LAB 03

IT23464032

```
[ec2-user@ip-172-31-33-146 lab02]$ gcc pi.c -o pi_serial -lm
./pi_serial
Final Estimation of Pi = 3.142200
[ec2-user@ip-172-31-33-146 lab02]$
```

```
// C program for the above approach

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

// Function to find estimated
// value of PI using Monte // Carlo algorithm
void monteCarlo(int N, int K)
{
// Stores X and Y coordinates
// of a random point double x, y;
// Stores distance of a random
// point from origin double d;

// Stores number of points // lying inside circle
int pCircle = 0;

// Stores number of points // lying inside square
int pSquare = 0;
 int i = 0;

// Parallel calculation of random // points lying inside a
circle
#pragma omp parallel firstprivate(x, y, d, i) reduction(+ : pCircle, pSquare) num_threads(K)
{
// Initializes random points
// with a seed
srand48((int)time(NULL));

for (i = 0; i < N; i++) {
// Finds random X co-ordinate x = (double)drand48();

// Finds random X co-ordinate
```

```
y = (double)drand48();

// Finds the square of distance
// of point (x, y) from origin d
= ((x * x) + (y * y));

// If d is less than or
// equal to 1
if (d <= 1) {
// Increment pCircle by 1 pCircle++;
}
// Increment pSquare by 1 pSquare++;
}
}
// Stores the estimated value of PI double pi = 4.0 *
((double)pCircle / (double)(pSquare));

// Prints the value in pi
printf("Final Estimation of Pi = %f\n",
pi); }
// Driver Code
int main()
{
// Input int N =
100000; int K =
8; // Function
call
monteCarlo(N, K);
}
```

```
[ec2-user@ip-172-31-33-146 lab02]$ gcc -fopenmp pi.c -o lab02.o -lm
[ec2-user@ip-172-31-33-146 lab02]$ ./lab02.o
Serial run with 1 threads → Pi = 3.1417484000, Time = 0.149038 sec
Parallel run with 1 threads → Pi = 3.1417484000, Time = 0.147780 sec
Parallel run with 2 threads → Pi = 3.1417766000, Time = 0.148203 sec
Speedup with 2 threads: 1.01x
Parallel run with 4 threads → Pi = 3.1418173000, Time = 0.329096 sec
Speedup with 4 threads: 0.45x
Parallel run with 8 threads → Pi = 3.1416641500, Time = 0.834861 sec
Speedup with 8 threads: 0.18x
Parallel run with 16 threads → Pi = 3.1415660500, Time = 1.684253 sec
Speedup with 16 threads: 0.09x
[ec2-user@ip-172-31-33-146 lab02]$
```

```
[[ec2-user@ip-172-31-33-146 Exercise03]$ gcc -fopenmp exercise.c -o exercise.o -lm
[[ec2-user@ip-172-31-33-146 Exercise03]$ ./exercise.o
Serial Result: -333333333303516200960.00000
Serial Time: 0.035539 seconds

Threads:  1, Parallel Result: -333333333303516200960.00000, Time: 0.034806 seconds, Diff from Serial: 0.0000000000
Threads:  2, Parallel Result: -333333333303516332032.00000, Time: 0.017664 seconds, Diff from Serial: 131072.0000000000
Speedup with 2 threads: 2.01x

Threads:  4, Parallel Result: -333333333303516200960.00000, Time: 0.017584 seconds, Diff from Serial: 0.0000000000
Speedup with 4 threads: 2.02x

Threads:  8, Parallel Result: -333333333303515480064.00000, Time: 0.017768 seconds, Diff from Serial: 720896.0000000000
Speedup with 8 threads: 2.00x

Threads: 16, Parallel Result: -333333333303515807744.00000, Time: 0.018169 seconds, Diff from Serial: 393216.0000000000
Speedup with 16 threads: 1.96x

[ec2-user@ip-172-31-33-146 Exercise03]$
```

```c
#include <stdio.h>
#include <math.h>
#include <omp.h>


double calculate_sum_serial(long long N) {
double S = 0.0;
for (long long i = 1; i <= N; i++) {
S += i - (i * i) + sqrt(2.0 * (i + 1));
}
return S;
}

double calculate_sum_parallel(long long N, int num_threads) {
double S = 0.0; omp_set_num_threads(num_threads);
#pragma omp parallel for reduction(+:S)
for (long long i = 1; i <= N; i++) {
S += i - (i * i) + sqrt(2.0 * (i + 1));
}
return S;
}

int main() { long long N = 10000000; // 10 million int
thread_counts[] = {1, 2, 4, 8, 16}; int size =
sizeof(thread_counts) / sizeof(thread_counts[0]);
```

```
double start, end, time_serial, time_parallel; double result_serial,
result_parallel;

// Serial run
start = omp_get_wtime(); result_serial = calculate_sum_serial(N);
end = omp_get_wtime(); time_serial = end - start;
printf("Serial Result: %.5f\n", result_serial); printf("Serial Time: %f seconds\n\n",
time_serial);

// Parallel runs
for (int i = 0; i < size; i++) { int threads = thread_counts[i];
start = omp_get_wtime();
result_parallel = calculate_sum_parallel(N, threads); end = omp_get_wtime();
time_parallel = end - start;

// Verify correctness
double diff = fabs(result_serial - result_parallel);
printf("Threads: %2d, Parallel Result: %.5f, Time: %f seconds, Diff from Serial: %.10f\n", threads, result_parallel,
time_parallel, diff);

if (threads > 1) {
printf("Speedup with %d threads: %.2fx\n\n", threads, time_serial / time_parallel);
}
}

return 0;
}
```