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

Course: High Performance Computing Lab

Practical No. 3

PRN: 22510057

Name: Ashutosh Gundu Birje

Batch: B8

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:

Assignment_No_3 > C 01_.c

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

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

    int *a = (int *)malloc(n * sizeof(int));
    int *b = (int *)malloc(n * sizeof(int));
    int dot_product = 0;

    printf("Enter elements of vector A:\n");
    for (int i = 0; i < n; i++) scanf("%d", &a[i]);

    printf("Enter elements of vector B:\n");
    for (int i = 0; i < n; i++) scanf("%d", &b[i]);

    double start = omp_get_wtime();

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

    double end = omp_get_wtime();

    printf("Minimum scalar product (dot product) = %d\n", dot_product);
    printf("Time taken: %lf seconds\n", end - start);

    free(a);
```

```
ashutosh@ash-880:~/Desktop/HPC-LAB$ cd Assignment_No_3
ashutosh@ash-880:~/Desktop/HPC-LAB/Assignment_No_3$ gcc -fopenmp 01_.c -o 01.exe
ashutosh@ash-880:~/Desktop/HPC-LAB/Assignment_No_3$ ./01.exe
Enter size of vectors: 3
Enter elements of vector A:
1 2 3
Enter elements of vector B:
1 2 3
Minimum scalar product (dot product) = 14
Time taken: 0.011485 seconds
ashutosh@ash-880:~/Desktop/HPC-LAB/Assignment_No_3$
```

Information and analysis:

Approach - Parallelized dot product using reduction to avoid race conditions.
Time complexity - O(n)
Analysis - Performance improves with vector size and higher threads. Reduction clause helps in combining partial results efficiently.

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:

Assignment_No_3 > C 02_.c

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

void matrix_add(int **a, int **b, int **c, int n) {
    #pragma omp parallel for collapse(2)
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            c[i][j] = a[i][j] + b[i][j];
}

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

    for (int t = 2; t <= 8; t *= 2) {
        omp_set_num_threads(t);
        printf("\nThreads: %d\n", t);
        for (int s = 0; s < 5; s++) {
            int n = sizes[s];
            int **a = malloc(n * sizeof(int *));
            int **b = malloc(n * sizeof(int *));
            int **c = malloc(n * sizeof(int *));
            for (int i = 0; i < n; i++) {
                a[i] = malloc(n * sizeof(int));
                b[i] = malloc(n * sizeof(int));
                c[i] = malloc(n * sizeof(int));
                for (int j = 0; j < n; j++) {
                    a[i][j] = rand() % 100;
                    b[i][j] = rand() % 100;
                }
            }
```

```
ashutosh@ash-880:~/Desktop/HPC-LAB$ cd Assignment_No_3
ashutosh@ash-880:~/Desktop/HPC-LAB/Assignment_No_3$ gcc -fopenmp 02_.c -o 02.exe
ashutosh@ash-880:~/Desktop/HPC-LAB/Assignment_No_3$ ./02.exe

Threads: 2
Matrix Size: 250 x 250, Time: 0.000207 seconds
Matrix Size: 500 x 500, Time: 0.000571 seconds
Matrix Size: 750 x 750, Time: 0.000916 seconds
Matrix Size: 1000 x 1000, Time: 0.001258 seconds
Matrix Size: 2000 x 2000, Time: 0.005322 seconds

Threads: 4
Matrix Size: 250 x 250, Time: 0.000223 seconds
Matrix Size: 500 x 500, Time: 0.000325 seconds
Matrix Size: 750 x 750, Time: 0.000511 seconds
Matrix Size: 1000 x 1000, Time: 0.000852 seconds
Matrix Size: 2000 x 2000, Time: 0.002925 seconds

Threads: 8
Matrix Size: 250 x 250, Time: 0.000189 seconds
Matrix Size: 500 x 500, Time: 0.000192 seconds
Matrix Size: 750 x 750, Time: 0.000372 seconds
Matrix Size: 1000 x 1000, Time: 0.000551 seconds
Matrix Size: 2000 x 2000, Time: 0.002364 seconds
ashutosh@ash-880:~/Desktop/HPC-LAB/Assignment_No_3$
```

Information and analysis:

Expected Behavior-
Speedup improves with larger matrix size and thread count, up to a limit.
Scaling Limit-
Diminishing returns beyond 4-8 threads due to memory/cache bottlenecks.

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:

Assignment_No_3 > C 03_.c

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

#define SIZE 200

void static_schedule_add(int *a, int *res, int scalar, int chunk) {
    #pragma omp parallel for schedule(static, chunk)
    for (int i = 0; i < SIZE; i++) {
        res[i] = a[i] + scalar;
    }
}

void dynamic_schedule_add(int *a, int *res, int scalar, int chunk) {
    #pragma omp parallel for schedule(dynamic, chunk)
    for (int i = 0; i < SIZE; i++) {
        res[i] = a[i] + scalar;
    }
}

void nowait_demo(int *a, int *b, int *res) {
    #pragma omp parallel
    {
        #pragma omp for nowait
        for (int i = 0; i < SIZE/2; i++)
            res[i] = a[i] + b[i];

        #pragma omp for nowait
        for (int i = SIZE/2; i < SIZE; i++)
            res[i] = a[i] + b[i];
    }
}
```

Assignment_No_3 > C 03_.c

```c
int main() {
    int a[SIZE], res[SIZE], b[SIZE];
    for (int i = 0; i < SIZE; i++) {
        a[i] = i;
        b[i] = 2 * i;
    }

    int scalar = 5;
    int chunk_sizes[] = {1, 5, 10, 20, 50};

    for (int c = 0; c < 5; c++) {
        int chunk = chunk_sizes[c];

        double start = omp_get_wtime();
        static_schedule_add(a, res, scalar, chunk);
        double end = omp_get_wtime();
        printf("STATIC Chunk %d: %lf seconds\n", chunk, end - start);

        start = omp_get_wtime();
        dynamic_schedule_add(a, res, scalar, chunk);
        end = omp_get_wtime();
        printf("DYNAMIC Chunk %d: %lf seconds\n", chunk, end - start);
    }

    printf("\nNOWAIT Clause demo:\n");
    nowait_demo(a, b, res);

    return 0;
}
```

```
ashutosh@ash-880:~/Desktop/HPC-LAB$ cd Assignment_No_3
ashutosh@ash-880:~/Desktop/HPC-LAB/Assignment_No_3$ gcc -fopenmp 03_.c -o 03.exe
ashutosh@ash-880:~/Desktop/HPC-LAB/Assignment_No_3$ ./03.exe
STATIC Chunk 1: 0.005944 seconds
DYNAMIC Chunk 1: 0.006979 seconds
STATIC Chunk 5: 0.002947 seconds
DYNAMIC Chunk 5: 0.000007 seconds
STATIC Chunk 10: 0.000002 seconds
DYNAMIC Chunk 10: 0.000002 seconds
STATIC Chunk 20: 0.000001 seconds
DYNAMIC Chunk 20: 0.000002 seconds
STATIC Chunk 50: 0.000001 seconds
DYNAMIC Chunk 50: 0.000002 seconds

NOWAIT Clause demo:
ashutosh@ash-880:~/Desktop/HPC-LAB/Assignment_No_3$
```

Information and analysis:

STATIC Schedule-

Best when iteration workload is uniform.

DYNAMIC Schedule-

Helps when workload is unpredictable or uneven.

Chunk Size-

Impacts load balancing and thread overhead.

Nowait Clause-

Prevents implicit barrier, useful for independent tasks.

Github Link:

https://github.com/Ashutoshbirje/HPC-LAB/tree/master/Assignment_No_3