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. 3

PRN: 2020BTECS00057

Title of practical:

Study and Implementation of schedule, nowait, reduction, ordered and collapse clauses

Problem Statement 1:
Analyse and implement a Parallel code for below program using OpenMP.
// C Program to find the minimum scalar product of two vectors (dot product)

Code:

```c
#include <stdio.h>
#include <time.h>
#include <omp.h>

int main(){
   int size;
   printf("Enter size of array = ");
   scanf("%d",&size);
   int arr1[size];
   int arr2[size];
   for(int i=0;i<size;i++)
   {
        arr1[i]=i;
        arr2[i]=i;
   }
   int n = sizeof(arr1)/sizeof(arr1[0]);
   clock_t st = clock();
   //asc
   for(int i=0; i<n; i++){
       for(int j=i+1; j<n; j++){ if(arr1[i]>arr1[j]){
               int temp = arr1[i];
               arr1[i] = arr1[j];
               arr1[j] = temp;
           }
```

```
    }
    //des
    for(int i=0; i<n; i++){
        for(int j=i+1; j<n;
        j++){
            if(arr2[i]<arr2[j]){
                int temp =
                arr2[i]; arr2[i]
                = arr2[j];
                arr2[j] = temp;
            }
        }
    }
    double product = 0;
    omp_set_num_threads(
    8);
    #pragma omp parallel for
    schedule(static,2) for(int i=0; i<n;
    i++)
    {
        product +=
        (double)arr1[i]*arr2[i]; int
        thread=omp_get_thread_num();
        printf("\n%d. Thread = %d, Product = %f",i,thread,product);
    }
    clock_t et = clock();
    double elapsed_time = (double)(et - st) / CLOCKS_PER_SEC;
    double elapsed_miliseconds = elapsed_time * 1000;
    printf("\n%f",product);
    printf("\nTime taken: %f miliseconds",
    elapsed_miliseconds); printf("\nTime taken: %f
    seconds\n", elapsed_time);
    return 0;
}
```

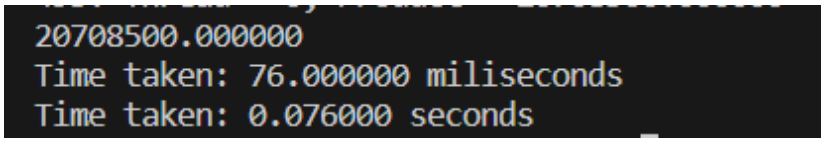
Screenshots:

Keeping number of threads constant and varying size of Data.

Threads = 8, Array size = 10

```
120.000000
Time taken: 4.000000 miliseconds
Time taken: 0.004000 seconds
```

Threads = 8, Array size = 500



Threads = 8, Array size = 1000

```
166167000.000000
Time taken: 151.000000 miliseconds
Time taken: 0.151000 seconds
```

Keeping data constant and increasing number of threads.

Threads = 10, Array size = 500

```
20708500.000000
Time taken: 71.000000 miliseconds
Time taken: 0.071000 seconds
```

Threads = 250, Array size = 500

```
20585908.000000
Time taken: 93.000000 miliseconds
Time taken: 0.093000 seconds
```

Threads = 500, Array size = 500

```
20663618.000000
Time taken: 110.000000 miliseconds
Time taken: 0.110000 seconds
```

Information and analysis:

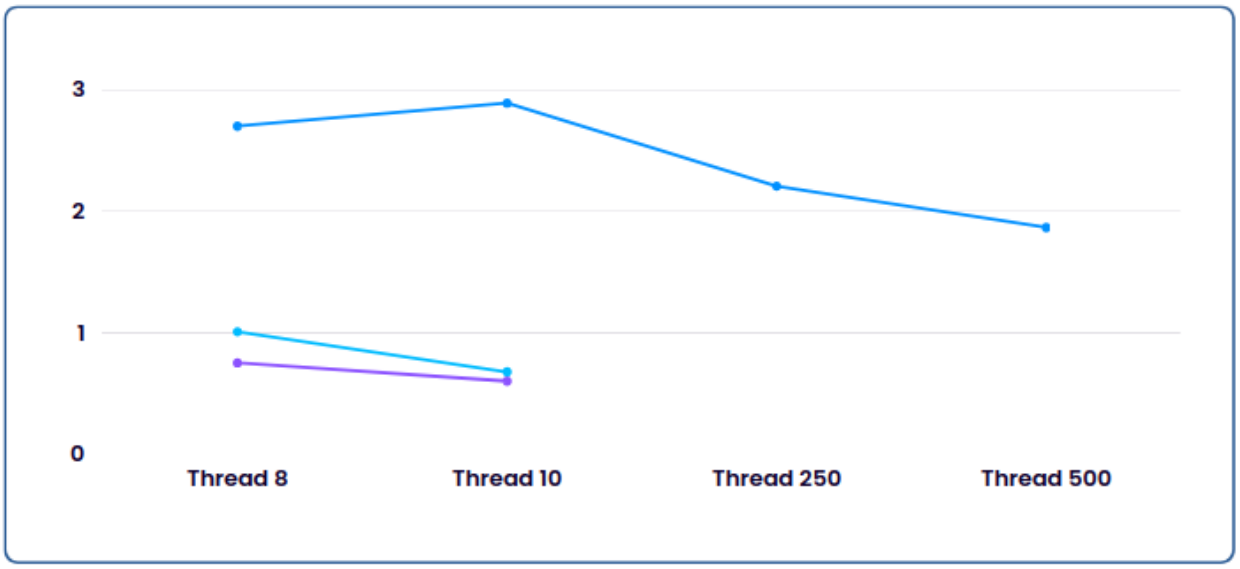
1. **schedule clause:** The schedule clause in OpenMP is used to specify how loop iterations are divided and scheduled among threads in a parallel loop construct.
    a. **Static Schedule (schedule(static, chunk)):** Divides iterations into contiguous chunks, distributing them statically among threads. Useful when loop iterations have roughly uniform workload
    b. **Dynamic Schedule (schedule(dynamic, chunk)):** Divides iterations into smaller, dynamic chunks, allowing threads to pick new chunks when they finish their current work. Useful when loop iterations have varying workloads.

Analysis:

Number of Threads	Data Size	Sequential Time(sec)(Ts)	Parallel Time(sec)(Tp)	Speedup(Ts/Tp)
8	10	0.003000	0.004000	0.75
8	500	0.205000	0.076000	2.697368421
8	1000	0.152000	0.151000	1.006622517
10	10	0.003000	0.005000	0.6

10	500	0.205000	0.071000	2.887323944
10	1000	0.152000	0.225000	0.675555556

250	500	0.205000	0.093000	2.204301075
500	500	0.205000	0.110000	1.863636364



Problem Statement 2:

Write OpenMP code for two 2D Matrix addition, vary the size of your matrices from 250, 500, 750, 1000, and 2000 and measure the runtime with one thread (Use functions in C in calculate the execution time or use GPROF)

i. For each matrix size, change the number of threads from 2,4,8., and plot the speedup versus the number of threads.

ii. Explain whether or not the scaling behaviour is as expected.

Code:

```c
#include<stdio.h>
#include <omp.h>

int main(){
    int dimention;

    printf("Enter dimention for 2D matrix = ");
    scanf("%d",&dimention);

    //Parallel code
    int mp1[dimention][dimention],mp2[dimention][dimention];
    double start_time_parallel = omp_get_wtime();
    #pragma omp parallel for schedule(dynamic) num_threads(1) collapse(2)
    for(int i=0;i<dimention;i++){
        for(int j=0;j<dimention;j++){
            mp1[i][j]=i+j;
            mp2[i][i]=i-j;
        }
    }

    int ans1[dimention][dimention];
    #pragma omp parallel for schedule(dynamic) num_threads(1) collapse(2)
    for(int i=0;i<dimention;i++){
        for(int j=0;j<dimention;j++){
            ans1[i][j]=mp1[i][j]+mp2[i][j];
        }
    }
    double end_time_parallel = omp_get_wtime();

    printf("\nParallel Method Time: %f seconds\n", (end_time_parallel -
start_time_parallel));
    return 0;
}
```

Screenshots:

Threads = 2

Matrix size = 250

```
PS E:\7 Sem\HPC LAB> .\a.exe
Enter dimention for 2D matrix = 250

Parallel Method Time: 0.000000 seconds
```

Matrix size = 300

```
PS E:\7 Sem\HPC LAB> .\a.exe
Enter dimention for 2D matrix = 300

Parallel Method Time: 0.009000 seconds
```

Matrix size = 350

```
PS E:\7 Sem\HPC LAB> .\a.exe
Enter dimention for 2D matrix = 350

Parallel Method Time: 0.015000 seconds
```

Matrix size = 415

```
PS E:\7 Sem\HPC LAB> .\a.exe
Enter dimention for 2D matrix = 415

Parallel Method Time: 0.018000 seconds
```

Threads = 4

Matrix size = 250

```
PS E:\7 Sem\HPC LAB> .\a.exe
Enter dimention for 2D matrix = 250

Parallel Method Time: 0.000000 seconds
```

Matrix size = 300

```
PS E:\7 Sem\HPC LAB> .\a.exe
Enter dimention for 2D matrix = 300

Parallel Method Time: 0.005000 seconds
```

Matrix size = 350

```
PS E:\7 Sem\HPC LAB> .\a.exe
Enter dimention for 2D matrix = 350

Parallel Method Time: 0.016000 seconds
```

Matrix size = 415

```
PS E:\7 Sem\HPC LAB> .\a.exe
Enter dimention for 2D matrix = 415

Parallel Method Time: 0.017000 seconds
```

Threads = 8

Matrix size = 250

```
PS E:\7 Sem\HPC LAB> .\a.exe
Enter dimention for 2D matrix = 250

Parallel Method Time: 0.006000 seconds
```

Matrix size = 300

```
PS E:\7 Sem\HPC LAB> .\a.exe
Enter dimention for 2D matrix = 300

Parallel Method Time: 0.015000 seconds
```

Matrix size = 350

```
PS E:\7 Sem\HPC LAB> .\a.exe
Enter dimention for 2D matrix = 350

Parallel Method Time: 0.017000 seconds
```

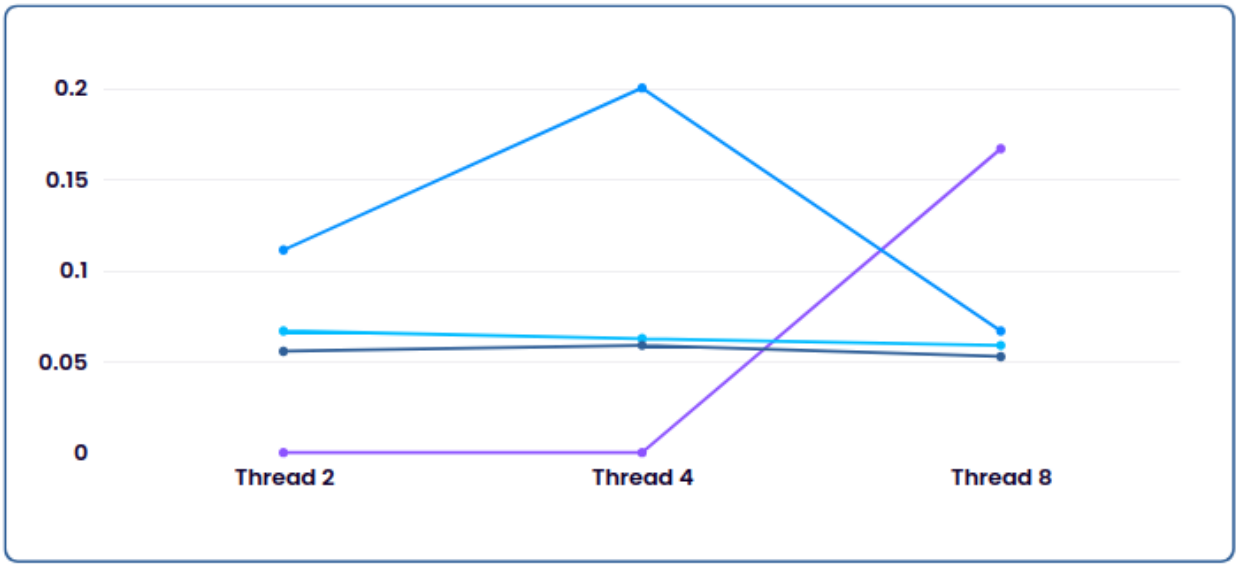
Matrix size = 415

```
PS E:\7 Sem\HPC LAB> .\a.exe
Enter dimention for 2D matrix = 415

Parallel Method Time: 0.019000 seconds
```

Information and analysis:

Number of Threads	Data Size	Sequential Time(sec)(Ts)	Parallel Time(sec)(Tp)	Speedup(Ts/Tp)
2	250	0.001000	0.000000	0
2	300	0.001000	0.009000	0.111111111
2	350	0.001000	0.015000	0.066666667
2	415	0.001000	0.018000	0.055555556
4	250	0.001000	0.000000	0
4	300	0.001000	0.005000	0.2
4	350	0.001000	0.016000	0.0625
4	415	0.001000	0.017000	0.058823529
8	250	0.001000	0.006000	0.166666667
8	300	0.001000	0.015000	0.066666667
8	350	0.001000	0.017000	0.058823529
8	415	0.001000	0.019000	0.052631579


0.2
0.15
0.1
0.05
0
Thread 2
Thread 4
Thread 8
250
300
350
415

It is observed that large number of data size requires more execution time independent from number of threads used to execute. There is slight increase in execution time while number of threads are increased, due to the mapping of logical thread to physical thread, but here increase in time is negligible.

Problem Statement 3:

For 1D Vector (size=200) and scalar addition, Write a OpenMP code with the following: i. Use STATIC schedule and set the loop iteration chunk size to various sizes when changing the size of your matrix. Analyze the speedup. ii. Use DYNAMIC schedule and set the loop iteration chunk size to various sizes when changing the size of your matrix. Analyze the speedup. iii. Demonstrate the use of nowait clause.

Code:

```c
#include <stdio.h>
#include <omp.h>

int main() {
    int n=0;
    printf("Enter Vector size: ");
    scanf("%d",&n);
    float vector[n];
    double scalar;
    printf("Enter scaler value: ");
```

```
    scanf("%f",&scalar);

    //Serial Code
    double start_time_serial =
    omp_get_wtime(); for (int i = 0; i < n;
    i++) {
        vector[i] = i + 100.987453323212;
    }

    for (int i = 0; i < n;
        i++) { vector[i] +=
        scalar;
    }
    double end_time_serial = omp_get_wtime();

    printf("Serial Method Time: %f seconds\n",
(end_time_serial - start_time_serial));

    //Parallel Code
    double start_time_parallel = omp_get_wtime();
    #pragma omp parallel for schedule(static,4) num_threads(2)
    private(scalar) for (int i = 0; i < n; i++) {
        vector[i] = i + 100.987453323212;
    }

    #pragma omp parallel for schedule(dynamic,4) num_threads(2)
    private(scalar) for (int i = 0; i < n; i++) {
        vector[i] += scalar;
    }
    double end_time_parallel = omp_get_wtime();

    printf("Parallel Method Time: %f seconds\n",
(end_time_parallel - start_time_parallel));

    return 0;
}
```

Screenshots:

Threads = 2 Vector Size= 100000

```
PS E:\7 Sem\HPC LAB> .\a.exe
Enter Vector size: 100000
Enter scaler value: 90900909090909.9090404640604646
Serial Method Time: 0.000000 seconds
Parallel Method Time: 0.015000 seconds
```

Threads = 4 Vector Size= 100000

```
PS E:\7 Sem\HPC LAB> .\a.exe
Enter Vector size: 10000
Enter scaler value: 90900909090909.9090404640604646
Serial Method Time: 0.000000 seconds
Parallel Method Time: 0.000000 seconds
PS E:\7 Sem\HPC LAB>
```

Threads = 8 Vector Size= 100000

```
PS E:\7 Sem\HPC LAB> .\a.exe
Enter Vector size: 10000
Enter scaler value: 90900909090909.9090404640604646
Serial Method Time: 0.000000 seconds
Parallel Method Time: 0.000000 seconds
```

As there is no sufficient data to perform parallelism, changing clause to static or dynamic, or varying the size of threads will not affect execution time.

Information and analysis:

2. **nowait clause:** Threads can continue execution immediately after completing their portion of work inside the parallel region, without waiting for others. They still synchronize at the end of the parallel region
3. **schedule clause:** The schedule clause in OpenMP is used to specify how loop iterations are divided and scheduled among threads in a parallel loop construct.
    a. **Static Schedule (schedule(static, chunk)):** Divides iterations into contiguous chunks, distributing them statically among threads. Useful when loop iterations have roughly uniform workload
    b. **Dynamic Schedule (schedule(dynamic, chunk)):** Divides iterations into smaller, dynamic chunks, allowing threads to pick new chunks when they finish their current work. Useful when loop iterations have varying workloads.