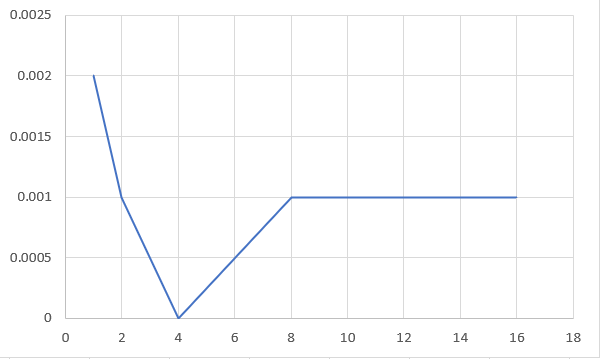


**Analysis:**

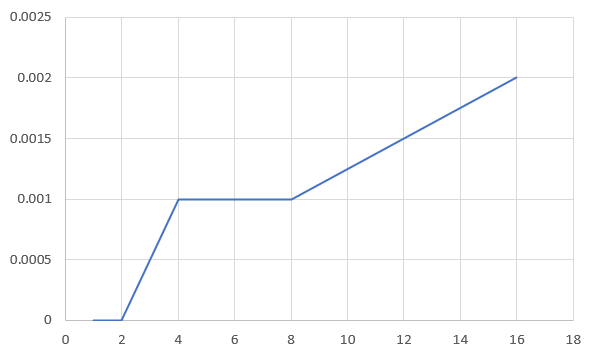
|  |  |  |
| --- | --- | --- |
| **No. of Threads** | **Input Size** | **Execution Time (seconds)** |
| 1 | 1000 | 0.00200 |
| 2 | 1000 | 0.00100 |
| 4 | 1000 | 0.00000 |
| 8 | 1000 | 0.00100 |
| 16 | 1000 | 0.00100 |
| 1 | 3000 | 0.000 |
| 2 | 3000 | 0.000 |
| 4 | 3000 | 0.00100 |
| 8 | 3000 | 0.001 |
| 16 | 3000 | 0.00200 |
| 1 | 9000 | 0.000 |
| 2 | 9000 | 0.0000 |
| 4 | 9000 | 0.001 |
| 8 | 9000 | 0.001 |
| **No. of Threads** | **Input Size** | **Execution Time (seconds)** |
| 16 | 9000 | 0.001 |
| 1 | 27000 | 0.001 |
| 2 | 27000 | 0.001 |
| 4 | 27000 | 0.001 |
| 8 | 27000 | 0.000 |
| 16 | 27000 | 0.002 |
| 1 | 81000 | 0.000 |
| 2 | 81000 | 0.001 |
| 4 | 81000 | 0.003 |
| 8 | 81000 | 0.002 |
| 16 | 81000 | 0.001 |

**Execution time vs number of threads graph:**

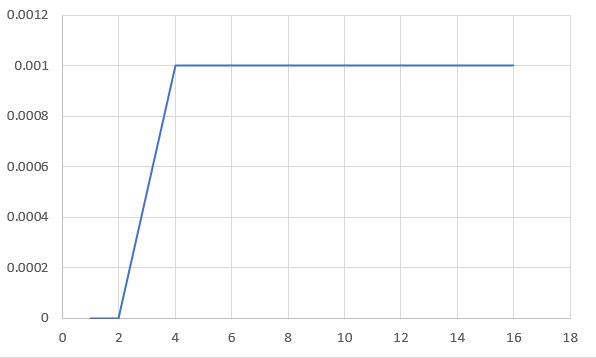
**For input size = 1000:**



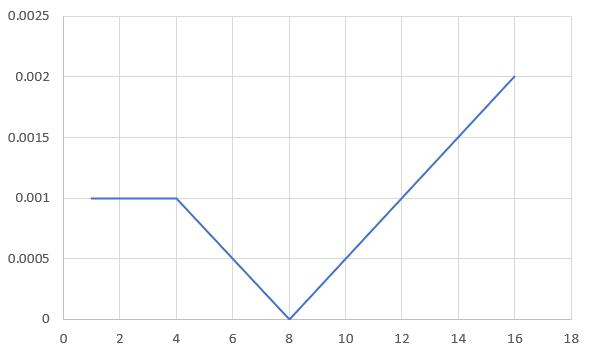
**For input size = 3000:**



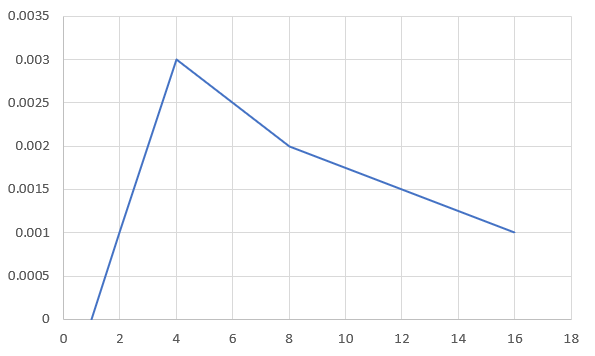
**For input size = 9000:**



**For input size = 27000:**



**For input size = 81000:**



For small input sizes (e.g., 1000, 3000), execution times are generally low and tend to decrease with an increasing number of threads.

For larger input sizes (e.g., 9000, 27000, 81000), execution times are generally low for fewer threads and can increase when using more threads, although there are variations.

The impact of the thread count on execution time varies based on the input size. In some cases, increasing the number of threads results in faster execution times, while in others, it remains efficient with fewer threads because of thread overheads.

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();

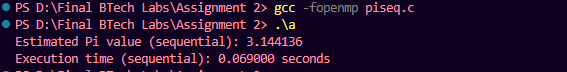
    double execution\_time = (double)(end\_time - start\_time) / CLOCKS\_PER\_SEC;

    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();

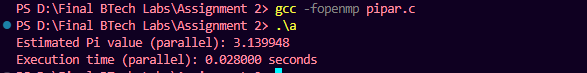
    double execution\_time = (double)(end\_time - start\_time) / CLOCKS\_PER\_SEC;

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

    return 0;

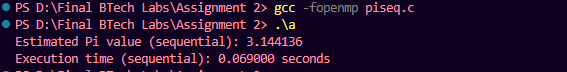
}

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

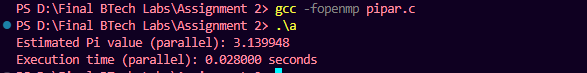


**Information:**

Execution time for sequential processing is:



Execution time for parallel processing is:



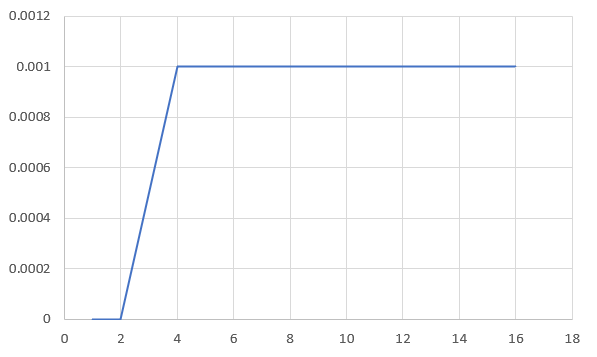
**Analysis:**

|  |  |  |  |
| --- | --- | --- | --- |
| **No. of Threads** | **Input Size** | **Estimated PI Value** | **Execution Time (seconds)** |
| **1** | **10000** | **3.137200** | **0.000** |
| **2** | **10000** | **3.147200** | **0.000** |
| **4** | **10000** | **3.152800** | **0.001** |
| **8** | **10000** | **3.153600** | **0.001** |
| **16** | **10000** | **3.161200** | **0.001** |
| **1** | **20000** | **3.122800** | **0.001** |
| **2** | **20000** | **3.129800** | **0.001** |

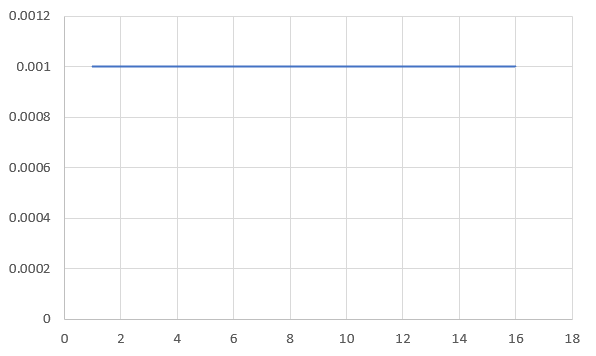
|  |  |  |  |
| --- | --- | --- | --- |
| **No. of Threads** | **Input Size** | **Estimated PI Value** | **Execution Time (seconds)** |
| **4** | **20000** | **3.156800** | **0.001** |
| **8** | **20000** | **3.160400** | **0.001** |
| **16** | **20000** | **3.160600** | **0.001** |
| **1** | **40000** | **3.131500** | **0.004** |
| **2** | **40000** | **3.140000** | **0.001** |
| **4** | **40000** | **3.151000** | **0.001** |
| **8** | **40000** | **3.151500** | **0.003** |
| **16** | **40000** | **3.153400** | **0.001** |
| **1** | **80000** | **3.140000** | **0.002** |
| **2** | **80000** | **3.150350** | **0.003** |
| **4** | **80000** | **3.144150** | **0.001** |
| **8** | **80000** | **3.156050** | **0.001** |
| **16** | **80000** | **3.151900** | **0.003** |
| **1** | **160000** | **3.128800** | **0.006** |
| **2** | **160000** | **3.139500** | **0.008** |
| **4** | **160000** | **3.140100** | **0.005** |
| **8** | **160000** | **3.145775** | **0.007** |
| **16** | **160000** | **3.147275** | **0.004** |

**Execution time vs number of threads graph:**

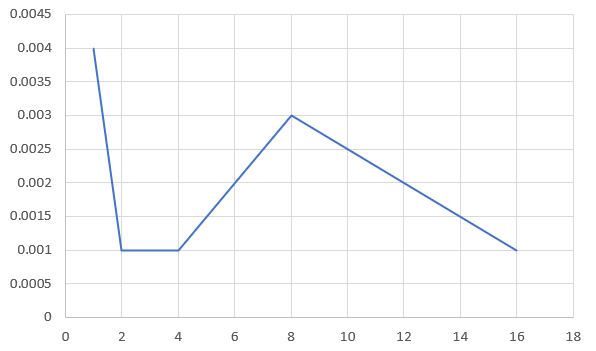
**For input size = 10000:**



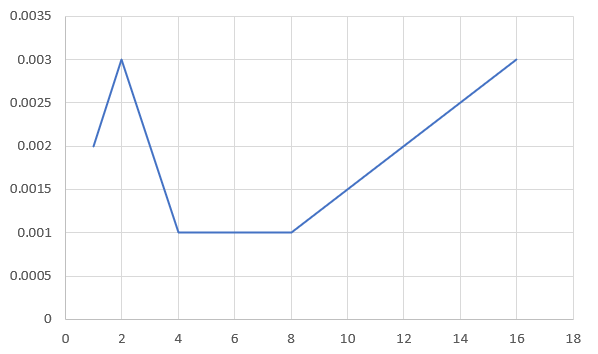
**For input size = 20000:**



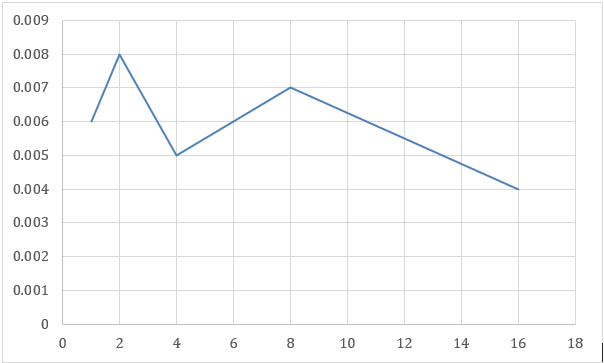
**For input size = 40000:**



**For input size = 80000:**



**For input size = 160000:**



The estimated value of PI varies slightly with different input sizes and numbers of threads, but it generally remains close to the actual value of PI (3.14159265359).

Execution times also vary, with smaller input sizes and fewer threads resulting in shorter execution times.

In general, increasing the number of threads does not always lead to faster execution times, especially for smaller input sizes, as there may be overhead associated with thread management.

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:**