**20BCE1550**

**Samridh Anand Paatni**

**CSE4001 Lab 06**

**Reductions**

**Q1. Write a program in OpenMP to find out the largest number in an array of 1000000 randomly generated numbers from 1 to 100000 using reduction clause. Compare the versions of serial, parallel for and reduction clause.**

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

#include <time.h>

#define N 1000000

#define MAX 100000

void findMinSerial(int ar[]) {

    int max = 0;

    for (int i = 0; i < N; i++) if (ar[i] > max) max = ar[i];

    printf("%d is the maximum element\n", max);

}

void findMinParallel(int ar[]) {

    int max = 0, i;

    #pragma omp parallel for shared(ar, max) private(i)

    for (i = 0; i < N; i++) {

        if (ar[i] > max) {

            #pragma omp critical

            {

                max = ar[i];

            }

        }

    }

    printf("%d is the maximum element\n", max);

}

void findMinReduction(int ar[]) {

    int max = 0, i;

    #pragma omp parallel for shared(ar) private(i) reduction(max: max)

    for (i = 0; i < N; i++) if (ar[i] > max) max = ar[i];

    printf("%d is the maximum element\n", max);

}

int main() {

    int ar[N];

    srand(clock());

    for (int i = 0; i < N; i++) ar[i] = rand() % MAX;

    double t = omp\_get\_wtime();

    findMinSerial(ar);

    t = omp\_get\_wtime() - t;

    printf("Serial execution took %fs\n", t);

    t = omp\_get\_wtime();

    findMinParallel(ar);

    t = omp\_get\_wtime() - t;

    printf("Parallel execution using a critical section took %fs\n", t);

    t = omp\_get\_wtime();

    findMinReduction(ar);

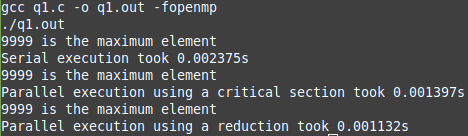
    t = omp\_get\_wtime() - t;

    printf("Parallel execution using a reduction took %fs\n", t);

    return 0;

}

**Output:**

****

**Q2. Write a program in OpenMP to find out the standard deviation of 1000000 randomly generated numbers using reduction clause. Document the development versions of serial, parallel for and reduction clause.**

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <math.h>

#include <omp.h>

#define N 1000000

#define MAX 100000

void sdSerial(int ar[]) {

    double mean = 0;

    for (int i = 0; i < N; i++) mean += ar[i];

    mean /= N;

    double runningSum = 0;

    for (int i = 0; i < N; i++) {

        runningSum += pow(ar[i] - mean, 2);

    }

    double sd = sqrt(runningSum / N);

    printf("sd = %f\n", sd);

}

void sdParallel(int ar[]) {

    double mean = 0;

    int i = 0;

    #pragma omp parallel for shared(ar, mean) private(i)

    for (i = 0; i < N; i++) {

        #pragma omp critical

        {

            mean += ar[i];

        }

    }

    mean /= N;

    double runningSum = 0;

    #pragma omp parallel for shared(runningSum, mean) private(i)

    for (i = 0; i < N; i++) {

        #pragma omp critical

        {

            runningSum += pow(ar[i] - mean, 2);

        }

    }

    double sd = sqrt(runningSum / N);

    printf("sd = %f\n", sd);

}

void sdReduction(int ar[]) {

    double mean = 0;

    int i = 0;

    #pragma omp parallel for shared(ar) private(i) reduction(+ : mean)

    for (i = 0; i < N; i++) mean += ar[i];

    mean /= N;

    double runningSum = 0;

    #pragma omp parallel for shared(ar, mean) private(i) reduction(+ : runningSum)

    for (i = 0; i < N; i++) runningSum += pow(ar[i] - mean, 2);

    double sd = sqrt(runningSum / N);

    printf("sd = %f\n", sd);

}

int main() {

    int ar[N];

    srand(clock());

    for (int i = 0; i < N; i++) ar[i] = rand() % MAX;

    double t = omp\_get\_wtime();

    sdSerial(ar);

    t = omp\_get\_wtime() - t;

    printf("Serial execution took %fs\n\n", t);

    t = omp\_get\_wtime();

    sdParallel(ar);

    t = omp\_get\_wtime() - t;

    printf("Parallel execution using a critical section took %fs\n\n", t);

    t = omp\_get\_wtime();

    sdReduction(ar);

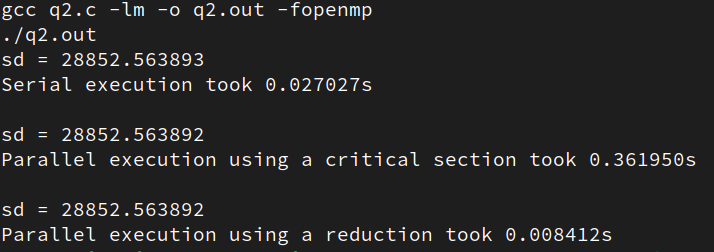
    t = omp\_get\_wtime() - t;

    printf("Parallel execution using a reduction took %fs\n", t);

    return 0;

}

**Output:**

****

**Q3. Write a multithreaded program using OpenMP to implement sequential and parallel version of the Monte Carlo algorithm for approximating Pi. Compare the results of sequential, loop-level parallelism and reduction clause with 10000000 samples.**

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

#include <math.h>

#include <time.h>

#define N 100000

#define M 100000

void serialMonteCarloPi(double xSamples[], double ySamples[]) {

    int counter = 0;

    for (int i = 0; i < N; i++) {

        double x = xSamples[i];

        double y = ySamples[i];

        if (x \* x + y \* y < 1) counter++;

    }

    printf("pi = %f\n", 4.0 \* (double) counter / (double) N);

}

void parallelMonteCarloPi(double xSamples[], double ySamples[]) {

    int counter = 0, i;

    double x, y;

    #pragma omp parallel for shared(xSamples, ySamples, counter) private(i, x, y)

    for (i = 0; i < N; i++) {

        x = xSamples[i];

        y = ySamples[i];

        if (x \* x + y \* y < 1) {

            #pragma omp critical

            {

                counter++;

            }

        }

    }

    printf("pi = %f\n", 4.0 \* (double) counter / (double) N);

}

void reductionMonteCarloPi(double xSamples[], double ySamples[]) {

    int counter = 0, i;

    double x, y;

    #pragma omp parallel for shared(xSamples, ySamples) private(i, x, y) reduction(+: counter)

    for (i = 0; i < N; i++) {

        x = xSamples[i];

        y = ySamples[i];

        if (x \* x + y \* y < 1) counter += 1;

    }

    printf("pi = %f\n", 4.0 \* (double) counter / (double) N);

}

int main() {

    srand(clock());

    double xSamples[N], ySamples[N];

    for (int i = 0; i < N; i++) {

        xSamples[i] = (double)(rand() % M) / M;

        ySamples[i] = (double)(rand() % M) / M;

    }

    double t = omp\_get\_wtime();

    serialMonteCarloPi(xSamples, ySamples);

    t = omp\_get\_wtime() - t;

    printf("Serial execution took %fs\n\n", t);

    t = omp\_get\_wtime();

    parallelMonteCarloPi(xSamples, ySamples);

    t = omp\_get\_wtime() - t;

    printf("Parallel execution using a critical section took %fs\n\n", t);

    t = omp\_get\_wtime();

    reductionMonteCarloPi(xSamples, ySamples);

    t = omp\_get\_wtime() - t;

    printf("Parallel execution using a reduction took %fs\n", t);

}

**Output:**

