20BCE1025

Abhishek N N

VIT®
UNIVERSITY
(Estd. u/s 3 of UGC Act 1956)
www.vit.ac.in
Vellore ▪ Chennai
CHENNAI CAMPUS
Vandalur - Kelambakkam Road, Chennai - 600127

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.

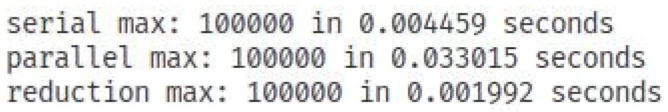
Code:

```
#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;
}
```

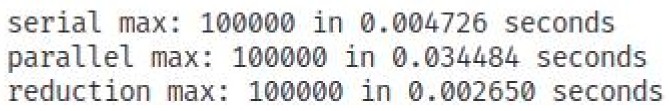
Output:

on first run



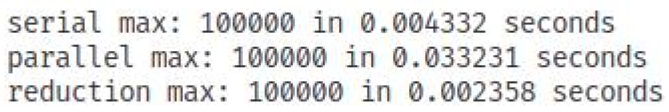
```
serial max: 100000 in 0.004459 seconds
parallel max: 100000 in 0.033015 seconds
reduction max: 100000 in 0.001992 seconds
```

on second run



```
serial max: 100000 in 0.004726 seconds
parallel max: 100000 in 0.034484 seconds
reduction max: 100000 in 0.002650 seconds
```

on third run



```
serial max: 100000 in 0.004332 seconds
parallel max: 100000 in 0.033231 seconds
reduction max: 100000 in 0.002358 seconds
```

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

2. 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.

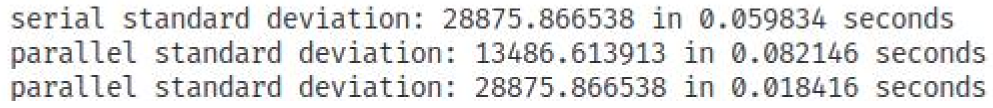
Documentation is done in code as comments

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;
}
```

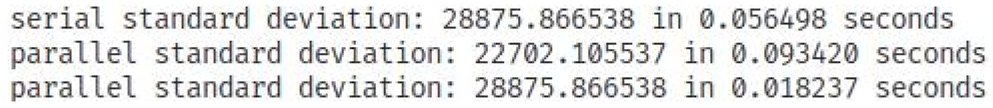
Output:

on first run



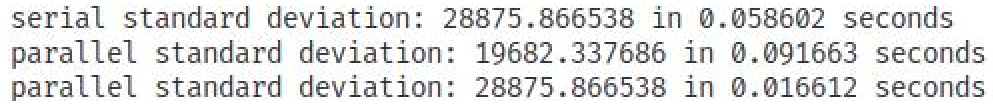
```
serial standard deviation: 28875.866538 in 0.059834 seconds
parallel standard deviation: 13486.613913 in 0.082146 seconds
parallel standard deviation: 28875.866538 in 0.018416 seconds
```

on Second run



```
serial standard deviation: 28875.866538 in 0.056498 seconds
parallel standard deviation: 22702.105537 in 0.093420 seconds
parallel standard deviation: 28875.866538 in 0.018237 seconds
```

on third run



```
serial standard deviation: 28875.866538 in 0.058602 seconds
parallel standard deviation: 19682.337686 in 0.091663 seconds
parallel standard deviation: 28875.866538 in 0.016612 seconds
```

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

3. 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.

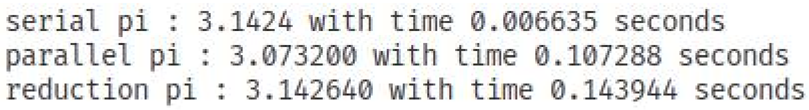
Code :

```
#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;
}
```

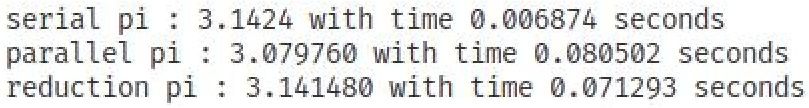
Output:

on first run



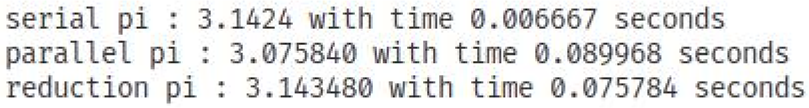
```
serial pi : 3.1424 with time 0.006635 seconds
parallel pi : 3.073200 with time 0.107288 seconds
reduction pi : 3.142640 with time 0.143944 seconds
```

on second run



```
serial pi : 3.1424 with time 0.006874 seconds
parallel pi : 3.079760 with time 0.080502 seconds
reduction pi : 3.141480 with time 0.071293 seconds
```

on third run



```
serial pi : 3.1424 with time 0.006667 seconds
parallel pi : 3.075840 with time 0.089968 seconds
reduction pi : 3.143480 with time 0.075784 seconds
```

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