DEPARTMENT OF INFORMATION TECHNOLOGY, NITK SURATHKAL

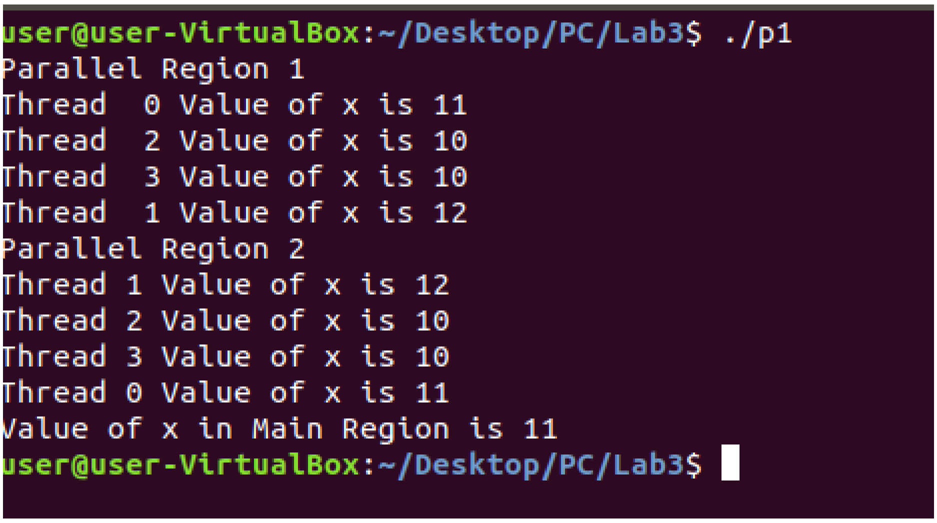
IT301 Parallel Programming

LAB 3 (26th August 2020)

Submitted by: Harsh Agarwal (181IT117)

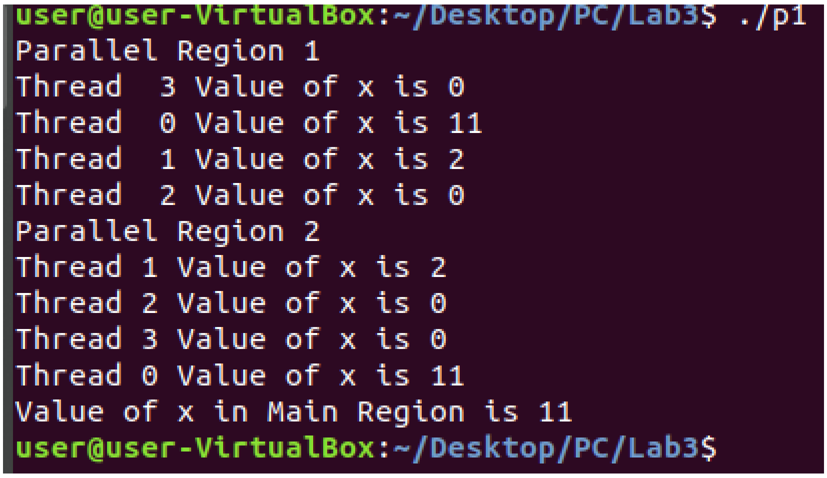
1. **Working og Threadprivate and copyin clause**

a) Threadprivate directive and copyin clause:



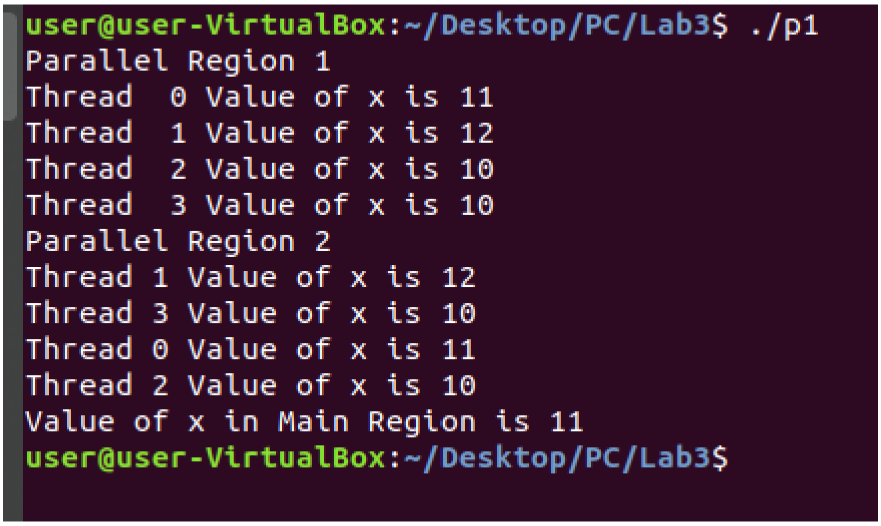
```
user@user-VirtualBox:~/Desktop/PC/Lab3$ ./p1
Parallel Region 1
Thread  0 Value of x is 11
Thread  2 Value of x is 10
Thread  3 Value of x is 10
Thread  1 Value of x is 12
Parallel Region 2
Thread 1 Value of x is 12
Thread 2 Value of x is 10
Thread 3 Value of x is 10
Thread 0 Value of x is 11
Value of x in Main Region is 11
user@user-VirtualBox:~/Desktop/PC/Lab3$ 
```

b) Remove copyin clause and check the output:



```
user@user-VirtualBox:~/Desktop/PC/Lab3$ ./p1
Parallel Region 1
Thread  3 Value of x is 0
Thread  0 Value of x is 11
Thread  1 Value of x is 2
Thread  2 Value of x is 0
Parallel Region 2
Thread 1 Value of x is 2
Thread 2 Value of x is 0
Thread 3 Value of x is 0
Thread 0 Value of x is 11
Value of x in Main Region is 11
user@user-VirtualBox:~/Desktop/PC/Lab3$
```

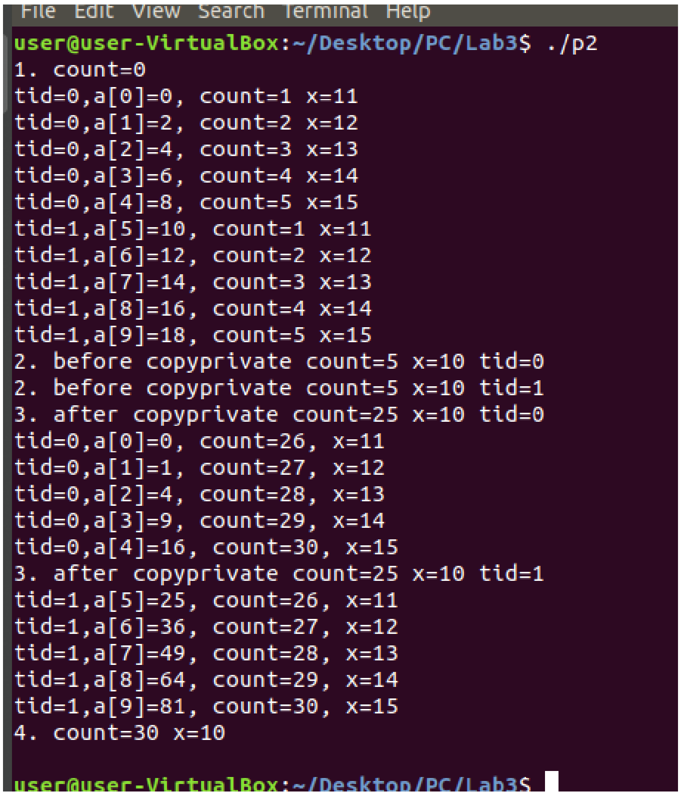
c) Remove copyin clause and initialize x globally:



```
user@user-VirtualBox:~/Desktop/PC/Lab3$ ./p1
Parallel Region 1
Thread  0 Value of x is 11
Thread  1 Value of x is 12
Thread  2 Value of x is 10
Thread  3 Value of x is 10
Parallel Region 2
Thread 1 Value of x is 12
Thread 3 Value of x is 10
Thread 0 Value of x is 11
Thread 2 Value of x is 10
Value of x in Main Region is 11
user@user-VirtualBox:~/Desktop/PC/Lab3$
```

Analysis:
The copyin clause shares the variable value (Only the value by coping) to the other threads, the same functionality can be achieved by making the variable global. The 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.

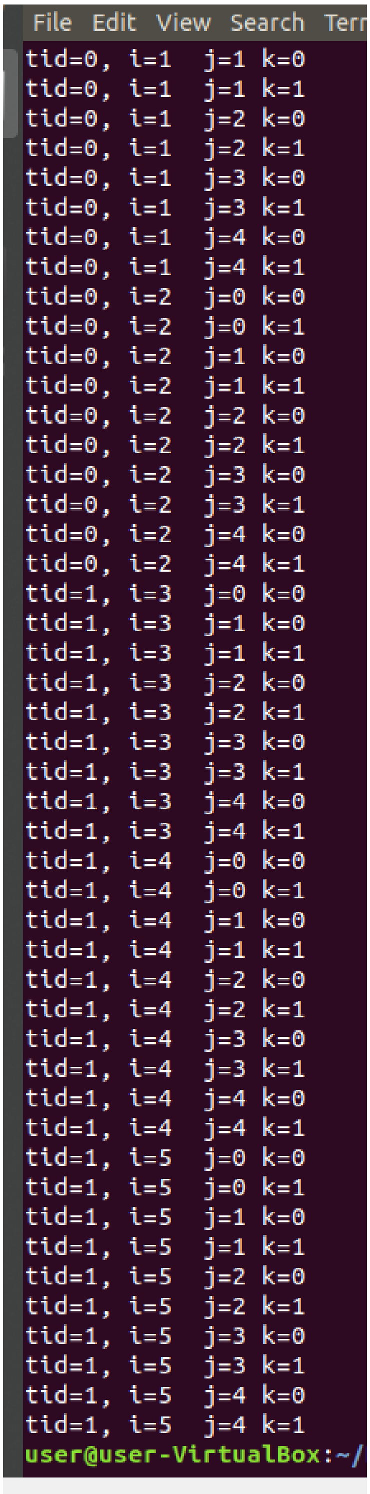
2. Learn the concept of firstprivate() and threadprivate():



```
user@user-VirtualBox:~/Desktop/PC/Lab3$ ./p2
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

user@user-VirtualBox:~/Desktop/PC/Lab3$
```

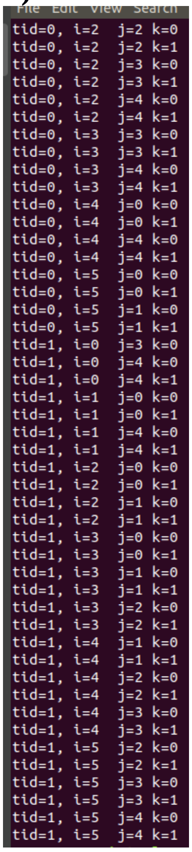
Analysis:
The threadprivate directive specifies that variables are replicated, with each thread having its own copy. The firstprivate directive specifies that each thread should have its own instance of a variable, and that the variable should be initialized with the value of the variable, because it exists before the parallel construct.

3. **Program to understand the concept of collapse():**
   a) **Without collapse( )**



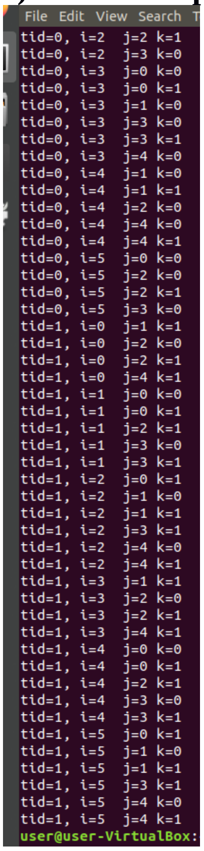
```
tid=0, i=1  j=1 k=0
tid=0, i=1  j=1 k=1
tid=0, i=1  j=2 k=0
tid=0, i=1  j=2 k=1
tid=0, i=1  j=3 k=0
tid=0, i=1  j=3 k=1
tid=0, i=1  j=4 k=0
tid=0, i=1  j=4 k=1
tid=0, i=2  j=0 k=0
tid=0, i=2  j=0 k=1
tid=0, i=2  j=1 k=0
tid=0, i=2  j=1 k=1
tid=0, i=2  j=2 k=0
tid=0, i=2  j=2 k=1
tid=0, i=2  j=3 k=0
tid=0, i=2  j=3 k=1
tid=0, i=2  j=4 k=0
tid=0, i=2  j=4 k=1
tid=1, i=3  j=0 k=0
tid=1, i=3  j=1 k=0
tid=1, i=3  j=1 k=1
tid=1, i=3  j=2 k=0
tid=1, i=3  j=2 k=1
tid=1, i=3  j=3 k=0
tid=1, i=3  j=3 k=1
tid=1, i=3  j=4 k=0
tid=1, i=3  j=4 k=1
tid=1, i=4  j=0 k=0
tid=1, i=4  j=0 k=1
tid=1, i=4  j=1 k=0
tid=1, i=4  j=1 k=1
tid=1, i=4  j=2 k=0
tid=1, i=4  j=2 k=1
tid=1, i=4  j=3 k=0
tid=1, i=4  j=3 k=1
tid=1, i=4  j=4 k=0
tid=1, i=4  j=4 k=1
tid=1, i=5  j=0 k=0
tid=1, i=5  j=0 k=1
tid=1, i=5  j=1 k=0
tid=1, i=5  j=1 k=1
tid=1, i=5  j=2 k=0
tid=1, i=5  j=2 k=1
tid=1, i=5  j=3 k=0
tid=1, i=5  j=3 k=1
tid=1, i=5  j=4 k=0
tid=1, i=5  j=4 k=1
user@user-VirtualBox:~/
```

b) With collapse(2):



```
tid=0, i=2  j=2 k=0
tid=0, i=2  j=2 k=1
tid=0, i=2  j=3 k=0
tid=0, i=2  j=3 k=1
tid=0, i=2  j=4 k=0
tid=0, i=2  j=4 k=1
tid=0, i=3  j=3 k=0
tid=0, i=3  j=3 k=1
tid=0, i=3  j=4 k=0
tid=0, i=3  j=4 k=1
tid=0, i=4  j=0 k=0
tid=0, i=4  j=0 k=1
tid=0, i=4  j=4 k=0
tid=0, i=4  j=4 k=1
tid=0, i=5  j=0 k=0
tid=0, i=5  j=0 k=1
tid=0, i=5  j=1 k=0
tid=0, i=5  j=1 k=1
tid=1, i=0  j=3 k=0
tid=1, i=0  j=4 k=0
tid=1, i=0  j=4 k=1
tid=1, i=1  j=0 k=0
tid=1, i=1  j=0 k=1
tid=1, i=1  j=4 k=0
tid=1, i=1  j=4 k=1
tid=1, i=2  j=0 k=0
tid=1, i=2  j=0 k=1
tid=1, i=2  j=1 k=0
tid=1, i=2  j=1 k=1
tid=1, i=3  j=0 k=0
tid=1, i=3  j=0 k=1
tid=1, i=3  j=1 k=0
tid=1, i=3  j=1 k=1
tid=1, i=3  j=2 k=0
tid=1, i=3  j=2 k=1
tid=1, i=4  j=1 k=0
tid=1, i=4  j=1 k=1
tid=1, i=4  j=2 k=0
tid=1, i=4  j=2 k=1
tid=1, i=4  j=3 k=0
tid=1, i=4  j=3 k=1
tid=1, i=5  j=2 k=0
tid=1, i=5  j=2 k=1
tid=1, i=5  j=3 k=0
tid=1, i=5  j=3 k=1
tid=1, i=5  j=4 k=0
tid=1, i=5  j=4 k=1
```

c) With collapse(3):



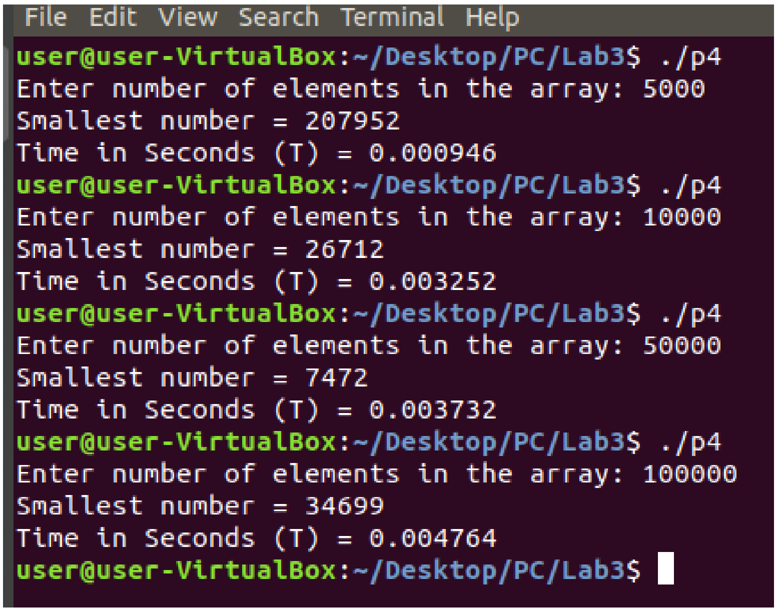
```
tid=0, i=2  j=2 k=1
tid=0, i=2  j=3 k=0
tid=0, i=3  j=0 k=0
tid=0, i=3  j=0 k=1
tid=0, i=3  j=1 k=0
tid=0, i=3  j=3 k=0
tid=0, i=3  j=3 k=1
tid=0, i=3  j=4 k=0
tid=0, i=4  j=1 k=0
tid=0, i=4  j=1 k=1
tid=0, i=4  j=2 k=0
tid=0, i=4  j=4 k=0
tid=0, i=4  j=4 k=1
tid=0, i=5  j=0 k=0
tid=0, i=5  j=2 k=0
tid=0, i=5  j=2 k=1
tid=0, i=5  j=3 k=0
tid=1, i=0  j=1 k=1
tid=1, i=0  j=2 k=0
tid=1, i=0  j=2 k=1
tid=1, i=0  j=4 k=1
tid=1, i=1  j=0 k=0
tid=1, i=1  j=0 k=1
tid=1, i=1  j=2 k=1
tid=1, i=1  j=3 k=0
tid=1, i=1  j=3 k=1
tid=1, i=2  j=0 k=1
tid=1, i=2  j=1 k=0
tid=1, i=2  j=1 k=1
tid=1, i=2  j=3 k=1
tid=1, i=2  j=4 k=0
tid=1, i=2  j=4 k=1
tid=1, i=3  j=1 k=1
tid=1, i=3  j=2 k=0
tid=1, i=3  j=2 k=1
tid=1, i=3  j=4 k=1
tid=1, i=4  j=0 k=0
tid=1, i=4  j=0 k=1
tid=1, i=4  j=2 k=1
tid=1, i=4  j=3 k=0
tid=1, i=4  j=3 k=1
tid=1, i=5  j=0 k=1
tid=1, i=5  j=1 k=0
tid=1, i=5  j=1 k=1
tid=1, i=5  j=3 k=1
tid=1, i=5  j=4 k=0
tid=1, i=5  j=4 k=1
user@user-VirtualBox:
```

Code:

```
#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<6;i++)
     for(j=0;j<5;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;
}
```

Analysis:
When there is no collapse then the threads share only the iterations of i but not j and k. But with collapse(2) first 2 loop iterations will be shared among the threads and the 3rd loop will not be shared and when with collapse(3) all the iterations of 3 loops will be shared among thethreads.

4. Analysis of program to find the minimum element in an array:



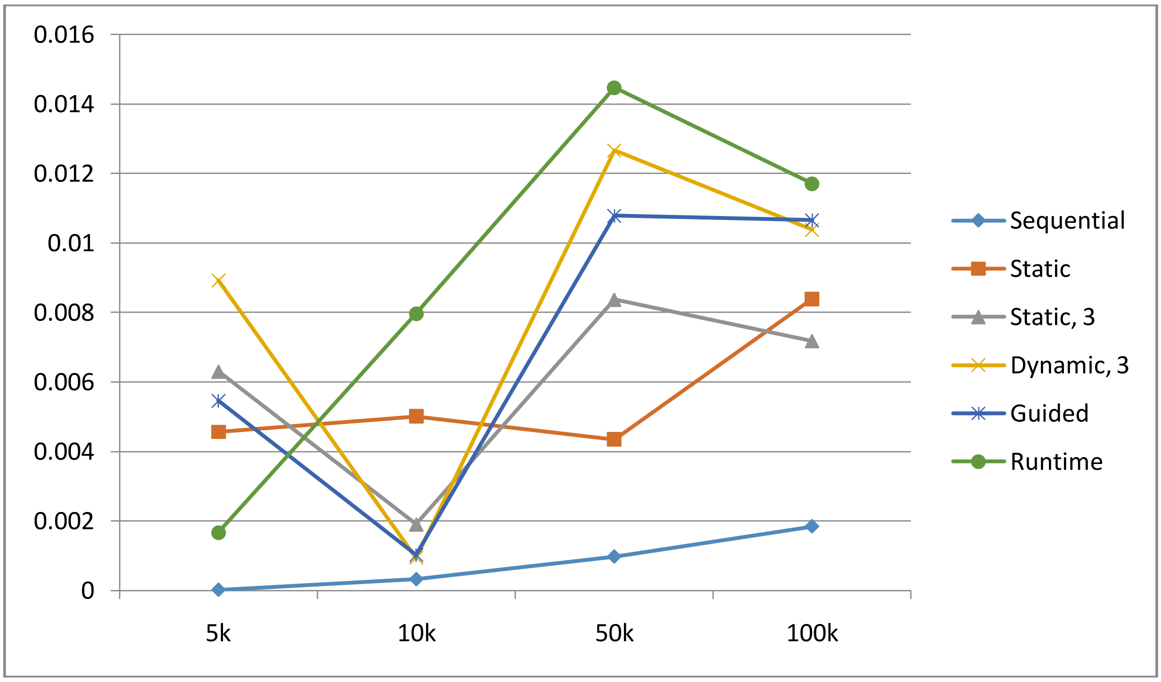
```
user@user-VirtualBox:~/Desktop/PC/Lab3$ ./p4
Enter number of elements in the array: 5000
Smallest number = 207952
Time in Seconds (T) = 0.000946
user@user-VirtualBox:~/Desktop/PC/Lab3$ ./p4
Enter number of elements in the array: 10000
Smallest number = 26712
Time in Seconds (T) = 0.003252
user@user-VirtualBox:~/Desktop/PC/Lab3$ ./p4
Enter number of elements in the array: 50000
Smallest number = 7472
Time in Seconds (T) = 0.003732
user@user-VirtualBox:~/Desktop/PC/Lab3$ ./p4
Enter number of elements in the array: 100000
Smallest number = 34699
Time in Seconds (T) = 0.004764
user@user-VirtualBox:~/Desktop/PC/Lab3$ 
```

Code:

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

int main()
{
    srand(time(0));

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


    int n, cur_min = 2147483647;
    printf("Enter number of elements in the array: ");
    scanf("%d", &n);

    int array[100000] = {2147483647};
   for (int a = 0; a < n; a++)
   {
      array[a] = rand();
   }

    gettimeofday(&TimeValue_Start, &TimeZone_Start);

#pragma omp parallel for
    for (int i = 0; i < n; i = i + 1)
    {
                    #pragma omp critical
                    if (array[i] < cur_min)
                            cur_min = array[i];
    }

    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("Smallest number = %d\n", cur_min);
    printf("Time in Seconds (T) = %lf\n",time_overhead);
}
```

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
Sequential	0.000146	0.000319	0.000970	0.001839
static	0.004558	0.005010	0.004342	0.008392
Static, 3	0.006306	0.001906	0.008375	0.007186
Dynamic, 3	0.008925	0.000934	0.012665	0.010378
Guided	0.005461	0.001024	0.010787	0.010653
runtime	0.001663	0.007972	0.014477	0.011716

Graph:



Analysis:

Here we observe that sequential execution takes lesser time than parallel execution as the computations here are smaller. But for larger programs with tougher computations parallelexecution will speed up. For the array size which is like 10^9 and all parallel execution will take less time than sequential execution.