**20BCE1550**

**Samridh Anand Paatni**

**CSE4001 Lab 03**

**Parallel ‘for’ Loops and Parallel Sorting**

**Q1.**

**Code:**

**q1\_dynamic.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

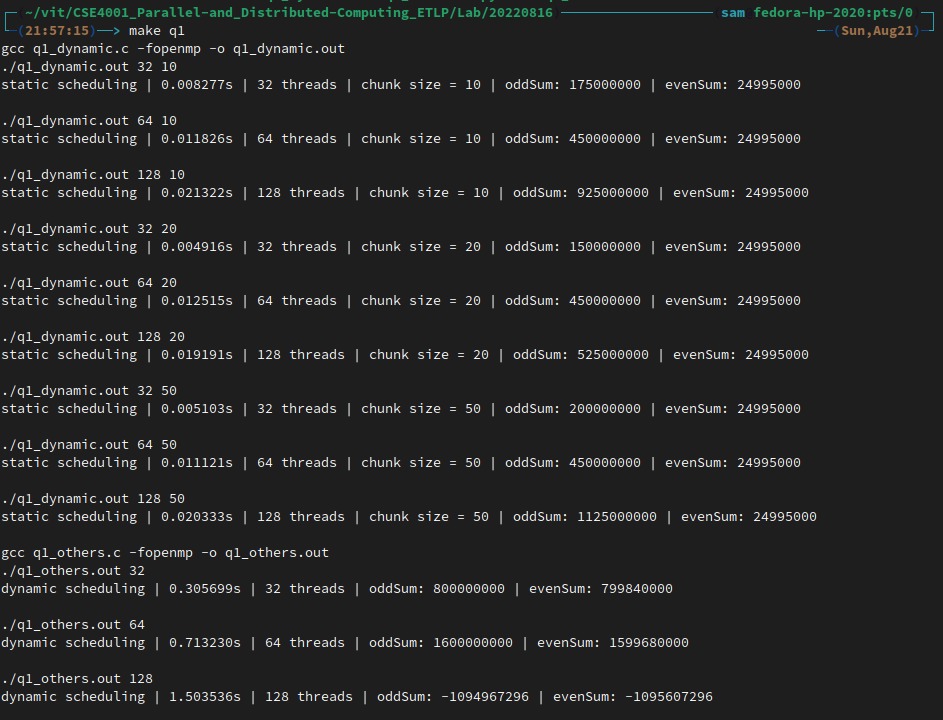
    );

    printf("\n\n");

    return 0;

}

**Output:**

****

**Q2.**

**Code:**

#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");

}

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:**

