**20BCE1025**

**Abhishek N N**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | | | | | |
|  | | | | | |
|  | | | | | |
| Programme | : | **B.Tech.(CSE)** | Semester | : | **Fall ’22-23** | |
| Course | : | **Parallel and Distributed Computing** | Code | : | **CSE4001** | |
| Faculty | : | **R. Kumar** | Slot | : | **L9+L10** | |

1. 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 <limits.h>

#include <omp.h>

#include <stdio.h>

#include <stdlib.h>

int main(void) {

omp\_set\_num\_threads(100);

int arr[1000000];

for (int i = 0; i < 1000000; i++) arr[i] = rand() % 100000;

// serial

double start = omp\_get\_wtime();

int m = INT\_MIN;

for (int i = 0; i < 1000000; i++) {

if (arr[i] > m) {

m = arr[i];

}

}

printf("serial max: %d in %f seconds\n", m, omp\_get\_wtime() - start);

// parallel

start = omp\_get\_wtime();

m = INT\_MIN;

#pragma omp parallel for shared(arr, m)

for (int i = 0; i < 1000000; i++) {

if (arr[i] > m) {

m = arr[i];

}

}

printf("parallel max: %d in %f seconds\n", m, omp\_get\_wtime() - start);

// reduction

start = omp\_get\_wtime();

m = INT\_MIN;

#pragma omp parallel for reduction(max : m)

for (int i = 0; i < 1000000; i++) {

if (arr[i] > m) {

m = arr[i];

}

}

printf("reduction max: %d in %f seconds\n", m, omp\_get\_wtime() - start);

return 0;

}

**Output:**

on first run



on second run



on third run



**Comparision:**

To get better results I runned code 3 times

It is evident that **serial** is slowest in all times and

**parallel** is faster but we got lucky particularly here and there was no synchronization issue (usually **parallel** gives wrong output when no synchronization constraints such as critical or reduction)

**reduction clause** was the fastest taking full advantage of parallelism

1. 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.

Documentation is done in code as comments

**Code:**

#include <limits.h>

#include <omp.h>

#include <stdio.h>

#include <stdlib.h>

#include<math.h>

// count of randomly generated numbers

#define N 1000000

int main(void) {

// setting no of omp threads

omp\_set\_num\_threads(100);

// creating array with random numbers

int arr[N];

for (int i = 0; i < N; i++) arr[i] = rand() % 100000;

// serial part

double start = omp\_get\_wtime();

double mean = 0;

// summing up

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

// now dividing by N to get mean

mean/=N;

// lets now find standard deviation

double sd=0;

// summing up squares of difference

for (int i=0;i<N;i++) sd+=pow(arr[i]-mean,2);

// dividing by N

sd/=N;

// taking square root

sd=sqrt(sd);

printf("serial standard deviation: %f in %f seconds\n", sd, omp\_get\_wtime() - start);

// parallel part

start = omp\_get\_wtime();

mean = 0;

// summing up with mean and arr as shared

#pragma omp parallel for shared(arr,mean)

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

// now dividing by N to get mean

mean/=N;

// lets now find standard deviation

sd=0;

// summing up squares of difference

// with mean and arr as shared

#pragma omp parallel for shared(arr,mean)

for (int i=0;i<N;i++) sd+=pow(arr[i]-mean,2);

// dividing by N

sd/=N;

// taking square root

sd=sqrt(sd);

printf("parallel standard deviation: %f in %f seconds\n", sd, omp\_get\_wtime() - start);

// reduction part

start = omp\_get\_wtime();

mean = 0;

// summing up with arr as shared and mean with reduction

#pragma omp parallel for shared(arr) reduction(+:mean)

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

// now dividing by N to get mean

mean/=N;

// lets now find standard deviation

sd=0;

// summing up squares of difference

// with arr as shared and mean with reduction

#pragma omp parallel for shared(arr) reduction(+:sd)

for (int i=0;i<N;i++) sd+=pow(arr[i]-mean,2);

// dividing by N

sd/=N;

// taking square root

sd=sqrt(sd);

printf("parallel standard deviation: %f in %f seconds\n", sd, omp\_get\_wtime() - start);

return 0;

}

**Output:**

on first run



on Second run



on third run



**Comparision:**

To get better results I runned code 3 times

It is evident that **serial** is slowest and gave correct output in all times and

**parallel** is faster but we got wrong output when no synchronization constraints such as critical or reduction is used

**reduction clause** was the fastest taking full advantage of parallelism and giving correct output

1. 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 <math.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <omp.h>

// for rand() function

#define SEED 35791246

#define niter 100000

int main() {

// serial

double start = omp\_get\_wtime();

srand(SEED); // initialize random numbers

int count = 0;

for (int i = 0; i < niter; i++) {

double x = (double)rand() / RAND\_MAX;

double y = (double)rand() / RAND\_MAX;

double z = x \* x + y \* y;

if (z <= 1) count++;

}

double pi = (double)count / niter \* 4;

printf("serial pi : %g with time %f seconds\n", pi, omp\_get\_wtime() - start);

// parallel (loop level)

start = omp\_get\_wtime();

srand(SEED); // initialize random numbers

count = 0;

#pragma omp parallel for shared(count)

for (int i = 0; i < niter; i++) {

double x = (double)rand() / RAND\_MAX;

double y = (double)rand() / RAND\_MAX;

double z = x \* x + y \* y;

if (z <= 1) count++;

}

pi = (double)count / niter \* 4;

printf("parallel pi : %f with time %f seconds\n", pi, omp\_get\_wtime() - start);

// reduction

start = omp\_get\_wtime();

srand(SEED); // initialize random numbers

count = 0;

#pragma omp parallel for reduction(+ : count)

for (int i = 0; i < niter; i++) {

double x = (double)rand() / RAND\_MAX;

double y = (double)rand() / RAND\_MAX;

double z = x \* x + y \* y;

if (z <= 1) count++;

}

pi = (double)count / niter \* 4;

printf("reduction pi : %f with time %f seconds\n", pi, omp\_get\_wtime() - start);

return 0;

}

**Output:**

on first run



on second run



on third run



**Comparision:**

To get better results I runned code 3 times

It is evident that **serial(sequential)** is fastest(due to excessive overhead in parallel systems) and gave correct output in all times and

**parallel** is slowest and we got wrong output when no synchronization constraints such as critical or reduction is used

**reduction clause** was the faster taking advantage of parallelism and giving correct output near to serial