Walchand College of Engineering, Sangli
Department of Computer Science and Engineering

Class: Final Year (Computer Science and Engineering)

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

Course: High Performance Computing Lab

Practical No. 2

Exam Seat No: 2020BTECS00041

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

Sequential:

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define SIZE 100000  // Define size of the vector
int main() {
    clock_t st=clock();
    int vector[SIZE];   // Create the vector and scalar
    int scalar =  50000;
    for (int i = 0; i < SIZE; i++)
    {
        vector[i] = 100000;
    }
    for (int i = 0; i < SIZE; i++)
    {
        vector[i] += scalar;
    }
    //printf("Resulting Vector: ");      // Display the resulting vector
    for (int i = 0; i < SIZE; i++) {
       // printf("%d ", vector[i]);
    }
    clock_t et = clock();
```

```
    double tt = (double)(et-st) / CLOCKS_PER_SEC;
    printf("Time required sequential program for vector scalar addition
size %d : %f \n",SIZE,tt);
    return 0;
}
```

Output:

```
PS D:\FinalYear\HPCL\Assignment 2\output> cd 'd:\FinalYear\HPCL\Assignment 2\output'
PS D:\FinalYear\HPCL\Assignment 2\output> & .\'q1sequential.exe'
Time required sequential program for vector scalar addition size 100000 : 0.000000
```

Parallel:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h> // Include OpenMP library
#define SIZE 100000  // Define size of the vector
#define NUM_THREADS 4   // Define number of threads
#define CHUNK 100
int main() {
    clock_t st=clock();
    omp_set_num_threads(NUM_THREADS);  // Set the number of threads
    int vector[SIZE];   // Create the vector and scalar
    int scalar = 50000;
    #pragma omp parallel for schedule(static,CHUNK)
    for (int i = 0; i < SIZE; i++)
    {
        vector[i] = 100000; // Generate vector in parallel
    }
    #pragma omp parallel for schedule(static,CHUNK)
    for (int i = 0; i < SIZE; i++)
    {
        vector[i] += scalar; //Perform vector-scalar addition in parallel
    }
    for (int i = 0; i < SIZE; i++) {
       // printf("%d ", vector[i]); }
    clock_t et = clock();
    double tt = (double)(et-st) / CLOCKS_PER_SEC;
    printf("\nTime required parallel program for Size %d and Chunk Size
%d: %f\n",SIZE,CHUNK,tt);
    return 0;
}
```

Screenshots: 1.Using static schedule

```
Num of threads: 4
Time required parallel program for Size 100000 and Chunk Size 100: 0.008000
```

```
Num of threads: 4
Time required parallel program for Size 100000 and Chunk Size 1000: 0.000000
```

```
Num of threads: 8
Time required parallel program for Size 100000 and Chunk Size 100: 0.016000
```

```
Num of threads: 8
Time required parallel program for Size 100000 and Chunk Size 1000: 0.015000
```

2. Using dynamic schedule

```
Num of threads: 4
Time required parallel program for Size 100000 and Chunk Size 100: 0.004000
```

```
Num of threads: 4
Time required parallel program for Size 100000 and Chunk Size 1000: 0.000000
```

```
Num of threads: 8
Time required parallel program for Size 100000 and Chunk Size 100: 0.000000
```

```
Num of threads: 8
Time required parallel program for Size 100000 and Chunk Size 1000: 0.004000
```

Analysis:

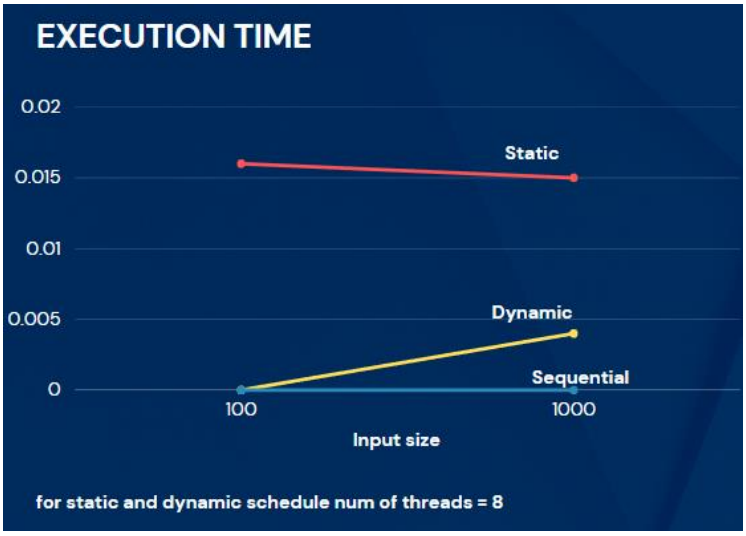
Static: Loop iterations are divided into pieces of size chunk and then statically assigned to threads. If chunk is not specified, iteration are evenly divided (if possible) divided contiguously among threads.

Dynamic: Loop iterations are divided into pieces of size chunk and then dynamically assignment to threads. When thread finishes one chunk, it is dynamically assigned to another. Default chunk size is 1.

Num of threads	Chunk Size	Static	Dynamic
4	100	0.008	0.004
4	1000	0.000	0.000
8	100	0.016	0.000
8	1000	0.015	0.004

As compared to static and dynamic schedule, time required to execute using statis schedule is more than time required to execute for dynamic schedule in parallel programming

Sequential time = 0.000000



Problem Statement 2: Calculation of value of Pi

Sequential Program:

```
// Calaculation of pi
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define size 100000
int main()
{
    clock_t st=clock();
    double step = 1.0 / size;
    double pi_sum = 0.0;
    double pi_estimate = 0.0;
    double local_sum = 0.0;
        for (int i = 0; i < size; i++)
        {
            double x = (i + 0.5) * step;
            local_sum += 4.0 / (1.0 + x * x);
        }
        #pragma omp critical
        pi_sum += local_sum;
    pi_estimate = step * pi_sum;
    printf("\nSize: %d", size);
    printf("\nEstimated Pi: %f\n", pi_estimate);
    clock_t et = clock();
    double tt = (double)(et-st) / CLOCKS_PER_SEC;
    printf("\nTime required sequential program : %f\n", tt);
    printf("\n");
    return 0;
}
```

Output:

```
Size: 100000
Estimated Pi: 3.141593

Time required sequential program : 0.000000
```

Parallel Program:

```
// Calaculation of pi
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <time.h>
#define NUM_THREADS 8   // Define number of threads
```

```c
#define CHUNK 1000
#define size 100000
int main()
{
    clock_t st=clock();
    double step = 1.0 / size;
    double pi_sum = 0.0;
    double pi_estimate = 0.0;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
        double local_sum = 0.0;
        #pragma omp for
        for (int i = 0; i < size; i++)
        {
            double x = (i + 0.5) * step;
            local_sum += 4.0 / (1.0 + x * x);
        }
        #pragma omp critical
        pi_sum += local_sum;
    }
    pi_estimate = step * pi_sum;
    printf("\nSize: %d", size);
    printf("\nThreads: %d", NUM_THREADS);
    printf("\nEstimated Pi: %f\n", pi_estimate);
    clock_t et = clock();
    double tt = (double)(et-st) / CLOCKS_PER_SEC;
    printf("\nTime required parallel program : %f\n", tt);
    printf("\n");
    return 0;
}
```

Screenshots:

```
Size: 100000
Threads: 4
Estimated Pi: 3.141593

Time required parallel program : 0.000000
```

```
Size: 100000
Threads: 10
Estimated Pi: 3.141593

Time required parallel program : 0.008000
```

```
Size: 100000
Threads: 8
Estimated Pi: 3.141593

Time required parallel program : 0.003000
```

```
Size: 100000
Threads: 14
Estimated Pi: 3.141593

Time required parallel program : 0.008000
```

Analysis:

Time executed for parallel program

Threads / Chunk Size	4	8	10	14
10	**0.008000**	**0.005000**	**0.008000**	**0.016000**
100	**0.011000**	**0.000000**	**0.013000**	**0.007000**
1000	**0.000000**	**0.003000**	**0.008000**	**0.008000**
10000	**0.000000**	**0.000000**	**0.005000**	**0.011000**

Critical construct: allows only single thread at a time to execute, so as we can see in the above chart as the number of threads are increasing time for execution also increases.

Github Link: https://github.com/Nikita226/HPCL/tree/main/A2