Assignment 2
2018201008
Tarun Mohandas
10 March 2019

Q 1.1

The static code runs fastest among the three.

```
Running 100000000 iterations on 10 threads dynamically.
All done!

real    0m0.244s
user    0m0.239s
sys     0m0.002s
```

dynamic

```
Running 100000000 iterations on 10 threads guided.
238.784000 milliseconds
All done!
```

static

```
Running 100000000 iterations on 10 threads statically.
240.877000 milli seconds
All done!
```

guide

2nd part

1_4_static

```
real    0m54.010s
user    0m0.002s
sys     0m0.003s
```

1_4_dynamic

```
real    0m36.018s
user    0m0.003s
sys     0m0.004s
```

In this case, each iteration of the loop takes different times. Therefore, static scheduling is will not work well. Dynamic scheduling will do well and distribute work across threads as per load.

Q2
The code has to be changed to be able to work as expected.

#pragma omp critical
and
#pragma omp parallel **shared(num_trials) private(x,y)**
and
#pragma omp for **reduction(+:Ncirc)**

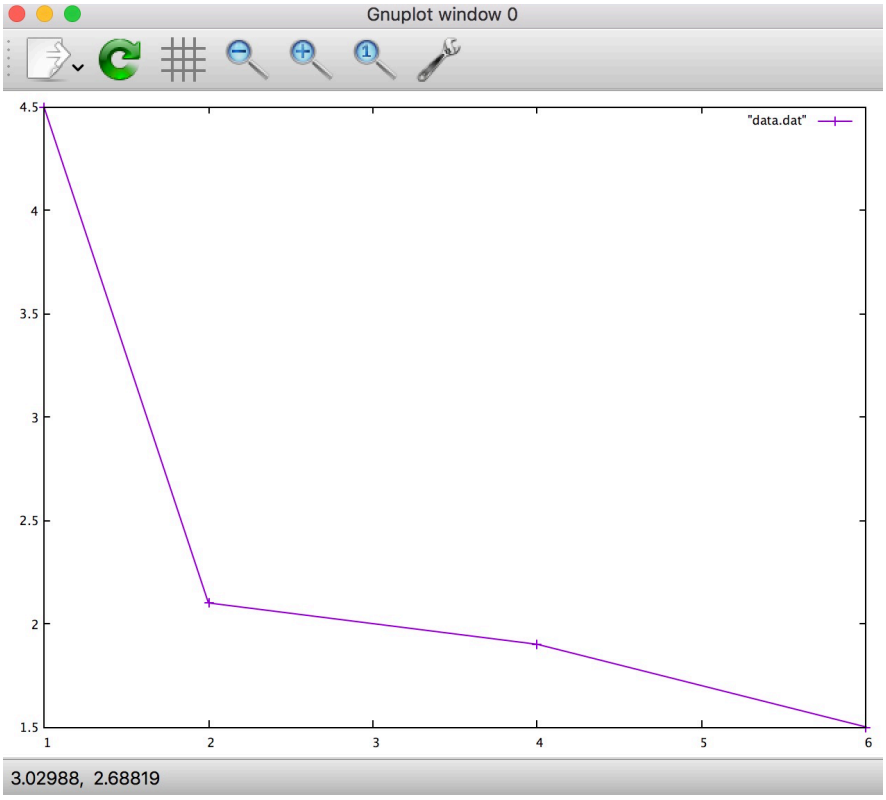
This will create 4 copies of Ncirc if there are 4 different threads. At the end of iteration, the values of 4 variables are added to the global variable.

```
int main () {
   long i;  long Ncirc = 0;
   double pi, x, y, test, time;
   double r = 1.0;   // radius of circle. Side of squrare is 2*r
   time = omp_get_wtime();
   #pragma omp parallel shared(num_trials) private(x,y)
   {

      {
           printf(" %d threads ",omp_get_num_threads());
           seed(-r, r);
      }

      #pragma omp for reduction(+:Ncirc)
      for(i=0;i<num_trials; i++)
      {
         x = drandom();
         y = drandom();
         test = x*x + y*y;

         if (test <= r*r) Ncirc++;
       }
```

```
1 threads
100000000 trials, pi is 3.141494  in 4.19
```

```
4 threads  4 threads  4 threads  4 threads
100000000 trials, pi is 3.141760  in 1.79
```

1 threads	4.5 s
2 threads	2.1 s
4 threads	1.9 s
6 threads	1.5 s



2_1_a

6 cores relative to 1 is 4.5/1.5 = 3

2_1_b

4 cores relative to 1 is 4.5/1.9 = 2.36

2_1_c

Thread creating and destruction overload causes this gap. The time taken on 4 cores is a little more than the time taken on 1 core.

Q3

The following changes had to be made in the code:

1. Race condition for loop variables i and j were present
2. Removed by making them private

```
do {
   diff = 0;
   #pragma omp parallel for private(i,j)
   for ( int i=1; i<n-1; i++) {
      // printf("Row %d processed by %d\n",i,omp_get_thread_num());
      for ( int j=1; j<n-1; j++) {
         b[i][j] = 0.25 * (a[i][j-1] + a[i-1][j]+ a[i+1][j] + a[i][j+1]);
         /* Determine the maximum change of the matrix elements */
         h = fabs(a[i][j] - b[i][j]);
         if (h > diff)
         diff = h;
      }
   }

#pragma omp parallel for private(i,j)
   for (int i=1; i<n-1; i++) {
   for ( int j=1; j<n-1; j++) {
   a[i][j] = b[i][j];
   }
   }

   k++;
   } while (diff > eps);
```

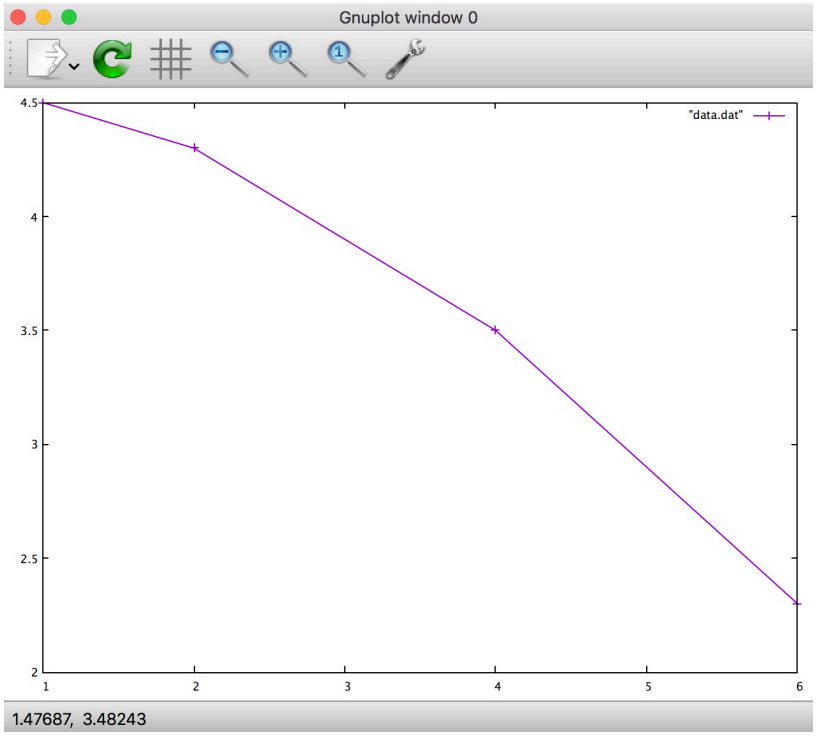
Before parallelizing: 4.5 seconds
After parallelizing (4 threads): 2.5 seconds

3_3_a Speed up for 6 cores = 4.5/2.3 = 1.95

```
Result: 353 iterations
  a[ 874][ 125] = 3.5430564493367842 1e-21
  a[ 874][  62] = 2.5713047903121342 5e-06
  a[ 500][ 500] = 0
  a[  62][ 874] = 2.5713047903121346 7e-06
  a[ 125][ 874] = 3.5430564493367842 1e-21
Runtime: 2.392 s
Performance: 0.588 GFlop/s
```

8 threads output

Thread number	1 thread	2 threads	4 threads	6 threads	8 threads
time	4.5 s	4.4 s	3.5 s	2.3 s	2.39 s



3_3_b Comparison with static, dynamic, and guided scheduling with chunk sizes 100 and 200

	Dynamic	Static	Guided
Chunk size 100	5.4	3.05	4.64
Chunk size 200	5.13	3.30	5.67

Why?

The best performance was obtained with chuck size 100 and static scheduling. Each iteration has approximately same amount of workload.

Q4

4.1 | No race condition. Balance is 0 after computation.

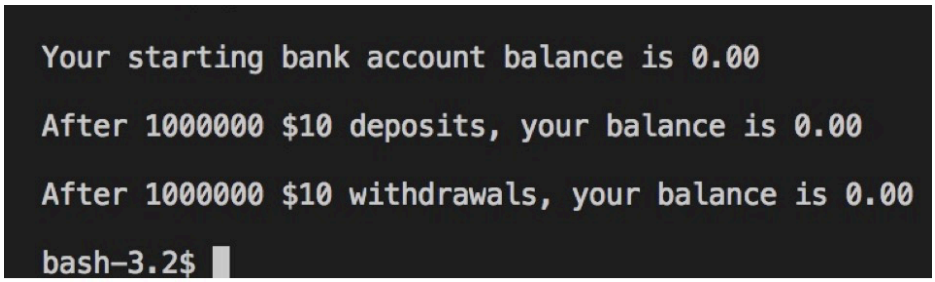
4.2 |

Due to race condition the data was inconsistent

4.3 |

By making it private, each thread made a local copy of the balance variable.

Data was consistent even after 1 and 4 threads.



4.4

Removed race condition my making atomic operations. Data remained consistent even for more threads.

4.5

In above part we ensured synchronization using atomic operations. Time taken was more. But now can make use of extra threads by dividing the work, create different balance variables and add them together.

Lot of time is saved using reduction operation. The decision of using ‘atomic’ operation or ‘reduction’ has to be done by the programmer as per context.

Running for 4 threads