Vinta Reethu
Operating systems - II
Indian Institute of Technology Hyderabad

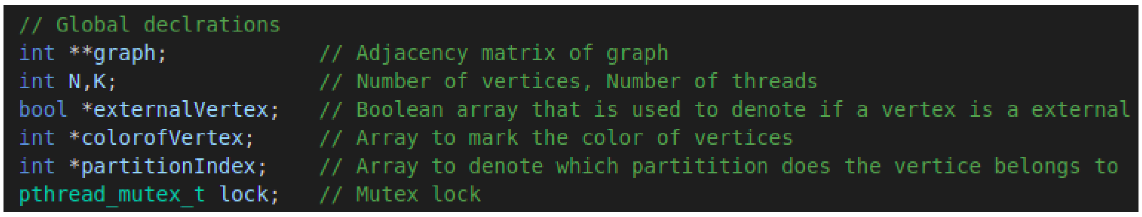
Assignment - 5

Vinta Reethu - ES18BTECH11028

April 12, 2021

Implementation details

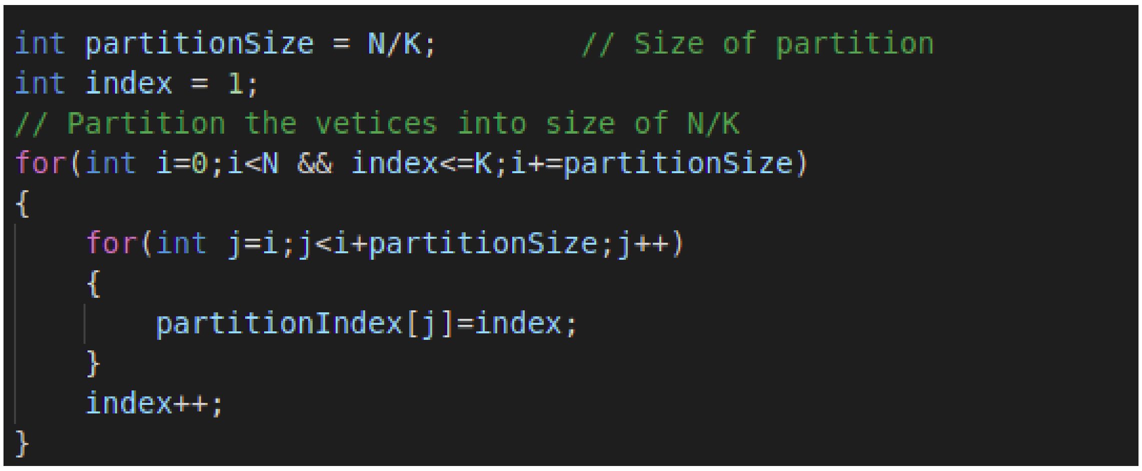
- These are the global declarations. These variables are to store values like adjacency matrix of graph, number of vertices, number of threads, and couple of arrays.



```
// Global declrations
int **graph;            // Adjacency matrix of graph
int N,K;                // Number of vertices, Number of threads
bool *externalVertex;   // Boolean array that is used to denote if a vertex is a external
int *colorofVertex;     // Array to mark the color of vertices
int *partitionIndex;    // Array to denote which partitition does the vertice belongs to
pthread_mutex_t lock;   // Mutex lock
```

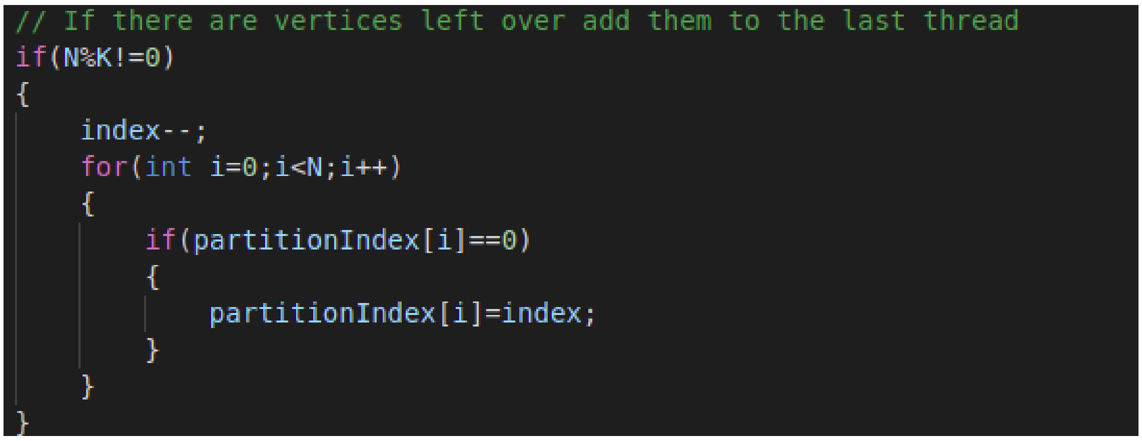
Algorithm Design

- We have a undirected graph with N vertices stored in the form adjacency matrix.
- After this we partition the vertices into K partitions, where size of each partition is N/K. We perform this partition in order from vertex 1 to vertex N. For example if N = 6 and K = 3 vertices 1, 2 get 1st partition and then vertices 3, 4 get into 2nd partition then vertices 5, 6 get into 3rd partition.



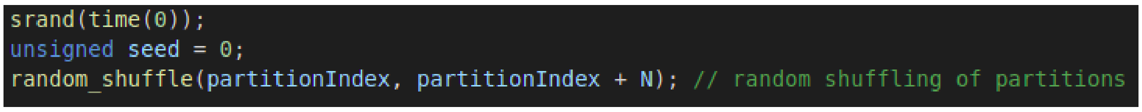
```
int partitionSize = N/K;        // Size of partition
int index = 1;
// Partition the vetices into size of N/K
for(int i=0;i<N && index<=K;i+=partitionSize)
{
    for(int j=i;j<i+partitionSize;j++)
    {
        partitionIndex[j]=index;
    }
    index++;
}
```

- If there are elements that couldn't get into any partition we add them to last partition. This happens when N is not divisible by K, so we add those vertices into the Kth partition.



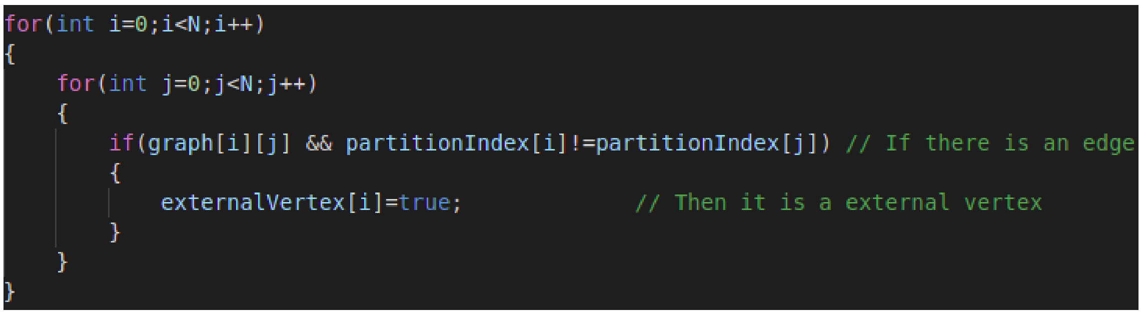
```
// If there are vertices left over add them to the last thread
if(N%K!=0)
{
    index--;
    for(int i=0;i<N;i++)
    {
        if(partitionIndex[i]==0)
        {
            partitionIndex[i]=index;
        }
    }
}
```

- For each vertex we store the partition index in a **partionIndex** array.
- After this, random shuffling is performed on the partionIndex array. This shuffles the vertices into different partitions randomly. For the above example because of the random shuffling vertices 4,2 can get into 1st partition and 1,6 can get into 2nd one and 3,5 into 3rd partition.



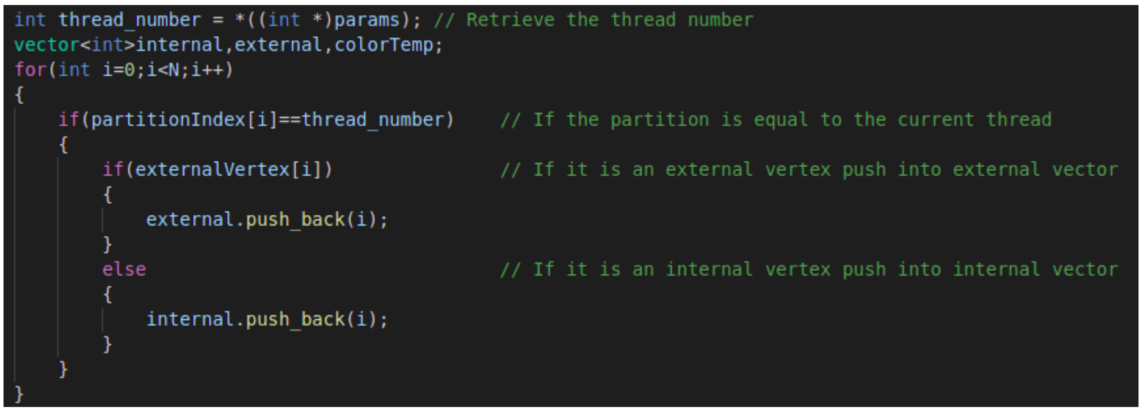
```
srand(time(0));
unsigned seed = 0;
random_shuffle(partitionIndex, partitionIndex + N); // random shuffling of partitions
```

- After this we try to check if the vertex is an external vertex or not. This is done by checking if there is an edge with other vertex, if there is an edge we check if these two vertices are in same partition. If they are in different partition it means that this is external vertex. We keep track of this in a boolean **externalVertex** array.



```
for(int i=0;i<N;i++)
{
    for(int j=0;j<N;j++)
    {
        if(graph[i][j] && partitionIndex[i]!=partitionIndex[j]) // If there is an edge
        {
            externalVertex[i]=true;             // Then it is a external vertex
        }
    }
}
```

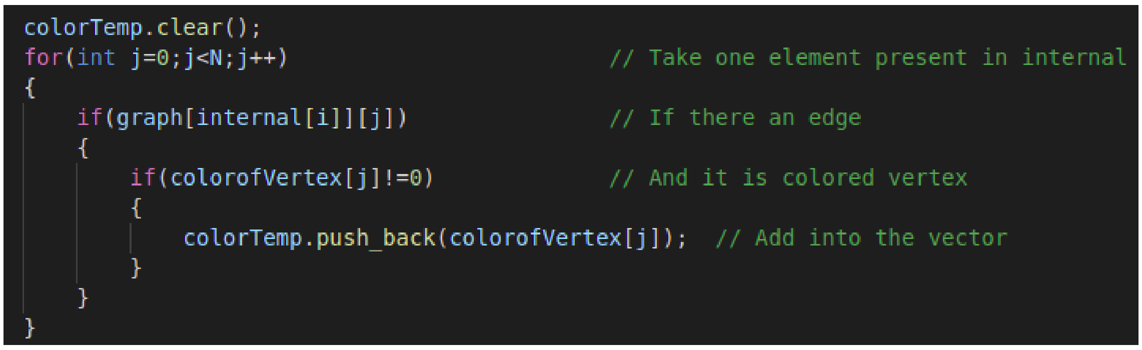
- After the pre processing is all done, K threads are created and thread function coloring is passed inside it along with thread number(which here it is also partition index).
- Once the thread enters the function we take all those vertices whose partition index is the current thread and check if the vertex is internal or external vertex, if it is internal vertex we add into internal vector else we add into external vector.



```
int thread_number = *((int *)params); // Retrieve the thread number
vector<int>internal,external,colorTemp;
for(int i=0;i<N;i++)
{
    if(partitionIndex[i]==thread_number)    // If the partition is equal to the current thread
    {
        if(externalVertex[i])               // If it is an external vertex push into external vector
        {
            external.push_back(i);
        }
        else                                // If it is an internal vertex push into internal vector
        {
            internal.push_back(i);
        }
    }
}
```

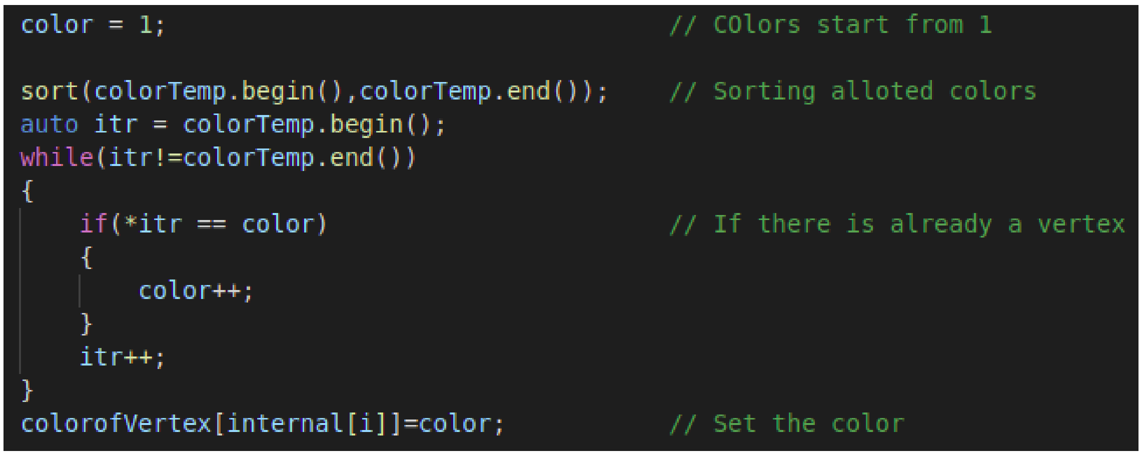
Internal vertices

- For each internal vertex we check the edges that are connected to it, if its neighbour is colored we add the color into the **colorTemp** vector.



```
colorTemp.clear();
for(int j=0;j<N;j++)                        // Take one element present in internal
{
    if(graph[internal[i]][j])               // If there an edge
    {
        if(colorofVertex[j]!=0)             // And it is colored vertex
        {
            colorTemp.push_back(colorofVertex[j]);  // Add into the vector
        }
    }
}
```

- We sort this colorTemp vector. Next we iterate through this colorTemp vector and give the smallest color that is not taken by any vertex.



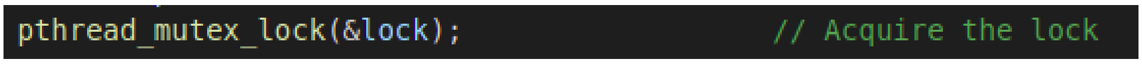
```
color = 1;                                  // COlors start from 1

sort(colorTemp.begin(),colorTemp.end());    // Sorting alloted colors
auto itr = colorTemp.begin();
while(itr!=colorTemp.end())
{
    if(*itr == color)                       // If there is already a vertex
    {
        color++;
    }
    itr++;
}
colorofVertex[internal[i]]=color;           // Set the color
```

External vertices

- Coarse and fine grained only differ in the number of locks and locking mechanism used. Once a thread acquires a lock the way of asssigning the color to the vertex is same in both the methods. Below is the implementation of locks in the two mechanisms, after that there is an explanation of how coloring is done for the external vertex.

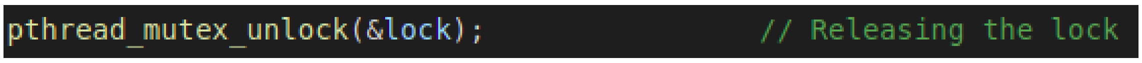
Coarse-Grained Lock :

- In this we use a mutex lock named **lock**. If a thread wants to color a external vertex it has to acquire the mutex lock if some other thread acquires the lock before this it has to wait until the lock is free.



```
pthread_mutex_lock(&lock);                    // Acquire the lock
```

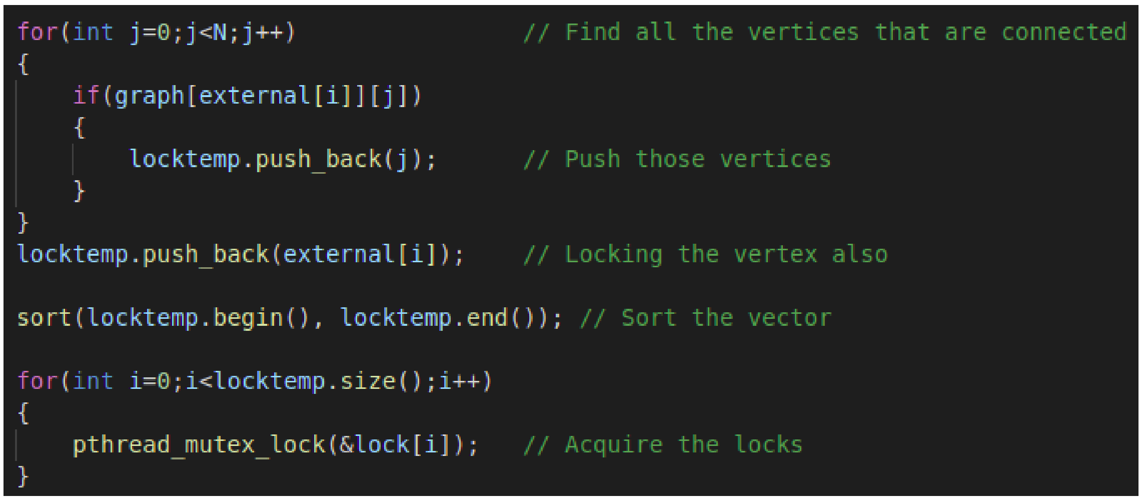
- Once we assign color to the external vertex we relase the mutex lock.



```
pthread_mutex_unlock(&lock);                  // Releasing the lock
```

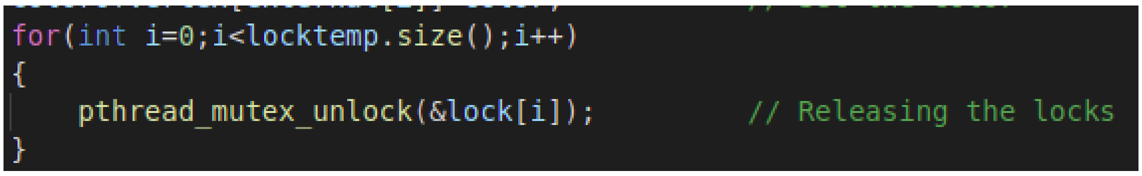
Fine-Grained Lock :

- In fine grained locking, we first store all the neighbours of the current vertex in a vector **lockTemp**. We also push the external vertex into it. Then we sort this vector and acquire lock one by one. If a lock is not available we wait until it is available. Since we are acquiring locks in increasing order deadlock will not occur.



```
for(int j=0;j<N;j++)                    // Find all the vertices that are connected
{
    if(graph[external[i]][j])
    {
        locktemp.push_back(j);          // Push those vertices
    }
}
locktemp.push_back(external[i]);        // Locking the vertex also

sort(locktemp.begin(), locktemp.end()); // Sort the vector

for(int i=0;i<locktemp.size();i++)
{
    pthread_mutex_lock(&lock[i]);       // Acquire the locks
}
```

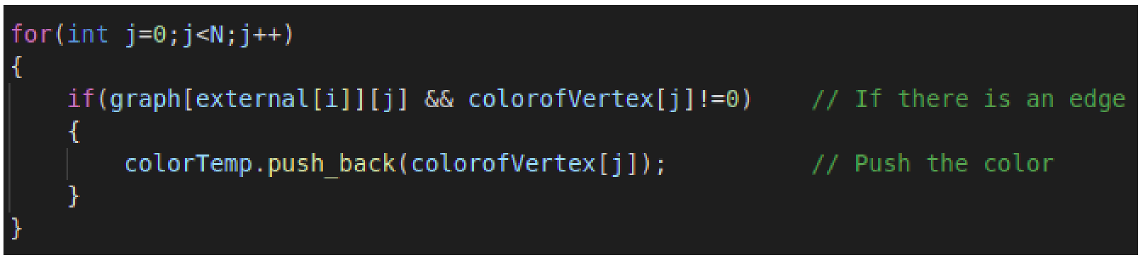
- Once we are done coloring the vertex we release the locks in the order we have acquired.



```
for(int i=0;i<locktemp.size();i++)
{
    pthread_mutex_unlock(&lock[i]);           // Releasing the locks
}
```

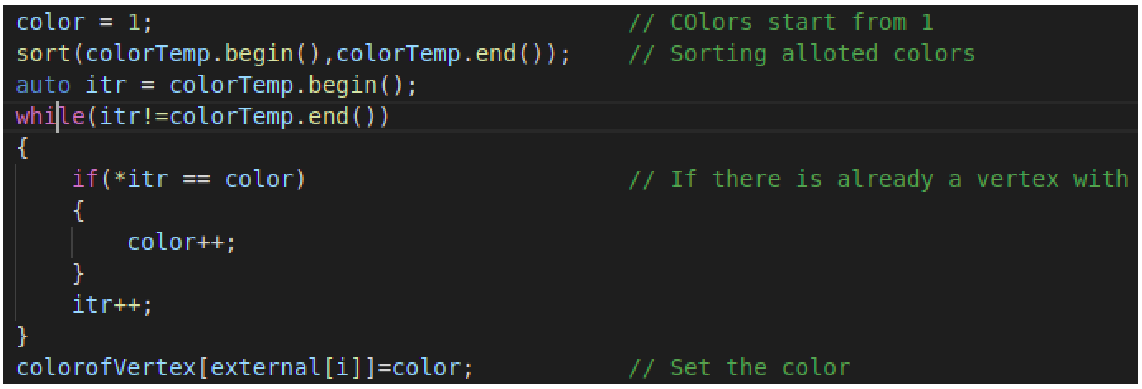
Coloring external vertex:

- For this external vertex we check the edges that are connected to it, if its neighbour is colored we add the color into the **colorTemp** vector.



```
for(int j=0;j<N;j++)
{
    if(graph[external[i]][j] && colorofVertex[j]!=0)    // If there is an edge
    {
        colorTemp.push_back(colorofVertex[j]);          // Push the color
    }
}
```

- We sort this colorTemp vector. Next we iterate through this colorTemp vector and give the smallest color that is not taken by any vertex.



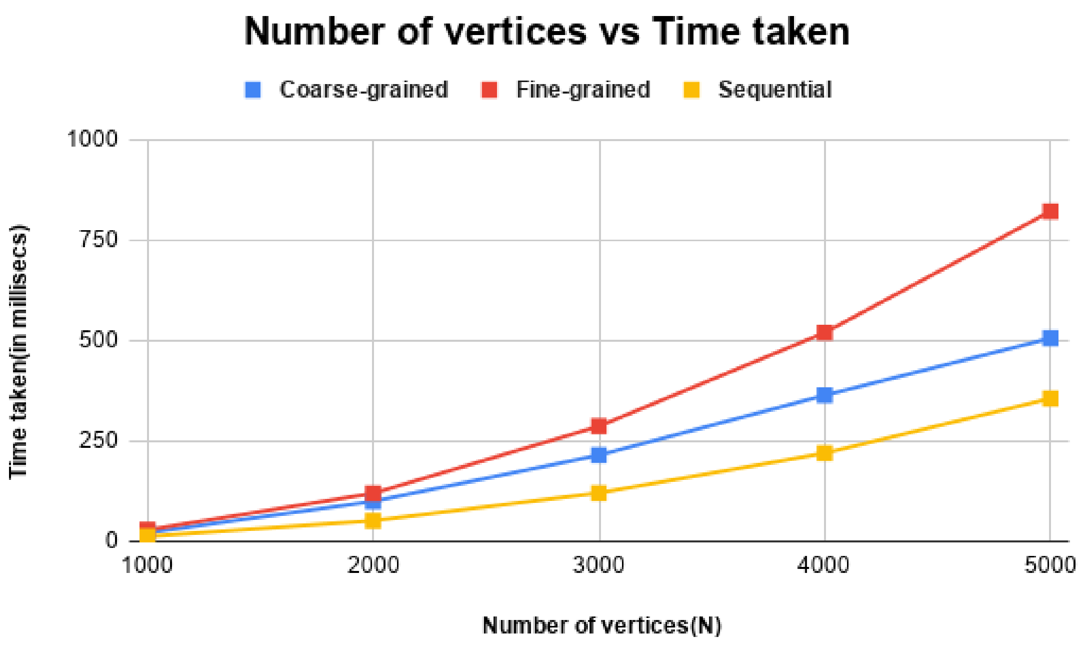
```
color = 1;                                   // COlors start from 1
sort(colorTemp.begin(),colorTemp.end());     // Sorting alloted colors
auto itr = colorTemp.begin();
while(itr!=colorTemp.end())
{
    if(*itr == color)                        // If there is already a vertex with
    {
        color++;
    }
    itr++;
}
colorofVertex[external[i]]=color;            // Set the color
```

- Time taken by the algorithm is measured using **chrono**.

Graph analysis

Plot 1

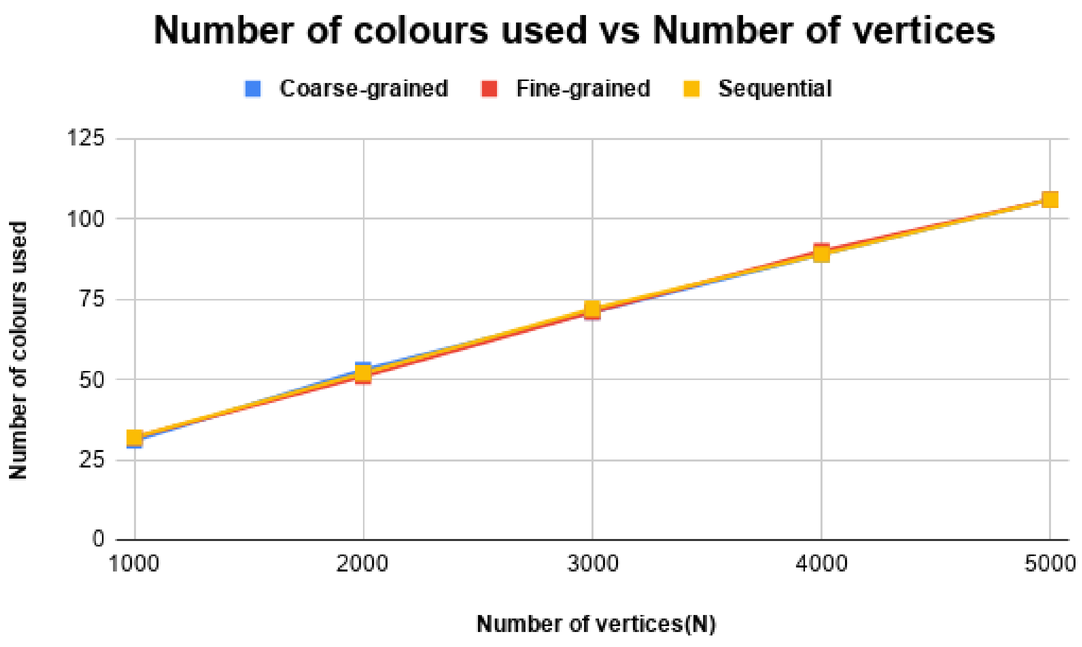
Here I have taken K = 50 and varied number of vertices in the graph from $1 * 10^3$ to $5 * 10^3$ in the increment of 10^3.



- Here as the number of vertices increase time taken in any method is also increasing as they have to color more number of vertices.
- Time taken by fine-grained is greater than coarse-grained because the number of locks needed to be acquired to color external vertex in fine-grained is more than 1 lock needed to be acquired in case of coarse-grained.
- Hence we can see in the graph that time taken by coarse is less than fine grained.
- In case of sequential there are no locks and any vertex to be colored need not wait hence it takes the least time.

Plot 2

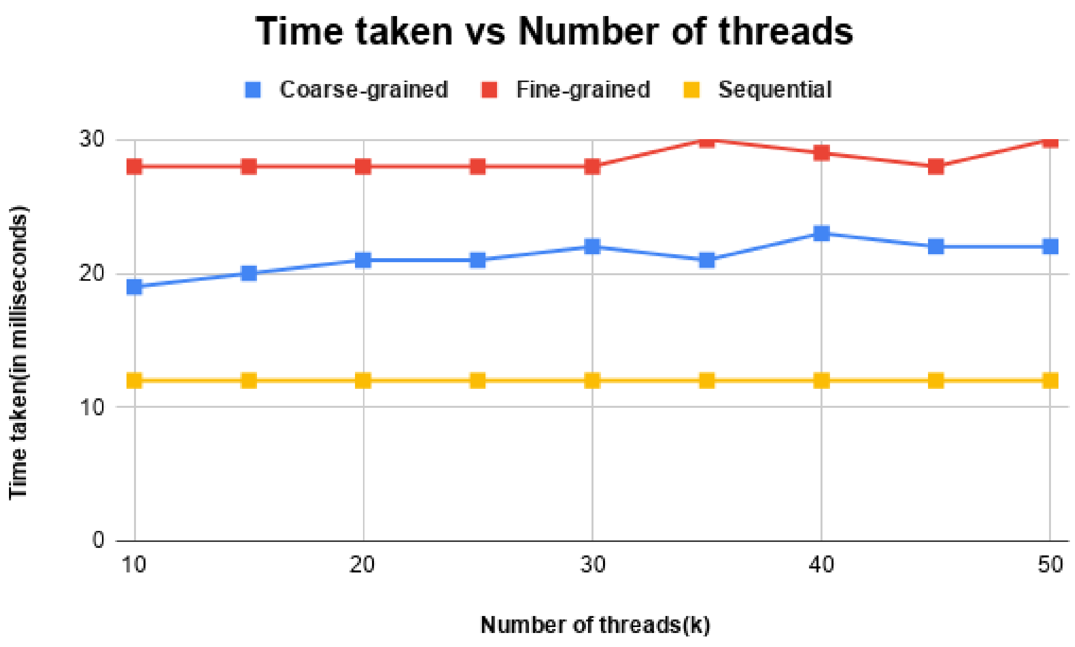
Here I have taken K = 50 and varied number of vertices in the graph from $1 * 10^3$ to $5 * 10^3$ in the increment of 10^3.



- As the number of vertices increase, the number of colors used by algorithm should also increase because the number of neighbouring vertices are increasing.
- The number of colors obtained in case of coarse, fine grained depend on the order in which they acquired the locks.
- In this setting it is observed that number of colors used are more or less the same in all the three algorithms.

Plot 3

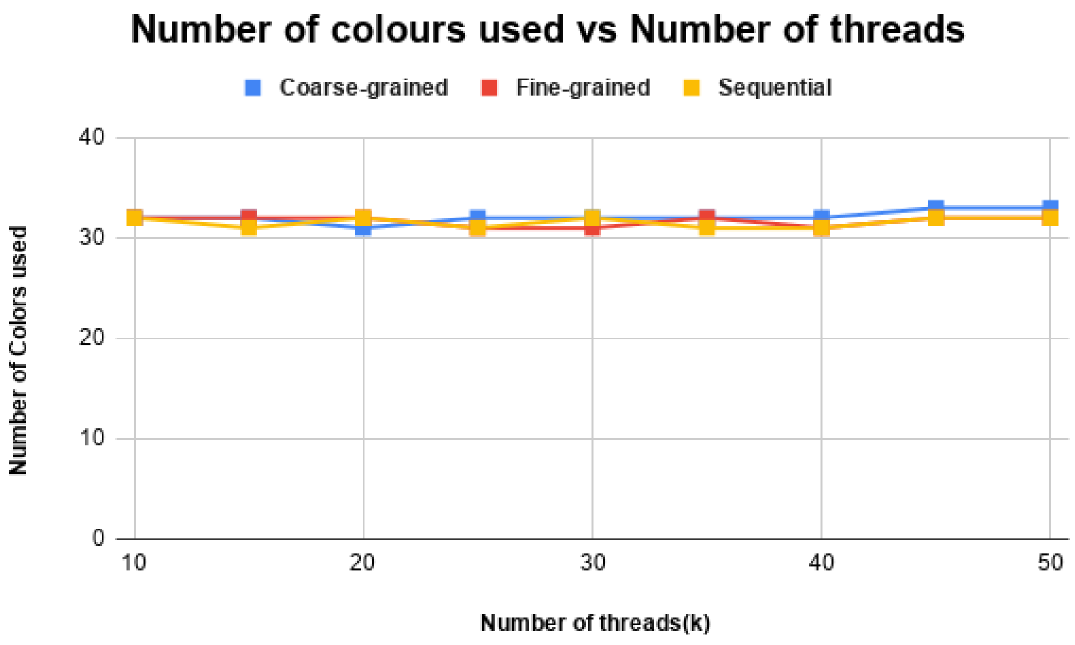
Here I have taken N = 10^3 and varied number of threads from 10 to 50 in the increment of 5..



- Here the number of vertices are fixed, hence even though number of threads increase time taken by sequential will not change.
- As number of threads increase, there is more competition for the locks hence time slightly increases.
- Time taken by fine-grained is greater than coarse-grained because the number of locks needed to be acquired to color external vertex in fine-grained is more than 1 lock needed to be acquired in case of coarse-grained.

Plot 4

Here I have taken N = 10^3 and varied number of threads from 10 to 50 in the increment of 5..



- Here the number of vertices are fixed, hence even though number of threads increase colors used by sequential should not differ by much.
- The number of colors obtained in case of coarse, fine grained depend on the order in which they acquired the locks.
- In this setting it is observed that number of colors used are more or less the same in all the three algorithms with slight variations here and there.