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

Class: Final Year (Computer Science and Engineering)

Year: 2024-25 **Semester:** 1

Course: High Performance Computing Lab

Practical No. 3

Exam Seat No: 22520007

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:

Assign3 > C para.c

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

#define VECTOR_SIZE 10

void bubble_sort_ascending(int *arr, int size) {
    for (int i = 0; i < size - 1; i++) {
        for (int j = 0; j < size - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // Swap
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

void bubble_sort_descending(int *arr, int size) {
    for (int i = 0; i < size - 1; i++) {
        for (int j = 0; j < size - i - 1; j++) {
            if (arr[j] < arr[j + 1]) {
                // Swap
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

```

```
double compute_dot_product(int *a, int *b, int size) {
    double dot_product = 0.0;

    #pragma omp parallel for reduction(+:dot_product)
    for (int i = 0; i < size; i++) {
        dot_product += a[i] * b[i];
    }

    return dot_product;
}

int main() {

    srand((unsigned int)time(NULL));

    int *vector1 = (int*)malloc(VECTOR_SIZE * sizeof(int));
    int *vector2 = (int*)malloc(VECTOR_SIZE * sizeof(int));


    for (int i = 0; i < VECTOR_SIZE; i++) {
        vector1[i] = rand() % 100;
        vector2[i] = rand() % 100;
    }

    printf("Vector 1:\n");
    for (int i = 0; i < VECTOR_SIZE; i++) {
        printf("%d ", vector1[i]);
    }
    printf("\n");

    printf("Vector 2:\n");
    for (int i = 0; i < VECTOR_SIZE; i++) {
        printf("%d ", vector2[i]);
    }
    printf("\n");


    bubble_sort_ascending(vector1, VECTOR_SIZE);


    bubble_sort_descending(vector2, VECTOR_SIZE);

    double start = omp_get_wtime();
    double ans = compute_dot_product(vector1, vector2, VECTOR_SIZE);
    double end = omp_get_wtime();
```

Information and analysis:

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:

Sequential :

Assign3 > C matrixSeq.c

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

#define MAX_SIZE 2000

void initialize_matrix(int *matrix, int size) {
    for (int i = 0; i < size * size; i++) {
        matrix[i] = rand() % 100;
    }
}

void print_matrix(int *matrix, int size) {
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            printf("%d ", matrix[i * size + j]);
        }
        printf("\n");
    }
}

void matrix_addition_sequential(int *A, int *B, int *C, int size) {
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            C[i * size + j] = A[i * size + j] + B[i * size + j];
        }
    }
}

double measure_time(int size) {
    int *A = (int *)malloc(size * size * sizeof(int));
    int *B = (int *)malloc(size * size * sizeof(int));
    int *C = (int *)malloc(size * size * sizeof(int));

    initialize_matrix(A, size);
    initialize_matrix(B, size);

    double start_time = (double)clock() / CLOCKS_PER_SEC;
    matrix_addition_sequential(A, B, C, size);
    double end_time = (double)clock() / CLOCKS_PER_SEC;

    free(A);
    free(B);
    free(C);

    return end_time - start_time;
}
```

```c

int main() {
    int sizes[] = {250, 500, 750, 1000, 2000};

    printf("Matrix Size,Time(s)\n");

    for (int i = 0; i < sizeof(sizes) / sizeof(sizes[0]); i++) {
        int size = sizes[i];
        double time = measure_time(size);
        printf("%d,%f\n", size, time);
    }

    return 0;
}

```

```
wlug@wlug-optiplex:~/Desktop/22520007$ cd Assign3
wlug@wlug-optiplex:~/Desktop/22520007/Assign3$ ./matSeq
Matrix Size,Time(s)
250,0.000221
500,0.000993
750,0.002266
1000,0.003991
2000,0.015462
wlug@wlug-optiplex:~/Desktop/22520007/Assign3$
```

Information and analysis:

Parallel :

Assign3 > C matrixPara.c

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

#define MAX_SIZE 2000

void initialize_matrix(int *matrix, int size) {
    for (int i = 0; i < size * size; i++) {
        matrix[i] = rand() % 100;
    }
}

void print_matrix(int *matrix, int size) {
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            printf("%d ", matrix[i * size + j]);
        }
        printf("\n");
    }
}

void matrix_addition(int *A, int *B, int *C, int size, int num_threads) {
    #pragma omp parallel num_threads(num_threads)
    {
        #pragma omp for collapse(2)
        for (int i = 0; i < size; i++) {
            for (int j = 0; j < size; j++) {
                C[i * size + j] = A[i * size + j] + B[i * size + j];
            }
        }
    }
}

```

```c
double measure_time(int size, int num_threads) {
    int *A = (int *)malloc(size * size * sizeof(int));
    int *B = (int *)malloc(size * size * sizeof(int));
    int *C = (int *)malloc(size * size * sizeof(int));

    initialize_matrix(A, size);
    initialize_matrix(B, size);

    double start_time = omp_get_wtime();
    matrix_addition(A, B, C, size, num_threads);
    double end_time = omp_get_wtime();

    free(A);
    free(B);
    free(C);

    return end_time - start_time;
}

int main() {
    int sizes[] = {250, 500, 750, 1000, 2000};
    int num_threads_list[] = {2, 4, 8};

    printf("Matrix Size,Threads,Time(s)\n");

    for (int i = 0; i < sizeof(sizes) / sizeof(sizes[0]); i++) {
        int size = sizes[i];
        for (int j = 0; j < sizeof(num_threads_list) / sizeof(num_threads_list[0]); j++) {
            int num_threads = num_threads_list[j];
            double time = measure_time(size, num_threads);
            printf("%d,%d,%f\n", size, num_threads, time);
        }

        printf("============================================================\n");
    }

    return 0;
}
```

```
wlug@wlug-optiplex:~/Desktop/22520007/Assign3$ ./matPara
Matrix Size,Threads,Time(s)
250,2,0.000228
250,4,0.000175
250,8,0.000148
============================================================
500,2,0.000579
500,4,0.000490
500,8,0.000405
============================================================
750,2,0.001282
750,4,0.000721
750,8,0.000666
============================================================
1000,2,0.002054
1000,4,0.001229
1000,8,0.001484
============================================================
2000,2,0.008319
2000,4,0.004785
2000,8,0.005195
============================================================
wlug@wlug-optiplex:~/Desktop/22520007/Assign3$
```

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:

Assign3 > C p3.c

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

#define VECTOR_SIZE 200


void init_vec(int *vec, int size) {
    for (int i = 0; i < size; i++) {
        vec[i] = rand() % 100;
    }
}

void add_static(int *vec, int scalar, int size, int chunk) {
    #pragma omp parallel for schedule(static, chunk)
    for (int i = 0; i < size; i++) {
        vec[i] += scalar;
    }
}


void add_dynamic(int *vec, int scalar, int size, int chunk) {
    #pragma omp parallel for schedule(dynamic, chunk)
    for (int i = 0; i < size; i++) {
        vec[i] += scalar;
    }
}

void add_nowait(int *vec, int scalar, int size) {
    #pragma omp parallel
    {
        #pragma omp for nowait
        for (int i = 0; i < size; i++) {
            vec[i] += scalar;
        }
    }
}
```

```c
int main() {
    int vec[VECTOR_SIZE];
    int scalar = 10;
    int chunks[] = {1, 10, 50, 100};


    init_vec(vec, VECTOR_SIZE);

    printf("STATIC Schedule:\n");
    for (int i = 0; i < 4; i++) {
        double start_time = omp_get_wtime();
        add_static(vec, scalar, VECTOR_SIZE, chunks[i]);
        double end_time = omp_get_wtime();
        printf("Chunk size: %d, Time: %.6f seconds\n", chunks[i], end_time - start_time);
    }

    printf("\nDYNAMIC Schedule:\n");
    for (int i = 0; i < 4; i++) {
        double start_time = omp_get_wtime();
        add_dynamic(vec, scalar, VECTOR_SIZE, chunks[i]);
        double end_time = omp_get_wtime();
        printf("Chunk size: %d, Time: %.6f seconds\n", chunks[i], end_time - start_time);
    }
    printf("\nUsing nowait clause:\n");
    double start_time = omp_get_wtime();
    add_nowait(vec, scalar, VECTOR_SIZE);
    double end_time = omp_get_wtime();
    printf("Nowait clause, Time: %.6f seconds\n", end_time - start_time);

    return 0;
}
```

```
wlug@wlug-optiplex:~/Desktop/22520007/Assign3$ ./pr3
STATIC Schedule:
Chunk size: 1, Time: 0.001268 seconds
Chunk size: 10, Time: 0.000014 seconds
Chunk size: 50, Time: 0.000010 seconds
Chunk size: 100, Time: 0.000010 seconds

DYNAMIC Schedule:
Chunk size: 1, Time: 0.000032 seconds
Chunk size: 10, Time: 0.000010 seconds
Chunk size: 50, Time: 0.000012 seconds
Chunk size: 100, Time: 0.000008 seconds

Using nowait clause:
Nowait clause, Time: 0.000010 seconds
wlug@wlug-optiplex:~/Desktop/22520007/Assign3$
```

Information and analysis: