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. 6

Exam Seat No: 2020BTECS00041

Title of practical: Implementation of OpenMP programs.

Implement following Programs using OpenMP with C:

1. Implementation of Prefix sum.
2. Implementation of Matrix-Vector Multiplication.

Problem Statement 1: Implementation of Prefix sum

Sequential Code:

```
#include <stdio.h>
#include <time.h>
#define N 100000
int main()
{
    clock_t start=clock();
    int arr[N];
    int value = 1000;
    for (int i = 0; i < N; i++)
        arr[i] = value;
    int prefixSum[N];
    prefixSum[0] = arr[0];
    for (int i = 1; i < N; i++)
        prefixSum[i] = prefixSum[i - 1] + arr[i];
    clock_t end=clock();
    printf("Time taken: %f\n\n",(double)(end-start)/CLOCKS_PER_SEC);
    return 0;
}
```

Screenshots:

```
PS D:\FinalYear\HPCL\Assignment 6> gcc q1sequential.c
PS D:\FinalYear\HPCL\Assignment 6> .\a.exe
Time taken: 0.016000
```

Parallel:

#pragma omp parallel

```
PS D:\FinalYear\HPCL\Assignment 6> .\a.exe
Time taken: 0.015000
```

#pragma omp critical

```
PS D:\FinalYear\HPCL\Assignment 6> .\a.exe
Time taken: 0.016000
```

Defining number of threads:

```
#pragma omp parallel num_threads(threads)
{
    for (int i = 1; i < N; i++)
        prefixSum[i] = prefixSum[i - 1] + arr[i];
}
```

Output:

```
PS D:\FinalYear\HPCL\Assignment 6> gcc -fopenmp q1_2.c
PS D:\FinalYear\HPCL\Assignment 6> .\a.exe

Time taken: 0.001000    Number of Threads: 1

PS D:\FinalYear\HPCL\Assignment 6> gcc -fopenmp q1_2.c
PS D:\FinalYear\HPCL\Assignment 6> .\a.exe

Time taken: 0.001000    Number of Threads: 2

PS D:\FinalYear\HPCL\Assignment 6> gcc -fopenmp q1_2.c
PS D:\FinalYear\HPCL\Assignment 6> .\a.exe

Time taken: 0.003000    Number of Threads: 4

PS D:\FinalYear\HPCL\Assignment 6> gcc -fopenmp q1_2.c
PS D:\FinalYear\HPCL\Assignment 6> .\a.exe

Time taken: 0.005000    Number of Threads: 8

PS D:\FinalYear\HPCL\Assignment 6> gcc -fopenmp q1_2.c
PS D:\FinalYear\HPCL\Assignment 6> .\a.exe

Time taken: 0.007000    Number of Threads: 16
```

```
PS D:\FinalYear\HPCL\Assignment 6> gcc -fopenmp q1_2.c
PS D:\FinalYear\HPCL\Assignment 6> .\a.exe

Time taken: 0.008000    Number of Threads: 32

PS D:\FinalYear\HPCL\Assignment 6> gcc -fopenmp q1_2.c
PS D:\FinalYear\HPCL\Assignment 6> .\a.exe

Time taken: 0.028000    Number of Threads: 64

PS D:\FinalYear\HPCL\Assignment 6> gcc -fopenmp q1_2.c
PS D:\FinalYear\HPCL\Assignment 6> .\a.exe

Time taken: 0.045000    Number of Threads: 128

PS D:\FinalYear\HPCL\Assignment 6> gcc -fopenmp q1_2.c
PS D:\FinalYear\HPCL\Assignment 6> .\a.exe

Time taken: 0.067000    Number of Threads: 256

PS D:\FinalYear\HPCL\Assignment 6> gcc -fopenmp q1_2.c
PS D:\FinalYear\HPCL\Assignment 6> .\a.exe

Time taken: 0.121000    Number of Threads: 512
```

Information:

Parallelism of a prefix sum algorithm involves dividing the computation of the prefix sum into multiple tasks that can be executed concurrently. The prefix sum operation can be parallelized by dividing the input sequence into smaller segments or chunks. Each thread is responsible for computing the prefix sum of its segment independently. Parallelism in a prefix sum algorithm aims to exploit multiple processing units to compute the prefix sum faster and more efficiently than a sequential approach.

#pragma omp critical directive ensures that only one thread at a time can execute the critical section of code, preventing data races and maintaining correctness

By varying the number of threads, we can control the degree of parallelism, optimizing the algorithm's performance for the available hardware and problem size. The choice of the number of threads and the algorithm variant can impact the overall performance on different hardware configurations.

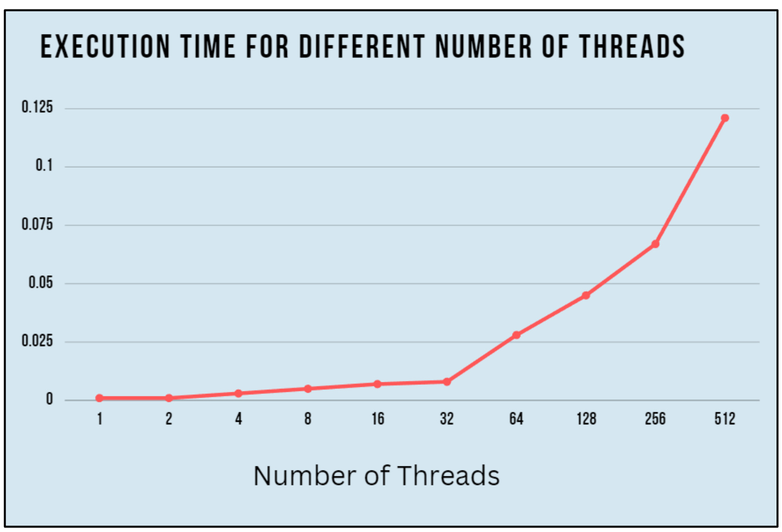
Analysis:

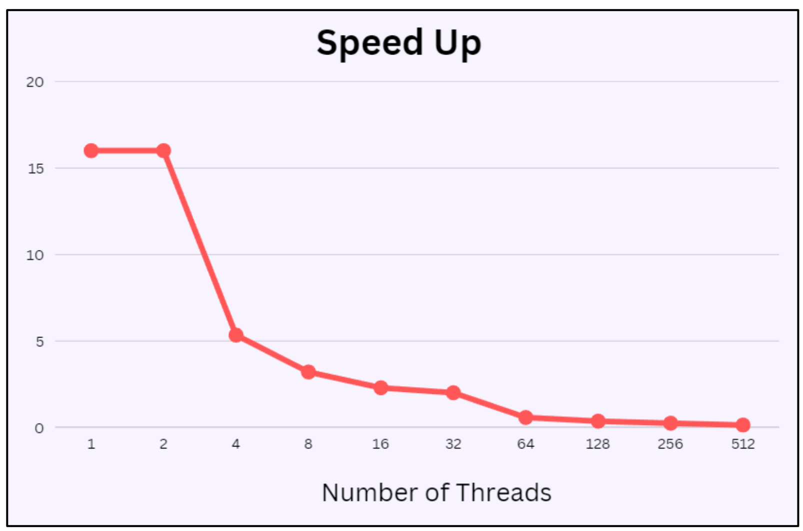
Execution Time taken for Sequential Algorithm = 0.016 sec

Execution Time taken for Parallel Algorithm

1. using #pragma omp parallel = 0.015 sec Speed Up = 1.06
2. using #pragma omp critical = 0.016 sec Speed Up = 1
3. defining different number of threads

Num of threads	*Execution Time*	*SpeedUp*
1	0.001 sec	16
2	0.001 sec	16
4	0.003 sec	5.333
8	0.005 sec	3.2
16	0.007 sec	2.286
32	0.008 sec	2
64	0.028 sec	0.571
128	0.045 sec	0.356
256	0.067 sec	0.239
512	0.121 sec	0.132





Problem Statement 2: Implementation of Matrix-Vector Multiplication.
Sequential Program:

```
#include <stdio.h>
#include <time.h>
#define n 700
int main()
{
    clock_t start = clock();
    float vector[n], result[n], matrix[n][n];
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            matrix[i][j]=100000.99999;

    for (int i = 0; i < n; i++)
        vector[i]=100000.99999;

    for (int i = 0; i < n; i++)
    {
        result[i] = 0;
        for (int j = 0; j < n; j++)
            result[i] += matrix[i][j] * vector[j];
    }
    clock_t end=clock();
    double t=(double)(end-start)/CLOCKS_PER_SEC;
    printf("\nSize of matrix: %d * %d",n, n);
    printf("\nSize of Vector: %d", n);
    printf("\nTime taken: %f\n\n",(double)(end-start)/CLOCKS_PER_SEC);
    return 0;
}
```

Output of Sequential Program:

```
Size of matrix: 400 * 400
Size of Vector: 400
Time taken: 0.008000
```

```
Size of matrix: 500 * 500
Size of Vector: 500
Time taken: 0.015000
```

```
Size of matrix: 600 * 600
Size of Vector: 600
Time taken: 0.017000
```

```
Size of matrix: 700 * 700
Size of Vector: 700
Time taken: 0.016000
```

Output of Parallel Program:

#pragma omp parallel

```
Size of matrix: 700 * 700
Size of Vector: 700
Time taken: 0.013000
```

#pragma omp for

```
Size of matrix: 700 * 700
Size of Vector: 700
Time taken: 0.008000
```

#pragma omp collapse(2)

```
Size of matrix: 700 * 700
Size of Vector: 700
Time taken: 0.016000
```

#pragma omp parallel reduction(+:x)

```
Size of matrix: 700 * 700
Size of Vector: 700
Time taken: 0.009000
```

#pragma omp critical

```
Size of matrix: 700 * 700
Size of Vector: 700
Time taken: 0.015000
```

#pragma omp nowait

```
Size of matrix: 400 * 400
Size of Vector: 400
Time taken: 0.003000
```

```
Size of matrix: 500 * 500
Size of Vector: 500
Time taken: 0.007000
```

```
Size of matrix: 600 * 600
Size of Vector: 600
Time taken: 0.008000
```

```
Size of matrix: 700 * 700
Size of Vector: 700
Time taken: 0.009000
```

Analysis:

Execution Time taken for Parallel Algorithm for size of matrix 700 * 700

1. using #pragma omp parallel = 0.013 sec
2. using #pragma omp for = 0.008 sec
3. using #pragma omp collapse(2) = 0.016 sec
4. using #pragma omp parallel reduction(+:x) = 0.009 sec
5. using #pragma omp critical = 0.015 sec

Defining different number of sizes of matrix

Size of Matrix	*Sequential Execution Time (Ts)*	*Parallel Execution Time (Tp) without using nowait*	*Parallel Execution Time (Tp) using nowait*
400 * 400	0.008 sec	0.004 sec	0.003 sec
500 * 500	0.015 sec	0.009 sec	0.007 sec
600 * 600	0.017 sec	0.013 sec	0.008 sec
700 * 700	0.016 sec	0.014 sec	0.009 sec

Size of Matrix	*Speed Up without using nowait*	*Speed Up using nowait*
400 * 400	2	2.667
500 * 500	1.667	2.142
600 * 600	1.308	2.125
700 * 700	1.142	1.778