Code Design:-

```
1.Input as adjacency matrix of size n*n. Total vertex = n.
2.For vertex v declare its partition by random between 1 and k.
3.Check for all vertices whether it is internal vertex or external vertex of i
t's own partition and store the result in ievertex array
4.Create a colour array of size n.

5.Create vector for each partition. This vector consists which vertices belong
 to partition. EX:- partition 4 contains <2,8,6> vertices.
6.Pass the vector into thread. EX:- partition 4 contains <2,8,6> vertices so
thread_id==4 will get <2,8,6> as input

Each thread will work as follows :-
    Grab a vertex from vector.
    check whether it is internal vertex or external vertex.(Compare the value
in ievertex array)
    if internal:
        colour with minimum available.
    if external:
        check with other threads and then colour.
```

The above was a rough pseudo code. The actual design implementation is as follows:-

Basic Data structures:-

```
int **adj;
int *partitionArray;
int *ievertex;
int *colour;
```

adj is adjacency matrix representation of graph.

PartitionArray is like first grab a vertex and decides in which partition it will go randomly.The index is that vertex and element is that partition to which the vertex will belong.Also 0<=number of partitions<k.

Ievertex:-After all vertices are in their respective partition then check for all vertices whether they are internal vertex or external vertex for that partition.

Ex:Vertex 5 is in partition 9 and it is an internal vertex of partition 9.So PartitionArray[5-1]=9 & Ievertex[5-1]=0 (0-Internal 1-External).

Colour is used to store colour of element.

If vertex 3 is initially uncoloured then colour[3-1]= -1.

If vertex 4 is coloured 5 then colour[3-1]= 5.

Functions:-

```
1.createPartitions (int *partitionArray,int n, int k);
```

To fill the partitionArray.

It goes like this:-For index 0 of the array OR for the vertex number 1:

Choose a partition p randomly between 0 and k-1.Since there are k partitions.

```
2.void checkVertex(int **adj,int *partition,int n,int k,int vertex,int partitionNum,int *ievertex);
```

It is used to fill ievertex array.

Our considered vertex is (int vertex).Now it is in partition (int partitionNum).Now from the graph adj,check whether the neighbours have the same partition (int partitionNum) by comparing their partition values from partitionArray.

Store the result in ievertex[vertex].

```
3.void ThFunctionCoarseGrain(int partitionNumber,vector<int> vertexInPartition, int n);
```

This function decides the colouring of partition vertices by resolving conflict on external vertices by using the coarse grain locks.

1.Grab vertices from the vector

2.Check whether the currently grabbed vertex is internal or external by comparing values in the ievertex array.(We previously calculated this for all vertices)

3.A.Internal vertex:-

Internal vertex will not cause any synchronisation problem with other threads.It's colouring is completely dependent on current thread/partition.So Colour it with minimum available(will be discussed shortly)

3.B.External vertex:-

External vertex will cause synchronisation problem with other threads. It's colouring is dependent on current thread/partition as well as threads which contains it's neighbours.

Now coarse grain is single lock for all external vertices trying to modify the colour array.

Implementation:-

```
mutex coarseLock;
```

This creates our lock.

```
1.coarseLock.lock();
```

2.Colour it with minimum available

```
3.coarseLock.unlock();
```

This lock will prevent other threads which wants to colour their external vertices. That is other threads have to wait until the colouring of this thread is done. This ensures synchronisation for colouring.

```
4.void ThFunctionFineGrain(int partitionNumber,vector<int> vertexInPartition,i
nt n)
```

Everything is same except for locking of the external vertex.

Fine Grain are multiple locks which is used to lock the external vertex as well it's neighbours.

For more clear picture . If an external vertex 8 in partition 3 wants to colour itself then it tries to acquire locks for itself as well as it's neighbours. (Here the neighbours maybe external vertices or internal vertices.But since internal vertices will not create conflict for colouring the current external vertex so we will exclude them from locking.This increases efficiency as compared to problem statement which states "When the thread Thi wishes to colour a boundary vertex bvx, it locks

all the neighbours of bvx and bvx as well.") .

NOW,To acquire lock for itself and it's external neighbours we do locking in increasing order as given in the problem statement "obtains the locks in an increasing (or decreasing) order of vertex ids to avoid deadlocks."

First,Create a vector for locking the vertex and it's external neighbours

Second,Create a vector which shows how many vertex are currently locked.

Now,The Fine Grain works as follows:-

While(1)

Acquire locks for vertex (1 lock) and external neighbours(m).

If Acquires all locks (LOCKS ==m+1) then Locking is success break out of the loop and continue execution

Else Failure, Release the currently locked vertices(stored in vector) to avoid deadlock. The currently locked vertices will become empty. Repeat the loop

Now we have acquired all locks,

So Colour with minimum available

And Release All Locks.

Logic behind selecting minimum colour:-

A vertex 5 has 2 edges(2 neighbours).The colour of vertex 5 should be different than colour of 2 edges(2 neighbours).Thus available colours for vertex 5 will be from 0 to number of edges(inclusive)

i.e. 0,1,2 these colours are available for vertex 5. If its neighbours have the following colouring:-

Same colouring:- -1 both uncoloured.So vertex 5 will be coloured 0.

0 to 2.Select the minimum available colour.

>2.Select 0 as the minimum colour.(Both coloured -29,so minimum available is 0)

Different colours:- There will always be colour in 0 to 2 that can be used to colour vertex 5.Since there are 3 possible minimum colours.

Generally Speaking, The minimum available colours will always lie between 0 and number of edges of that vertex(both inclusive).

Implementation of Colouring:-

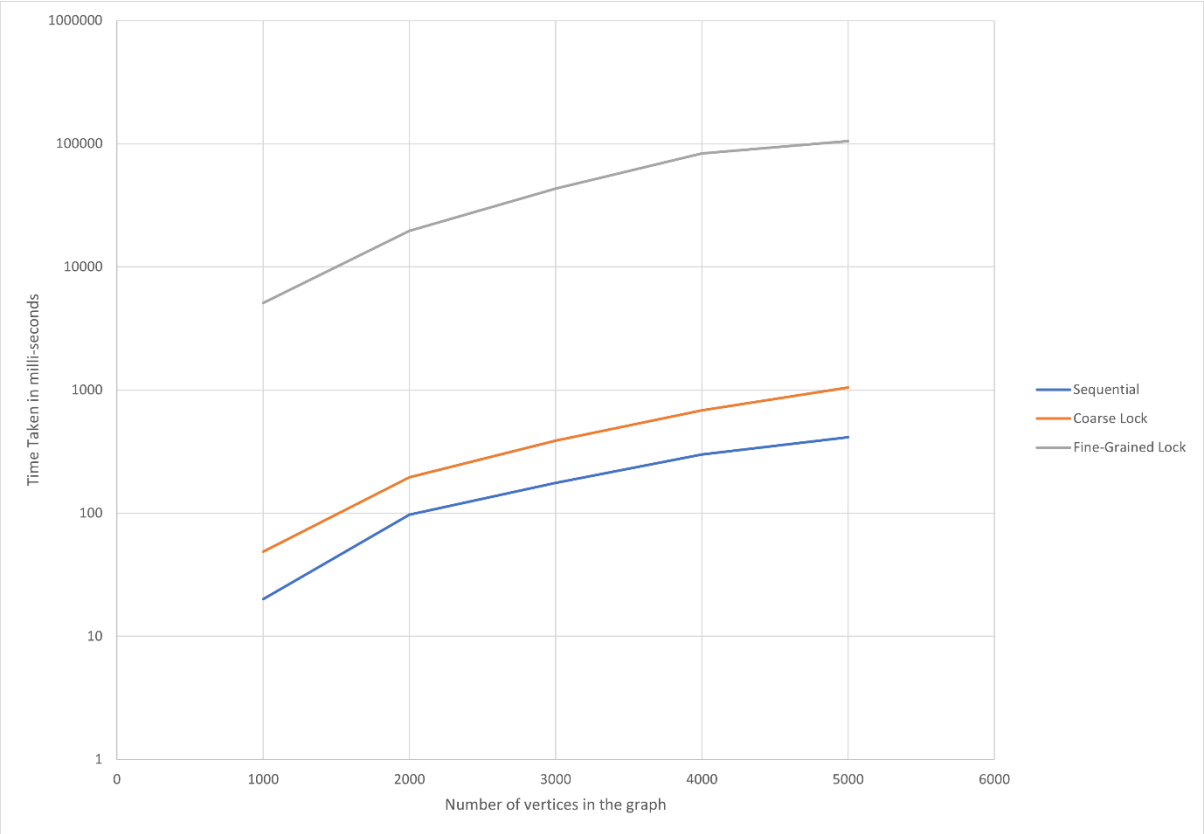
Create local_available array of size edges+1.

Check neighbour colours in this range of local availables.(Since out of range values of colours will be of no use to us .Also ignore -1 colour).

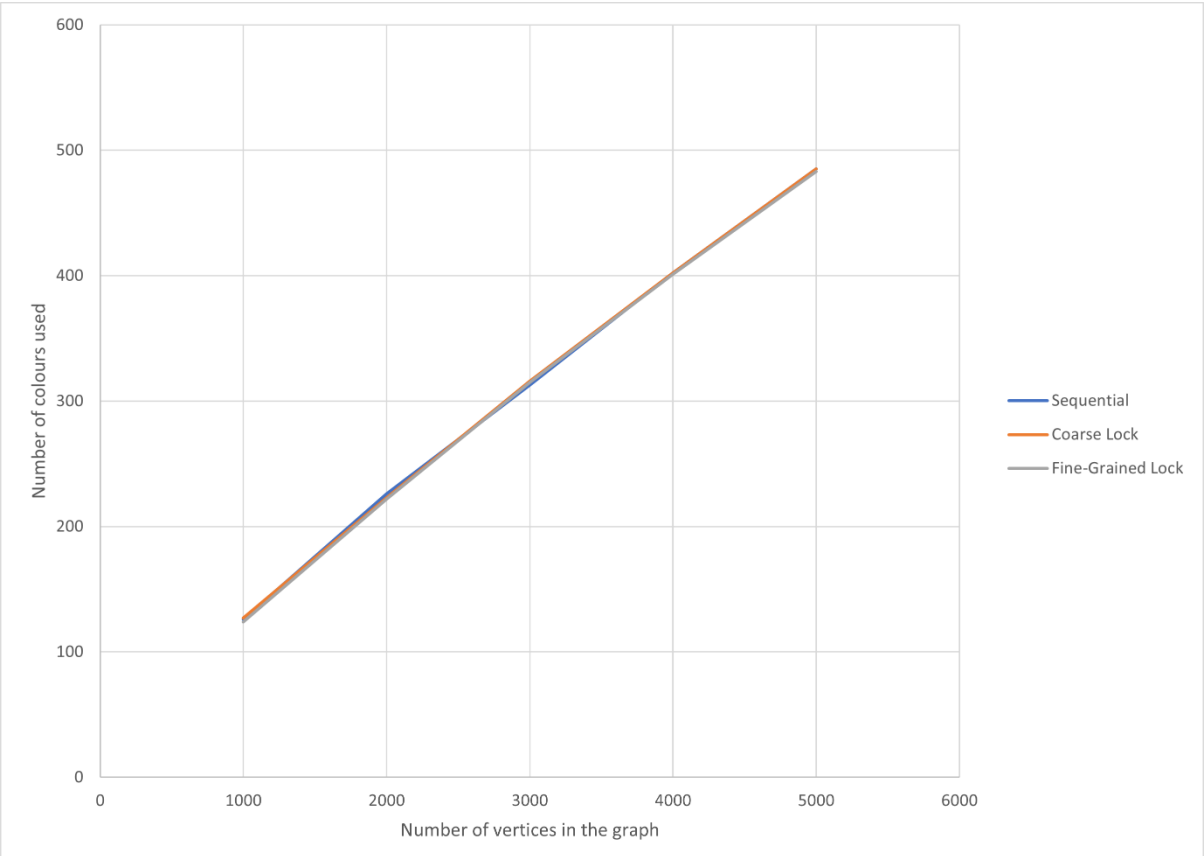
Find the lowest index which is available.

We successfully coloured with minimum available.

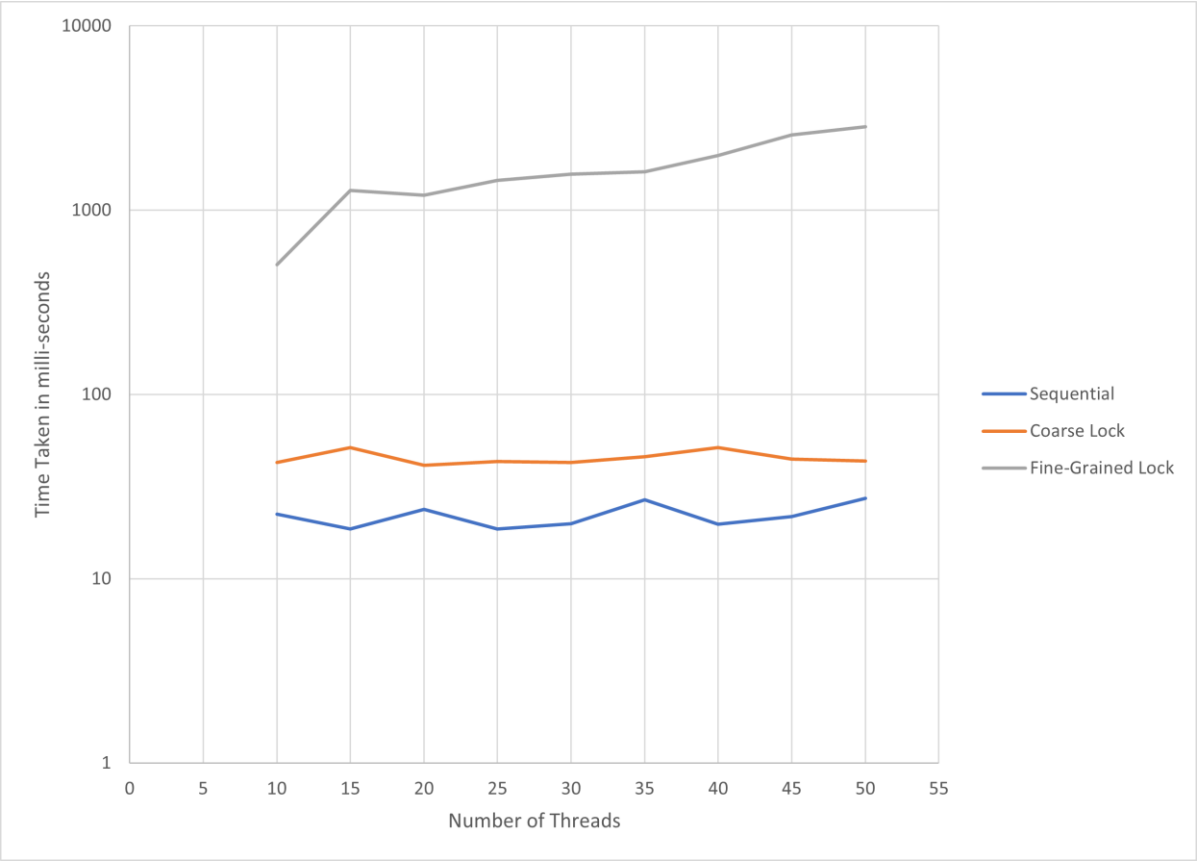
PLOT 1:-


1000000
100000
10000
1000
100
10
1
Time Taken in milli-seconds
0
1000
2000
3000
4000
5000
6000
Number of vertices in the graph
Sequential
Coarse Lock
Fine-Grained Lock

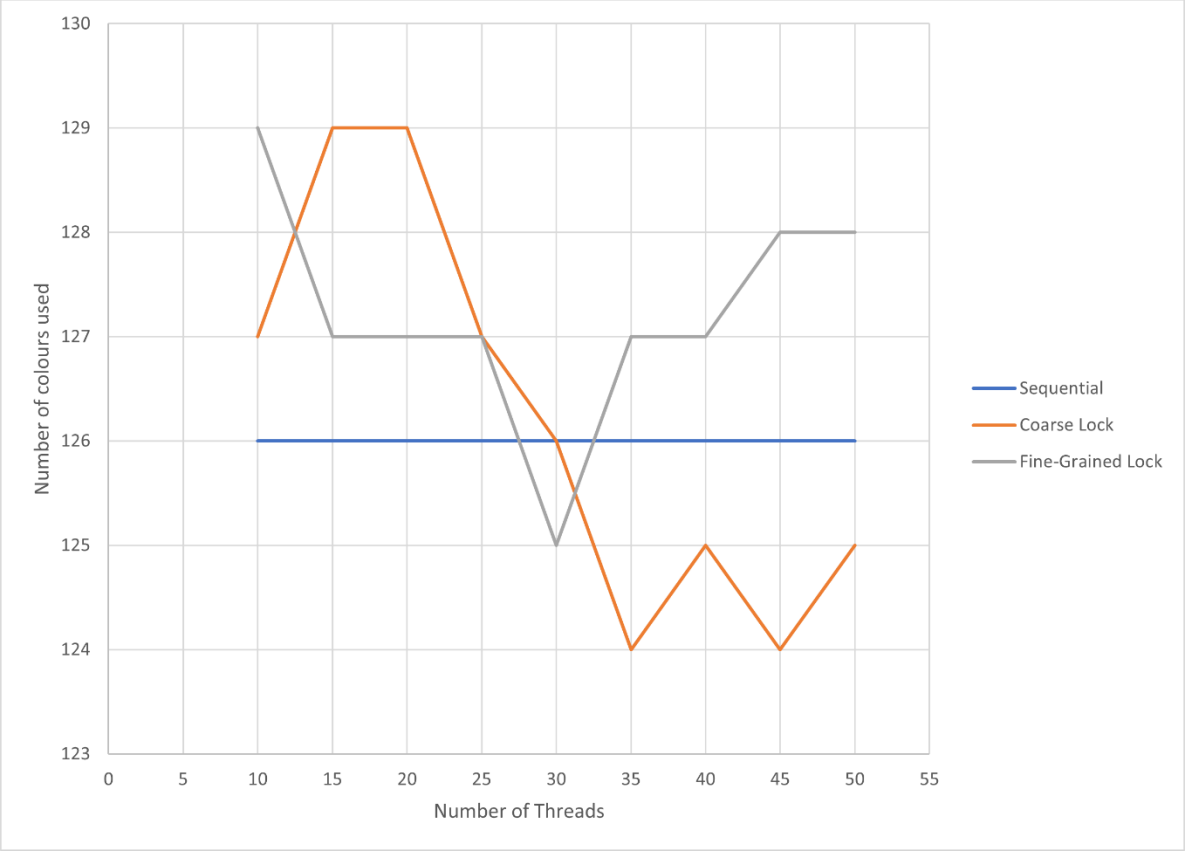
PLOT 2:-


600
500
400
300
200
100
0
Number of colours used
0
1000
2000
3000
4000
5000
6000
Number of vertices in the graph
Sequential
Coarse Lock
Fine-Grained Lock

PLOT 3:-


10000
1000
100
10
1
Time Taken in milli-seconds
0
5
10
15
20
25
30
35
40
45
50
55
Number of Threads
Sequential
Coarse Lock
Fine-Grained Lock

PLOT 4:-


130
129
128
127
126
125
124
123
Number of colours used
0
5
10
15
20
25
30
35
40
45
50
55
Number of Threads
Sequential
Coarse Lock
Fine-Grained Lock

Analysis of Plot1:-

Fine grain takes most amount of time since it locks the vertex and neighbours.

Coarse Grains take time between Fine grain and sequential since it has only one lock.

Sequential does not take much amount of time since it does not wait for any lock

Analysis of Plot2:-

Fine grain, Coarse Grain and Sequential algorithm use almost nearly same amount of colours when n is order of 10^3

Analysis of Plot3:-

Fine grain still takes most amount of time since it locks the vertex and neighbours.

Coarse Grains take time between Fine grain and sequential since it has only one lock.

Sequential does not take much amount of time since it does not wait for any lock.

Analysis of Plot4:-

Fine grain and Coarse Grain oscillate nearby Sequential.

In sequential number of colours used is independent of threads.