

NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA SURATHKAL

DEPARTMENT OF INFORMATION TECHNOLOGY

**IT 301 Parallel Computing LAB 3 (Ritik Pansuriya - 181IT237)**

26th August 2020

Faculty: Dr. Geetha V and Mrs. Tanmayee

---

**1. Program 1 - copyin clause provides a mechanism to copy the value of a threadprivate variable of the master thread to the threadprivate variable of each other member of the team that is executing the parallel region. Barrier is used to take all the thread simultaneously at one position.**

**Execute following code and observe the working of threadprivate directive and copyin clause:**

```
#include<stdio.h>

#include<omp.h>

int tid,x;

#pragma omp threadprivate(x,tid)

void main()

{
x=10;

#pragma omp parallel num_threads(4) copyin(x)

{
tid=omp_get_thread_num();

#pragma omp master

{
printf("Parallel Region 1 \n");

x=x+1;

}
```

```

#pragma omp barrier

if(tid==1)

x=x+2;

printf("Thread % d Value of x is %d\n",tid,x);

}//#pragma omp barrier

#pragma omp parallel num_threads(4)

{

#pragma omp master

{

printf("Parallel Region 2 \n");

}

#pragma omp barrier

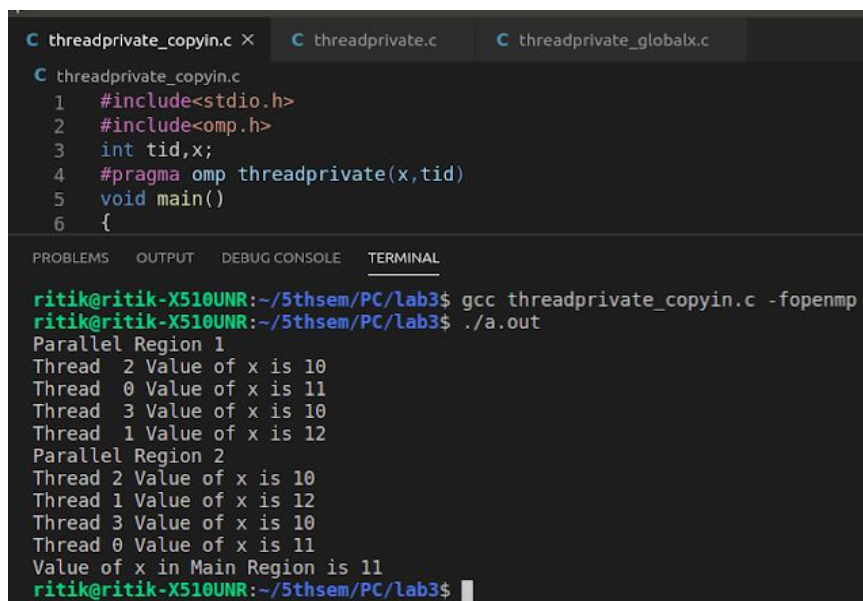
printf("Thread %d Value of x is %d\n",tid,x);

}

printf("Value of x in Main Region is %d\n",x);

}

```



The screenshot shows a code editor with three tabs: 'threadprivate\_copyin.c', 'threadprivate.c', and 'threadprivate\_globalx.c'. The active tab is 'threadprivate\_copyin.c', which contains the following code:

```

1  #include<stdio.h>
2  #include<omp.h>
3  int tid,x;
4  #pragma omp threadprivate(x,tid)
5  void main()
6  {

```

Below the code editor is a terminal window with the following output:

```

ritik@ritik-X510UNR:~/5thsem/PC/Lab3$ gcc threadprivate_copyin.c -fopenmp
ritik@ritik-X510UNR:~/5thsem/PC/Lab3$ ./a.out
Parallel Region 1
Thread 2 Value of x is 10
Thread 0 Value of x is 11
Thread 3 Value of x is 10
Thread 1 Value of x is 12
Parallel Region 2
Thread 2 Value of x is 10
Thread 1 Value of x is 12
Thread 3 Value of x is 10
Thread 0 Value of x is 11
Value of x in Main Region is 11
ritik@ritik-X510UNR:~/5thsem/PC/Lab3$

```

As `threadprivate()` assigns `x=10` to the master thread, `copyin(x)` copies into all other threads the value assigned in the master thread. Barrier clause indicates that all thread must wait at that position for all thread at that place. Main region is changed by the master region.

**DO the following: 1. Remove the `copyin` clause and check the output.**

```
C threadprivate_copyin.c
1  #include<stdio.h>
2  #include<omp.h>
3  int tid,x;
4  #pragma omp threadprivate(x,tid)
5  void main()
6  {

ritik@ritik-X510UNR:~/5thsem/PC/lab3$ gcc threadprivate.c -fopenmp
ritik@ritik-X510UNR:~/5thsem/PC/lab3$ ./a.out
Parallel Region 1
Thread 1 Value of x is 2
Thread 0 Value of x is 11
Thread 3 Value of x is 0
Thread 2 Value of x is 0
Parallel Region 2
Thread 3 Value of x is 0
Thread 2 Value of x is 0
Thread 1 Value of x is 2
Thread 0 Value of x is 11
Value of x in Main Region is 11
ritik@ritik-X510UNR:~/5thsem/PC/lab3$
```

As `threadprivate()` assigns `x=10` to the master thread, as no `copyin` so thread is initialized to 0. Barrier clause indicates that all thread must wait at that position for all thread at that place. Main region is changed by the master region.

**2. Remove the `copyin` clause and initialize `x` globally.**

As `x` is assigned globally, all threads will copy the same value.

```
C threadprivate_copyin.c X C threadprivate.c C threadprivate_globalx.c
C threadprivate_copyin.c
1  #include<stdio.h>
2  #include<omp.h>
3  int tid,x;
4  #pragma omp threadprivate(x,tid)
5  void main()
6  {

ritik@ritik-X510UNR:~/5thsem/PC/lab3$ gcc threadprivate_globalx.c -fopenmp
ritik@ritik-X510UNR:~/5thsem/PC/lab3$ ./a.out
Parallel Region 1
Thread 1 Value of x is 12
Thread 3 Value of x is 10
Thread 2 Value of x is 10
Thread 0 Value of x is 11
Parallel Region 2
Thread 1 Value of x is 12
Thread 2 Value of x is 10
Thread 3 Value of x is 10
Thread 0 Value of x is 11
Value of x in Main Region is 11
ritik@ritik-X510UNR:~/5thsem/PC/lab3$
```

## 2. Program 2 - Learn the concept of firstprivate() and threadprivate()

```
#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

int count=0;

#pragma omp threadprivate(count)

int main (void) {

int x=10, y=20, a[10], b[10], c[10], i;

//int count=0;

for(i=0; i<10; i++)

b[i]=c[i]=i;

printf("1. count=%d\n", count);

#pragma omp parallel num_threads(2) copyin(count)

{

#pragma omp for schedule(static,5) firstprivate(x)

for(i=0; i<10; i++)

{

int tid1=omp_get_thread_num();

a[i]=b[i]+c[i];

count++;

x++;

printf("tid=%d, a[%d]=%d, count=%d x=%d\n", tid1, i, a[i], count, x);

}

#pragma omp barrier

printf("2. before copyprivate count=%d x=%d tid=%d\n", count, x, omp_get_thread_num());
```

```

#pragma omp single copyprivate(count)
{
count=count+20;
}

printf("3. after copyprivate count=%d x=%d tid=%d\n",count,x,omp_get_thread_num());

#pragma omp for schedule(static,5) firstprivate(x)
for(i=0;i<10;i++)
{
int tid1=omp_get_thread_num();
a[i]=b[i]*c[i];
count++;
x++;

printf("tid=%d,a[%d]=%d, count=%d, x=%d\n",tid1,i,a[i],count,x);
}
}

#pragma omp barrier

printf("4. count=%d x=%d\n",count,x);

printf("\n");

return 0;
}

```

---As count is initialized globally and threadprivate(count) count is global to all the threads. Everytime firstprivate(x) is encountered x=10 is initialized in each thread for that region alone. However as count is global to the thread its value is retained whereas x's value isn't not global so it's value is not retained. Single copyprivate(count) only one time count value is added by 20.

```
C threadprivate_copyin.c C first_threadprivate.c X C threadprivate.c C threadprivate_glo
C first_threadprivate.c
37 #pragma omp barrier
38 printf("4. count=%d x=%d\n",count,x);
39 return 0;
40 }
41

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
ritik@ritik-X510UNR:~/5thsem/PC/lab3$ gcc first_threadprivate.c -fopenmp
ritik@ritik-X510UNR:~/5thsem/PC/lab3$ ./a.out
1. count=0
tid=0,a[0]=0, count=1 x=11
tid=0,a[1]=2, count=2 x=12
tid=0,a[2]=4, count=3 x=13
tid=0,a[3]=6, count=4 x=14
tid=0,a[4]=8, count=5 x=15
tid=1,a[5]=10, count=1 x=11
tid=1,a[6]=12, count=2 x=12
tid=1,a[7]=14, count=3 x=13
tid=1,a[8]=16, count=4 x=14
tid=1,a[9]=18, count=5 x=15
2. before copyprivate count=5 x=10 tid=0
2. before copyprivate count=5 x=10 tid=1
3. after copyprivate count=25 x=10 tid=0
tid=0,a[0]=0, count=26, x=11
tid=0,a[1]=1, count=27, x=12
tid=0,a[2]=4, count=28, x=13
tid=0,a[3]=9, count=29, x=14
tid=0,a[4]=16, count=30, x=15
3. after copyprivate count=25 x=10 tid=1
tid=1,a[5]=25, count=26, x=11
tid=1,a[6]=36, count=27, x=12
tid=1,a[7]=49, count=28, x=13
tid=1,a[8]=64, count=29, x=14
tid=1,a[9]=81, count=30, x=15
4. count=30 x=10
ritik@ritik-X510UNR:~/5thsem/PC/lab3$
```

### 3. Program to understand the concept of collapse()

```
#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

int main (void) {

int i,j;

#pragma omp parallel

{

#pragma omp for schedule(static,3) private(i,j)

for(i=0;i<6;i++)

for(j=0;j<5;j++)

{

int tid2=omp_get_thread_num();

printf("tid=%d, i=%d j=%d\n",omp_get_thread_num(),i,j);
```

```

}

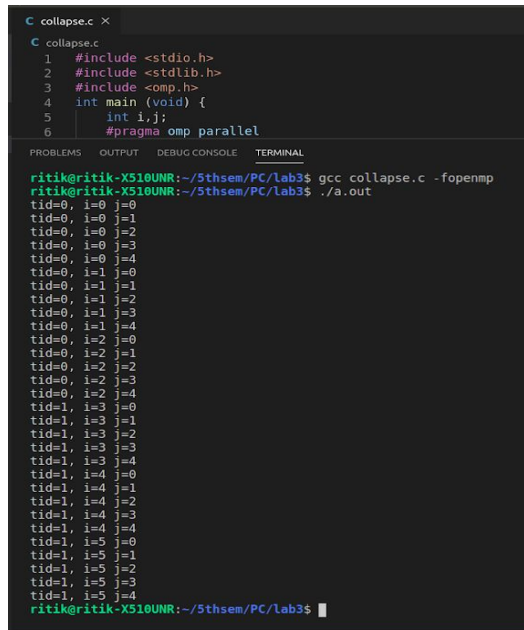
}

return 0;

}

```

Consider three for loops and check the result with no collapse():



The screenshot shows a C code editor with a file named `collapse.c`. The code is as follows:

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <omp.h>
4 int main (void) {
5     int i,j;
6     #pragma omp parallel

```

Below the code editor, the terminal output is displayed. It shows the compilation and execution of the program using OpenMP. The output consists of 20 lines, each representing an iteration of the three nested loops, showing the thread ID (tid) and the values of i and j. The iterations are ordered sequentially by thread, demonstrating that without the `collapse()` directive, the normal order of iteration is followed.

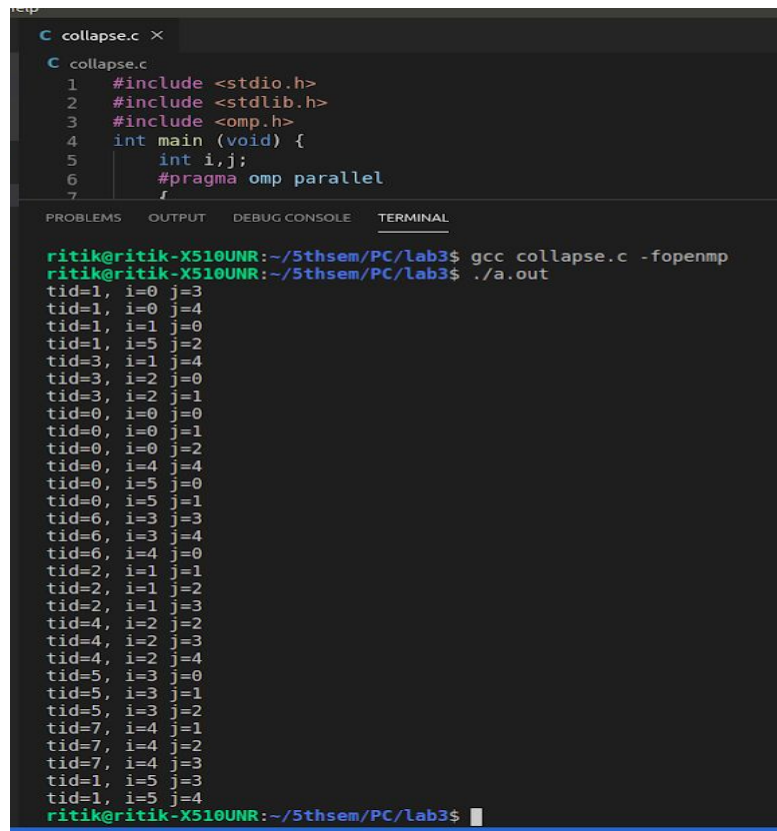
```

ritik@ritik-X510UNR:~/5thsem/PC/Lab3$ gcc collapse.c -fopenmp
ritik@ritik-X510UNR:~/5thsem/PC/Lab3$ ./a.out
tid=0, i=0 j=0
tid=0, i=0 j=1
tid=0, i=0 j=2
tid=0, i=0 j=3
tid=0, i=0 j=4
tid=0, i=1 j=0
tid=0, i=1 j=1
tid=0, i=1 j=2
tid=0, i=1 j=3
tid=0, i=1 j=4
tid=0, i=2 j=0
tid=0, i=2 j=1
tid=0, i=2 j=2
tid=0, i=2 j=3
tid=0, i=2 j=4
tid=1, i=3 j=0
tid=1, i=3 j=1
tid=1, i=3 j=2
tid=1, i=3 j=3
tid=1, i=3 j=4
tid=1, i=4 j=0
tid=1, i=4 j=1
tid=1, i=4 j=2
tid=1, i=4 j=3
tid=1, i=4 j=4
tid=1, i=5 j=0
tid=1, i=5 j=1
tid=1, i=5 j=2
tid=1, i=5 j=3
tid=1, i=5 j=4
ritik@ritik-X510UNR:~/5thsem/PC/Lab3$

```

No collapse so normal order of the iteration is followed by i,j.

## collapse(2)



```
C collapse.c x
C collapse.c
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <omp.h>
4  int main (void) {
5      int i,j;
6      #pragma omp parallel
7      {
          for (i=0; i<10; i++)
              for (j=0; j<10; j++)
                  printf("tid=%d, i=%d j=%d\n", omp_get_thread_num(), i, j);
      }
  }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

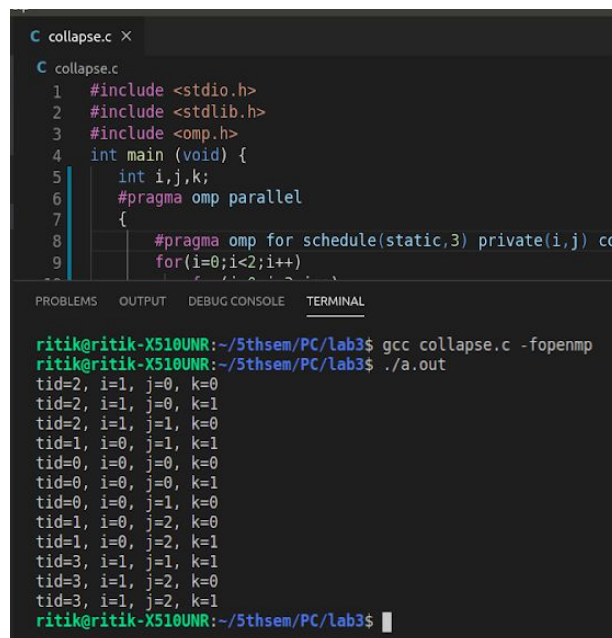
```
ritik@ritik-X510UNR:~/5thsem/PC/lab3$ gcc collapse.c -fopenmp
ritik@ritik-X510UNR:~/5thsem/PC/lab3$ ./a.out
tid=1, i=0 j=3
tid=1, i=0 j=4
tid=1, i=1 j=0
tid=1, i=5 j=2
tid=3, i=1 j=4
tid=3, i=2 j=0
tid=3, i=2 j=1
tid=0, i=0 j=0
tid=0, i=0 j=1
tid=0, i=0 j=2
tid=0, i=4 j=4
tid=0, i=5 j=0
tid=0, i=5 j=1
tid=6, i=3 j=3
tid=6, i=3 j=4
tid=6, i=4 j=0
tid=2, i=1 j=1
tid=2, i=1 j=2
tid=2, i=1 j=3
tid=4, i=2 j=2
tid=4, i=2 j=3
tid=4, i=2 j=4
tid=5, i=3 j=0
tid=5, i=3 j=1
tid=5, i=3 j=2
tid=7, i=4 j=1
tid=7, i=4 j=2
tid=7, i=4 j=3
tid=1, i=5 j=3
tid=1, i=5 j=4
ritik@ritik-X510UNR:~/5thsem/PC/lab3$
```

As both the for loop are not interconnected the order of execution can be anything.



**collapse(3).**

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
    int i,j,k;
    #pragma omp parallel
    {
        #pragma omp for schedule(static,3) private(i,j) collapse(3)
        for(i=0;i<2;i++)
            for(j=0;j<3;j++)
            {
                for(k=0; k<2; k++){
                    int tid2=omp_get_thread_num();
                    printf("tid=%d, i=%d, j=%d, k=%d\n",omp_get_thread_num(),i,j,k);
                }
            }
    }
    return 0;
}
```



The screenshot shows a code editor with a file named `collapse.c`. The code is identical to the one in the previous block. Below the editor, the terminal output shows the execution of the program. The output consists of 12 lines of thread IDs and loop indices, demonstrating a random execution order due to the `collapse(3)` directive.

```
C collapse.c x
C collapse.c
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <omp.h>
4  int main (void) {
5      int i,j,k;
6      #pragma omp parallel
7      {
8          #pragma omp for schedule(static,3) private(i,j) co
9          for(i=0;i<2;i++)
10             for(j=0;j<3;j++)
11             {
12                 for(k=0; k<2; k++){
13                     int tid2=omp_get_thread_num();
14                     printf("tid=%d, i=%d, j=%d, k=%d\n",omp_get_thread_num(),i,j,k);
15                 }
16             }
17      }
18      return 0;
19  }

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
ritik@ritik-X510UNR:~/5thsem/PC/lab3$ gcc collapse.c -fopenmp
ritik@ritik-X510UNR:~/5thsem/PC/lab3$ ./a.out
tid=2, i=1, j=0, k=0
tid=2, i=1, j=0, k=1
tid=2, i=1, j=1, k=0
tid=1, i=0, j=1, k=1
tid=0, i=0, j=0, k=0
tid=0, i=0, j=0, k=1
tid=0, i=0, j=1, k=0
tid=1, i=0, j=2, k=0
tid=1, i=0, j=2, k=1
tid=3, i=1, j=1, k=1
tid=3, i=1, j=2, k=0
tid=3, i=1, j=2, k=1
ritik@ritik-X510UNR:~/5thsem/PC/lab3$
```

Same as collapse 2 instead we have i, j, k in random fashion.

#### **4. 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;

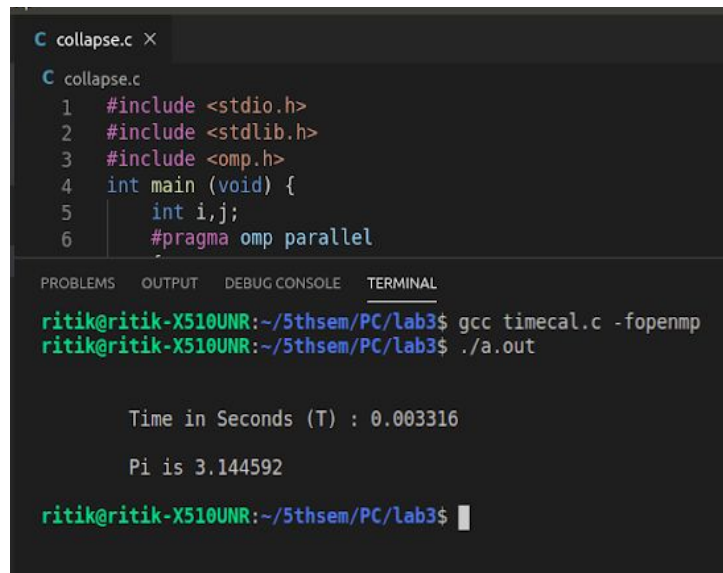
printf("\n\n\tTime in Seconds (T) : %lf\n",time_overhead);

```

```
pi=pi/N;
```

```
printf("\n \tPi is %f\n\n",pi);
```

```
}
```



The screenshot shows a code editor with a file named `collapse.c`. The code is as follows:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <omp.h>
4 int main (void) {
5     int i,j;
6     #pragma omp parallel
```

Below the code editor is a terminal window with the following output:

```
ritik@ritik-X510UNR:~/5thsem/PC/lab3$ gcc timecal.c -fopenmp
ritik@ritik-X510UNR:~/5thsem/PC/lab3$ ./a.out

Time in Seconds (T) : 0.003316

Pi is 3.144592

ritik@ritik-X510UNR:~/5thsem/PC/lab3$
```

**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 10k, 50k and 100k. Analyse the result for maximum number of threads and various schedule() functions. Based on observation, perform analysis of the total execution time and explain the result by plotting the graph. [increase array size until parallel execution time is less than sequential execution.]**

```
#include <stdio.h>

#include <sys/time.h>

#include <omp.h>

#include <stdlib.h>

int main(){

    struct timeval TimeValue_Start;

    struct timezone TimeZone_Start;

    struct timeval TimeValue_Final;

    struct timezone TimeZone_Final;

    long time_start1, time_end1;

    double time_overhead1;

    int n=100000;

        int array[n],i;

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

        array[i]=rand();

    }

    long time_start2, time_end2;

    double time_overhead2;

        int cur_min2=array[0];

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

```

#pragma omp parallel for schedule(guided,8) num_threads(8)

for(i=0; i<n; i++){

    if(array[i]<cur_min2)

        #pragma omp critical

        if(array[i]<cur_min2)

            cur_min2=array[i];

}

gettimeofday(&TimeValue_Final, &TimeZone_Final);

time_start2 = TimeValue_Start.tv_sec * 1000000 + TimeValue_Start.tv_usec;

time_end2 = TimeValue_Final.tv_sec * 1000000 + TimeValue_Final.tv_usec;

time_overhead2 = (time_end2 - time_start2)/1000000.0;

printf("\n\n\tTime in Parallal Seconds (T) : %lf\n",time_overhead2);

    printf("The smallest number in given array is %d\n",cur_min2);

    int cur_min1=array[0];

gettimeofday(&TimeValue_Start, &TimeZone_Start);

for(int i=1; i<n; i++){

    if(cur_min1 > array[i]) cur_min1=array[i];

}

gettimeofday(&TimeValue_Final, &TimeZone_Final);

time_start1 = TimeValue_Start.tv_sec * 1000000 + TimeValue_Start.tv_usec;

time_end1 = TimeValue_Final.tv_sec * 1000000 + TimeValue_Final.tv_usec;

time_overhead1 = (time_end1 - time_start1)/1000000.0;

printf("\n\n\tTime in Sequential Seconds (T) : %lf\n",time_overhead1);

    printf("The smallest number in given array is %d\n",cur_min1);

```

```

return 0;
}

```

Schedule()	Total Execution time for number of iterations 5K	Total execution for number of iterations 10K	Total execution for number of iterations 50K	Total execution for number of iterations 100K	Total execution for number of iterations 1000K
Sequential execution	0.000013	0.000029	0.000141	0.00028	0.002478
Static	0.001694	0.005555	0.001018	0.003020	0.001505
Static, chunksize	0.000928	0.015404	0.015404	0.005376	0.002266
Dynamic, chunksize	0.004858	0.007924	0.003158	0.003586	0.001580
Guided	0.001997	0.000845	0.015811	0.001770	0.002008
runtime	0.004074	0.002	0.003956	0.004378	0.029139

Sequential execution gave better runtimes for lower iterations due to the fact that for the threading process the time taken to thread the process is also added along with the execution time. However when the iterations are very large then parallel code has lesser runtime. Also increasing the number of threads reduces the runtime but upto some extent(20 thread gives lowest runtime for me). Value of N=1000000 leads that sequential has more execution time then parallel execution.

