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

Class: Final Year B.Tech(Computer Science and Engineering)

Year: 2025-26 **Semester:** 1

Course: High Performance Computing Lab

PRN: 22510064 (Parshwa Herwade)

Github Link : Sem-7-Assign/HPC lab at main · parshwa913/Sem-7-Assign · GitHub

Practical No. 3

Exam Seat No:

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)

Screenshots:

dot_product.c > main()

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

int main() {
    int n;
    printf("Enter size of vectors: ");
    scanf("%d", &n);

    int *a = malloc(n * sizeof(int));
    int *b = malloc(n * sizeof(int));
    printf("Enter elements of first vector:\n");
    for (int i = 0; i < n; i++) scanf("%d", &a[i]);
    printf("Enter elements of second vector:\n");
    for (int i = 0; i < n; i++) scanf("%d", &b[i]);

    int dot = 0;
    #pragma omp parallel for reduction(+:dot) schedule(static)
    for (int i = 0; i < n; i++) {
        dot += a[i] * b[i];
    }

    printf("Parallel Dot Product: %d\n", dot);
    free(a);
    free(b);
    return 0;
}
```

OUTPUT:

```
PS C:\Users\Parshwa\Desktop\ASSIGN\HPC lab\22510064_HPC_A3> gcc -fopenmp dot_product.c -o dot_product
>> ./dot_product
Enter size of vectors: 5
Enter elements of first vector:
1 3 -2 5 6
Enter elements of second vector:
2 4 -4 6 -7
Parallel Dot Product: 10
PS C:\Users\Parshwa\Desktop\ASSIGN\HPC lab\22510064_HPC_A3>
```

Information and analysis:

The scalar product is calculated using dot += a[i] * b[i]; inside a parallel for loop.

reduction(+:dot) allows each thread to maintain a local copy of the variable and safely reduce it at the end.
schedule(static) distributes iterations evenly among threads, ideal for uniform workloads. Alternate schedules like dynamic and guided can be used to observe effects on performance.
With small vector sizes, speedup is not noticeable due to thread overhead, but on larger vectors, performance can improve.
Since the final result depends on all iterations, nowait and ordered are not typically used here.

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.

Screenshots:

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main() {
    int size, threads;
    printf("Enter matrix size (NxN): ");
    scanf("%d", &size);
    printf("Enter number of threads: ");
    scanf("%d", &threads);
    omp_set_num_threads(threads);
    int **A = malloc(size * sizeof(int*));
    int **B = malloc(size * sizeof(int*));
    int **C = malloc(size * sizeof(int*));
    for (int i = 0; i < size; i++) {
        A[i] = malloc(size * sizeof(int));
        B[i] = malloc(size * sizeof(int));
        C[i] = malloc(size * sizeof(int));
        for (int j = 0; j < size; j++) {
            A[i][j] = rand() % 10;
            B[i][j] = rand() % 10;
        }
    }
    double start = omp_get_wtime();
    #pragma omp parallel for collapse(2)
    for (int i = 0; i < size; i++)
        for (int j = 0; j < size; j++)
            C[i][j] = A[i][j] + B[i][j];
    double end = omp_get_wtime();
    printf("Matrix Size: %d, Threads: %d, Time: %f sec\n", size, threads, end - start);
    for (int i = 0; i < size; i++) {
        free(A[i]); free(B[i]); free(C[i]);
    }
    free(A); free(B); free(C);
    return 0;
}
```

```
Parallel Dot Product: 10
PS C:\Users\Parshwa\Desktop\ASSIGN\HPC lab\22510064_HPC_A3> gcc -fopenmp matrix_add.c -o matrix_add
PS C:\Users\Parshwa\Desktop\ASSIGN\HPC lab\22510064_HPC_A3> ./matrix_add
Enter matrix size (NxN): 500
Enter number of threads: 4
Matrix Size: 500, Threads: 4, Time: 0.001000 sec
PS C:\Users\Parshwa\Desktop\ASSIGN\HPC lab\22510064_HPC_A3>
```

Information and analysis:

Dynamic memory allocation is used to allow user-defined matrix sizes.
Addition of corresponding matrix elements is performed in parallel using a nested for loop.

collapse(2) clause merges both loops, enabling OpenMP to schedule all iterations in a flat loop.
Runtime is measured using omp_get_wtime() for benchmarking.
Performance increases with more threads, but speedup saturates due to factors like memory contention and overhead.
For small matrices (e.g., 250x250), speedup is minimal, but for large matrices (e.g., 1000x1000+), gains are significant.

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.

Screenshots:

vector_add.c > main()

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main() {
    int size, chunk, threads;
    printf("Enter vector size: ");
    scanf("%d", &size);
    printf("Enter chunk size: ");
    scanf("%d", &chunk);
    printf("Enter number of threads: ");
    scanf("%d", &threads);
    int *A = malloc(size * sizeof(int));
    for (int i = 0; i < size; i++) A[i] = rand() % 100;
    omp_set_num_threads(threads);
    omp_set_schedule(omp_sched_static, chunk);
    double start = omp_get_wtime();
    #pragma omp parallel
    {
        #pragma omp for schedule(runtime) nowait
        for (int i = 0; i < size; i++)
            A[i] += 5;
    }
    double end = omp_get_wtime();
    printf("Schedule: Static, Chunk: %d, Threads: %d, Time: %f sec\n", chunk, threads, end - start);
    omp_set_schedule(omp_sched_dynamic, chunk);
    start = omp_get_wtime();
    #pragma omp parallel
    {
        #pragma omp for schedule(runtime) nowait
        for (int i = 0; i < size; i++)
            A[i] += 5;
    }
    end = omp_get_wtime();
    printf("Schedule: Dynamic, Chunk: %d, Threads: %d, Time: %f sec\n", chunk, threads, end - start);
    free(A);
    return 0;
}
```

```
PS C:\Users\Parshwa\Desktop\ASSIGN\HPC lab\22510064_HPC_A3> gcc -fopenmp vector_add.c -o vector_add
PS C:\Users\Parshwa\Desktop\ASSIGN\HPC lab\22510064_HPC_A3> ./vector_add
Enter vector size: 1000
Enter chunk size: 50
Enter number of threads: 4
Schedule: Static, Chunk: 50, Threads: 4, Time: 0.001000 sec
Schedule: Dynamic, Chunk: 50, Threads: 4, Time: 0.000000 sec
PS C:\Users\Parshwa\Desktop\ASSIGN\HPC lab\22510064_HPC_A3>
```

Information and analysis:

Vector is dynamically allocated and initialized.
The scalar +5 operation is parallelized using schedule(runtime) which allows switching between static and dynamic during runtime using omp_set_schedule().
nowait clause skips implicit barrier after loop — useful if no synchronization is needed after the loop.
Timing data shows that static schedule performs better for evenly distributed work (like scalar add), while dynamic may perform better in unbalanced workloads.
Varying chunk size changes how many iterations each thread handles before asking for more — too small increases overhead; too large causes load imbalance.