National Institute of Technology Karnataka Surathkal,
Department of Information Technology,
IT 300 - Parallel Computing

Lab 5: 10/09/2018

1. Consider the code which numerically approximates pi:

```
#include<stdio.h>
int main()
{
double pi,x;
int i,N;
pi=0.0;
N=1000;
#pragma omp parallel for
for(i=0;i<=N;i++)
{
x=(double)i/N;
pi+=4/(1+x*x);
}
pi=pi/N;
printf("Pi is %f\n",pi);
}
```

Compile this with gcc main.c -o test, ignoring the -fopenmp options, this means that the **#pragma omp parallel for** will be interpreted as a comment i.e. ignored. We run it and this is the result:

```
prog@abc-system:~$ gcc test.c -o test
prog@abc-system:~$ ./test
Pi is 3.142592
```

Now compile with the -fopenmp option and run:

```
prog@abc-system:~$ gcc test.c -o test -fopenmp
prog@abc-system:~$ ./test
Pi is 2.785016
```

[Check for more than one run. You will get different values]

Let's examine what went wrong. Well, by default and as we have not specified it as private, the **variable x is shared**. This means all threads have the same memory address of the variable x.

Therefore, thread i will compute some value at x and store it at memory address &x, thread j will then compute its value of x and store it at &x **BEFORE** thread i has used its value to make its contribution to pi. The threads are all over writing each others values of x because they all have the same memory address for x. Our first correction is that x must be made private:
#pragma omp parallel for private(x)

Secondly, we have a **"Race Condition"** for pi. Let me illustrate this with a simple example. Here is what would ideally happen:

- Thread 1 reads the current value of pi : 0
- Thread 1 increments the value of pi : 1
- Thread 1 stores the new value of pi: 1
- Thread 2 reads the current value of pi: 1
- Thread 2 increments the value of pi: 2
- Thread 2 stores the value of pi: 2

What is actually happening is more like this:

- Thread 1 reads the current value of pi: 0
- Thread 2 reads the current value of pi: 0
- Thread 1 increments pi: 1
- Thread 2 increments pi: 1
- Thread 1 stores its value of pi: 1
- Thread 2 stores its value of pi: 1

The way to correct this is to tell the code to execute the read/write of pi only one thread at a time. This can be achieved with critical or atomic.

Add **#pragma omp atomic**

Just before pi get's updated and you'll see that it works.

This scenario crops up time and time again where you are updating some value inside a parallel loop so in the end it had its own clause made for it. All the above can be achieved by simply making pi a **reduction variable**.

Reduction

To make pi a reduction variable the code is changed as follows:

```
int main(void){
double pi,x;
int i,N;
pi=0.0;
N=1000;
#pragma omp parallel for private(x) reduction(+:pi)
for(i=0;i<=N;i++){
x=(double)i/N;
pi+=4/(1+x*x);
```

```
}
pi=pi/N;
printf("Pi is %f\n",pi);
}
```

This is simply the quick and neat way of achieving all what we did above.

Check the result using atomic and/ reduction .

2. **How to compare sequential and parallel program execution times. ?**

   Include following header files in the program.

```
#include <sys/time.h>
#include <stdlib.h>
```

   Declare following variables.

```
struct timeval  TimeValue_Start;
struct timezone TimeZone_Start;

struct timeval  TimeValue_Final;
struct timezone TimeZone_Final;
long   time_start, time_end;
double  time_overhead;
```

   Just before starting parallel region code , note down the time(start time)

```
gettimeofday(&TimeValue_Start, &TimeZone_Start);
```

   After finishing parallel region, get end time.

```
gettimeofday(&TimeValue_Final, &TimeZone_Final);
```

   Calculate the overhead time as follows:

```
time_start = TimeValue_Start.tv_sec * 1000000 + TimeValue_Start.tv_usec;
time_end = TimeValue_Final.tv_sec * 1000000 + TimeValue_Final.tv_usec;
        time_overhead = (time_end - time_start)/1000000.0;
printf("\n\n\t\t Time in Seconds (T)         : %lf",time_overhead);
```

   Example:

```
#include <stdio.h>
#include <sys/time.h>
#include <omp.h>
#include <stdlib.h>

int main(void){
struct timeval  TimeValue_Start;
struct timezone TimeZone_Start;
struct timeval  TimeValue_Final;
struct timezone TimeZone_Final;
long   time_start, time_end;
double  time_overhead;
```

```
double pi,x;
int i,N;
pi=0.0;
N=1000;
gettimeofday(&TimeValue_Start, &TimeZone_Start);
#pragma omp parallel for private(x) reduction(+:pi)
for(i=0;i<=N;i++){
x=(double)i/N;
pi+=4/(1+x*x);
}
gettimeofday(&TimeValue_Final, &TimeZone_Final);
time_start = TimeValue_Start.tv_sec * 1000000 + TimeValue_Start.tv_usec;
time_end = TimeValue_Final.tv_sec * 1000000 + TimeValue_Final.tv_usec;
time_overhead = (time_end - time_start)/1000000.0;
printf("\n\n\tTime in Seconds (T) : %lf\n",time_overhead);
pi=pi/N;
printf("\n \tPi is %f\n\n",pi);
}
```

---

3. Write a sequential program to find the smallest element in an array. Convert the same program for parallel execution. Initialise array with random numbers. Consider an array size as 100, 500 and 1000. Analyse the result for maximum number of threads and various schedule() function. Based on observation, perform analysis of the total execution time and explain the result by plotting the graph.

Schedule()	Total execution time for Number of iterations =100	Total execution time for Number of iterations =500	Total execution time for Number of iterations =1000
Sequential execution			
static			
Static, chunksize			
dynamic			
guided			
runtime			