20BCE1550
Samridh Anand Paatni
CSE4001 Lab 03
Parallel ‘for’ Loops and Parallel Sorting

Q1.

Code:

q1_dynamic.c

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


int main(int argc, char ** argv) {
    int numThreads = atoi(argv[1]);
    int chunkSize = atoi(argv[2]);
    int limit = 10000;
    int oddSum = 0;
    int evenSum = 0;
    int i = 0;
    clock_t t = clock();

    #pragma omp parallel shared(evenSum, chunkSize) private(i) num_threads(numThreads)
    {
        #pragma omp parallel for schedule(dynamic, chunkSize) reduction(+ : evenSum)
        for (i = 0; i < limit; i += 2) evenSum += i;
    }

    #pragma omp parallel shared(oddSum, chunkSize) private(i) num_threads(numThreads)
    {
        #pragma omp parallel for schedule(dynamic, chunkSize) reduction(+ : oddSum)
        for (i = 1; i < limit; i += 2) oddSum += i;
    }

    t = clock() - t;

    printf(
        "static scheduling | %fs | %d threads | chunk size = %d | oddSum: %d | evenSum: %d
",
        ((double)t/CLOCKS_PER_SEC),
        numThreads,
        chunkSize,
        oddSum,
        evenSum
    );

    printf("\n\n");
```

```
    return 0;
}
```

Q1_other.c

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

int main(int argc, char ** argv) {
    int numThreads = atoi(argv[1]);
    int limit = 10000;
    int oddSum = 0;
    int evenSum = 0;
    int i = 0;
    clock_t t = clock();

    #pragma omp parallel shared(evenSum) private(i) num_threads(numThreads)
    {
        #pragma omp parallel for schedule(static)
        for (i = 0; i < limit; i += 2) {
            #pragma omp critical
            {
                evenSum += i;
            }
        }
    }

    #pragma omp parallel shared(oddSum) private(i) num_threads(numThreads)
    {
        #pragma omp parallel for schedule(static)
        for (i = 1; i < limit; i += 2) {
            #pragma omp critical
            {
                oddSum += i;
            }
        }
    }

    t = clock() - t;

    printf(
        "dynamic scheduling | %fs | %d threads | oddSum: %d | evenSum: %d ",
        ((double)t/CLOCKS_PER_SEC),
        numThreads,
        oddSum,
        evenSum
    );

    printf("\n\n");
    return 0;
```

```
}
```

Output:

```
~/vit/CSE4001_Parallel-and_Distributed-Computing_ETLP/Lab/20220816 ──── sam fedora-hp-2020:pts/0
(21:57:15)──> make q1                                                    ──(Sun,Aug21)
gcc q1_dynamic.c -fopenmp -o q1_dynamic.out
./q1_dynamic.out 32 10
static scheduling | 0.008277s | 32 threads | chunk size = 10 | oddSum: 175000000 | evenSum: 24995000

./q1_dynamic.out 64 10
static scheduling | 0.011826s | 64 threads | chunk size = 10 | oddSum: 450000000 | evenSum: 24995000

./q1_dynamic.out 128 10
static scheduling | 0.021322s | 128 threads | chunk size = 10 | oddSum: 925000000 | evenSum: 24995000

./q1_dynamic.out 32 20
static scheduling | 0.004916s | 32 threads | chunk size = 20 | oddSum: 150000000 | evenSum: 24995000

./q1_dynamic.out 64 20
static scheduling | 0.012515s | 64 threads | chunk size = 20 | oddSum: 450000000 | evenSum: 24995000

./q1_dynamic.out 128 20
static scheduling | 0.019191s | 128 threads | chunk size = 20 | oddSum: 525000000 | evenSum: 24995000

./q1_dynamic.out 32 50
static scheduling | 0.005103s | 32 threads | chunk size = 50 | oddSum: 200000000 | evenSum: 24995000

./q1_dynamic.out 64 50
static scheduling | 0.011121s | 64 threads | chunk size = 50 | oddSum: 450000000 | evenSum: 24995000

./q1_dynamic.out 128 50
static scheduling | 0.020333s | 128 threads | chunk size = 50 | oddSum: 1125000000 | evenSum: 24995000

gcc q1_others.c -fopenmp -o q1_others.out
./q1_others.out 32
dynamic scheduling | 0.305699s | 32 threads | oddSum: 800000000 | evenSum: 799840000

./q1_others.out 64
dynamic scheduling | 0.713230s | 64 threads | oddSum: 1600000000 | evenSum: 1599680000

./q1_others.out 128
dynamic scheduling | 1.503536s | 128 threads | oddSum: -1094967296 | evenSum: -1095607296
```

Q2.

Code:

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

#define N 10000
#define NUM_THREADS 4

/*
2. Write a parallel program to sort N elements in an array using OpenMP
```

```
    i. Bubble Sort
    ii. Quick Sort
*/

void serialBubbleSort(int array[]) {
    int i, temp;
    for (int times = 0; times < N; times++) {
        for (int i = 0; i < N - 1; i ++) {
            if (array[i] > array[i + 1]) {
                temp = array[i];
                array[i] = array[i + 1];
                array[i + 1] = temp;
            }
        }
    }
}

void parallelBubbleSort(int array[]) {
    int i, temp;

    for (int times = 0; times < N; times++) {
        #pragma omp parallel num_threads(NUM_THREADS) shared(array) private(i, temp)
        {
            // even phase
            #pragma omp for schedule(static)
            for (int i = 0; i < N - 1; i += 2) {
                if (array[i] > array[i + 1]) {
                    temp = array[i];
                    array[i] = array[i + 1];
                    array[i + 1] = temp;
                }
            }

            // odd phase
            #pragma omp for schedule(static)
            for (int i = 1; i < N - 1; i += 2) {
                if (array[i] > array[i + 1]) {
                    temp = array[i];
                    array[i] = array[i + 1];
                    array[i + 1] = temp;
                }
            }
        }
    }
}

int partition (int a[], int start, int end) {
    int pivot = a[end];
    int i = start - 1;
    for (int j = start; j <= end - 1; j++) {
        if (a[j] < pivot) {
            i++;
            int t = a[i];
            a[i] = a[j];
```

```
            a[j] = t;
        }
    }
    int t = a[i+1];
    a[i+1] = a[end];
    a[end] = t;
    return (i + 1);
}

void serialQuickSortHelper(int a[], int start, int end){
    if (start < end){
        int p = partition(a, start, end);
        serialQuickSortHelper(a, start, p - 1);
        serialQuickSortHelper(a, p + 1, end);
    }
}

void serialQuickSort(int array[]) {
    serialQuickSortHelper(array, 0, N - 1);
}

void parallelQuickSortHelper(int a[], int start, int end) {
    if (start < end) {
        int p = partition(a, start, end);
        #pragma omp parallel sections
        {
            #pragma omp section
            {
                parallelQuickSortHelper(a, start, p - 1);
            }
            #pragma omp section
            {
                parallelQuickSortHelper(a, p + 1, end);
            }
        }
    }
}

void parallelQuickSort(int array[]) {
    parallelQuickSortHelper(array, 0, N - 1);
}

void print(int a[]) {
    for (int i = 0; i < N; i++) {
        printf(
            "%s%d%s",
            i == 0 ? "[" : "",
            a[i],
            i == N - 1 ? "]" : ", "
        );
    }
    printf("\n");
}
```

```c
void testAlgorithm(char name[], void (*algorithm)(int *), int array[]) {
    clock_t t = clock();

    algorithm(array);

    t = clock() - t;

    // print(array);

    printf(
        "%s took %f seconds\n",
        name,
        ((double)t) / CLOCKS_PER_SEC
    );
    printf("\n");
}

int main(int argc, char ** argv) {
    int a[N], b[N], c[N], d[N];
    srand(time(NULL));

    // generate random numbers to sort
    for (int i = 0; i < N; i++) {
        int r = rand() % 1000;
        a[i] = r;
        b[i] = r;
        c[i] = r;
        d[i] = r;
    }

    // show the random array
    // printf("Original Array:\n");
    // print(a);
    // printf("\n");

    // run each algorithm
    testAlgorithm("Serial Bubble Sort", serialBubbleSort, a);
    testAlgorithm("Parallel Bubble Sort", parallelBubbleSort, b);
    testAlgorithm("Serial Quicksort", serialQuickSort, c);
    testAlgorithm("Parallel Quicksort", parallelQuickSort, d);

    return 0;
}
```

Output:

```
gcc q2.c -fopenmp -o q2.out
./q2.out
Serial Bubble Sort took 42.511410 seconds

Parallel Bubble Sort took 47.823454 seconds

Serial Quicksort took 0.026261 seconds

Parallel Quicksort took 0.112549 seconds
```