CS3523 : Operating Systems 2
Assignment 5
Implementing Graph Colouring Algorithm using locks

P. Naga Hari Teja
CS19BTECH11021

Problem Statement / Objective:

To implement the greedy graph colouring algorithm in parallel using threads and synchronize them with the help of locks.

Input:

The input to the program will be a text file, named “input_params.txt”, containing the total number of threads k followed by number of vertices n in graph. From next line graph is represented in adjacency matrix format.

Implementation

The following two approaches are used to colour external vertices.

Coarse-Grained Lock

For colouring an external vertex of a partition, the thread has to obtain the common global lock. The colour is then assigned in a greedy manner by looking at all the neighbour vertices of that external vertex.

Fine-Grained Lock

For colouring an external vertex of a partition, the thread has to obtain all locks corresponding to it’s neighbour vertices and also of vertex itself. The locks are obtained in an increasing order of vertex number. After obtaining the locks the colour is assigned in a greedy manner by looking at all the neighbour vertices of that external vertex.

Header Files

```
#include<iostream>
#include<fstream>
#include<sstream>
#include<thread>
#include<pthread.h>
#include<vector>
#include<set>
#include<chrono>
#include<ctime>
#include<cstdlib>
```

Class Node

Used for storing vertex number, partition number and colour of a node (vertex).

Note : Vertex Number, Partition Number and Colour are considered as int data type.

Global Variables

k : Total number of partitions / threads
n : Total number of vertices in graph

Note : k, n are considered as int data type.

vertices : Vector of nodes

partitions : Vector of vectors to store vertices in each parttion

adjMatrix : A 2d array representing adjacency matrix of the graph

Note : It is implemented as bool type and it's size is 10^4 x 10^4.

Lock in case of Coarse Grained Approach

mutex : Global mutex Lock, implemented as type pthread_mutex_t

Locks in case of Fine Grained Approach

mutexLocks : Vector of mutex locks implemented as type pthread_mutext_t

Program Execution

Opening Required Files

1. The "input_params.txt" is opened in read mode for reading input.
2. The file "output.txt" is opened (if not present then created) in the same directory as that of executable file. The contents are erased if already present.
3. The program terminates with exit status 1 if files cannot be created or opened.

Reading Input and Initializing vectors

1. Values of k, n and adjacency matrix entries are read from input file.
2. vertices (vector) is initialized with n Nodes.
3. partitions (vector of vector) is also initialized with k empty vectors.

Note : In case of Coarse Grained Approach, mutex is initialized using

```
pthread_mutex_init (&mutex, NULL);
```

In case of Fine grained approach, mutexLocks vector is filled with mutex locks and they are initialized in a similar way as above.

Partitioning the vertices into k disjoint non empty sets

1. A partition length is randomly chosen.
2. Some number of vertices (equal to partition length) are randomly chosen and added to partition.
3. The above two steps are repeated for k-1 times.
4. Finally all the remaining elements are considered as kth partition.

Creating Threads to colour each partition and noting time

1. The start time is noted in startTime by using chrono::duration_cast<Microseconds>(SystemClock::now().time_since_epoch()).count()
2. ‘k’ number of new threads are created and they begin execution from colorPartition function.
3. Each thread performs colouring of it’s assigned partition.
4. The main thread then waits for completion of n threads.

Common Thread Execution for both approaches in colorPartition

1. A set colorsUsed is created for storing colours of adjacent vertices of a particular vertex.
2. A vector externalVertices is created for storing external vertices.
3. For each vertex in the partition, it is determined whether all it’s adjacent vertices are present in the partition.
4. If all it’s adjacent vertices are present then it is designated as internal vertex, if not then added to externalVertices vector.
5. If a vertex is determined to be internal then all it’s adjacent vertices colours (if coloured) are determined and added to the set colorsUsed.
6. Then the vertex is coloured with the least colour which is not used from 1 to n and the set is cleared.
7. In the similar way, all the internal vertices are coloured.

Note : Program Execution now differs in two approaches based on colouring of external vertices.

Program Execution in Coarse Grained Approach

1. To colour an external vertex the thread tries to obtain the common global lock mutex using pthread_mutex_lock(&mutex).
2. After obtaining the mutex lock, then all it’s adjacent vertices colours (if coloured) are determined and added to the set colorsUsed.
3. Then the vertex is coloured with the least colour which is not used from 1 to n and the set is cleared.
4. In the similar way, all the external vertices are coloured.

5. Finally the global lock is released using pthread_mutex_unlock(&mutex).
6. Finally the thread exits.

Program Execution in Fine Grained Approach

1. To colour an external vertex the thread tries to obtain all it's neighbour vertices locks and also the lock for vertex itself in an **increasing order** using pthread_mutex_lock(&mutex).
2. After obtaining all the required mutex locks, then all it's adjacent vertices colours (if coloured) are determined and added to the set colorsUsed.
3. Then the vertex is coloured with the least colour which is not used from 1 to n and the set is cleared.
4. In the similar way, all the external vertices are coloured.
5. Finally all the mutex locks are released using pthread_mutex_unlock(&mutex).
6. Finally the thread exits.

Common Program Execution in both approaches after every thread exits

1. The end time is noted in endTime by using chrono::duration_cast<Microseconds>(SystemClock::now().time_since_epoch()).count()
2. Total Number of colours used is determined.
3. The required output is then printed to "output.txt"
4. Finally all the files are closed.

Output

One output file is generated, which contains the approach used, total no of colours used, time taken by the approach and colour assigned to each vertex.

Ex:

```
Coarse-Grained Lock
No of colours used: 3
Time taken by the algorithm using: 0.146 Milliseconds
Colours:
v1 - 1, v2 - 1, v3 - 1, v4 - 2, v5 - 3
```

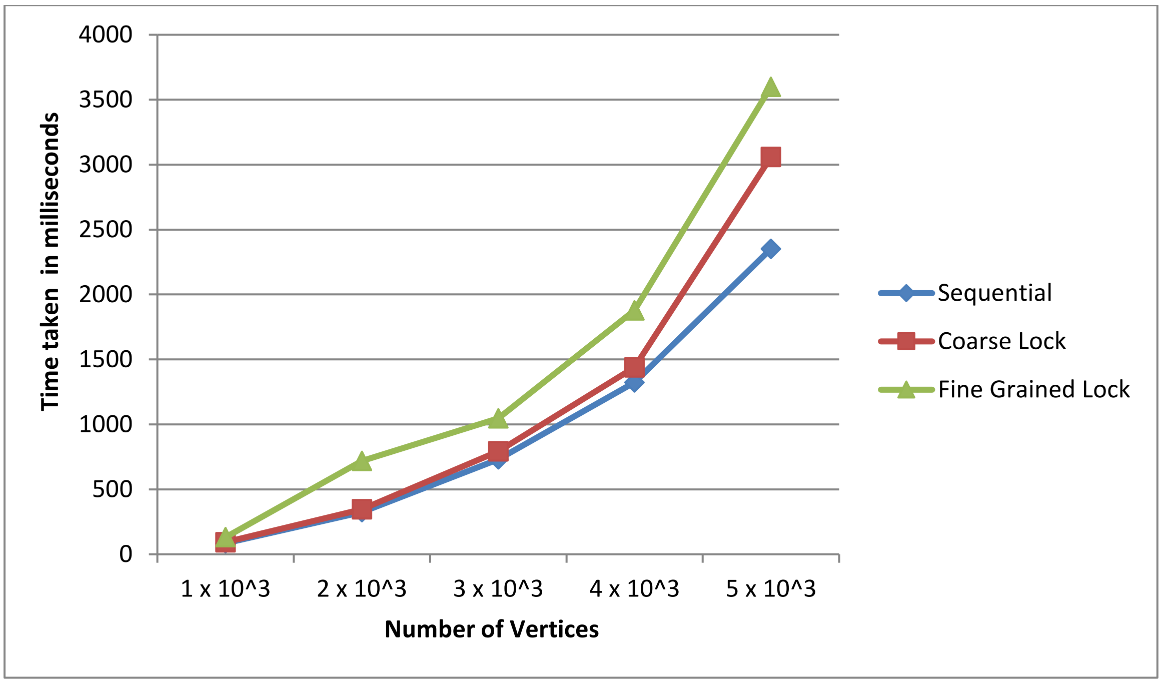
Note:
Time is calculated in Milliseconds with precision up to micro seconds.

Performance Comparison

<u>**Note : The number of vertices are taken in order of 10^3.**</u>

Plot 1

Note: The number of threads / partitions (k) is fixed to 25.

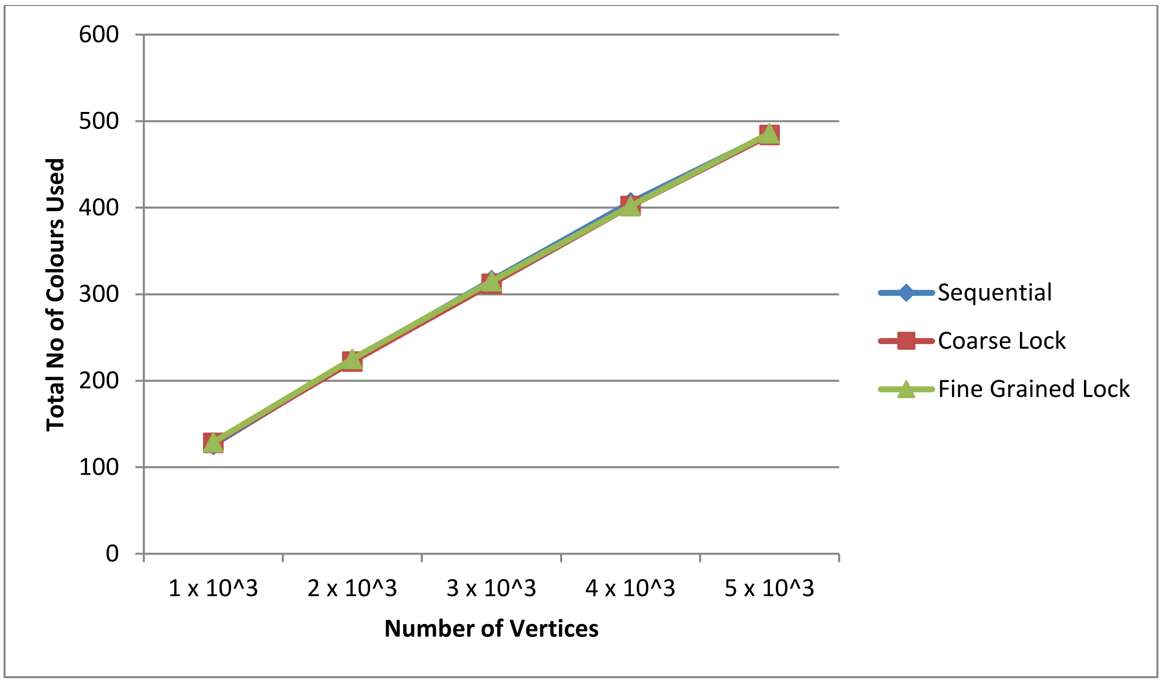


Analysis

1. As number of vertices are increasing the time taken for all 3 approaches is also increasing.
2. Sequential performs better than both Coarse Lock and Fine Grained Lock.
3. The increase in time due to overhead of thread creation dominates the decrease in time due to parallel execution using locks.

Plot 2

Note: The number of threads / partitions (k) is fixed to 25.

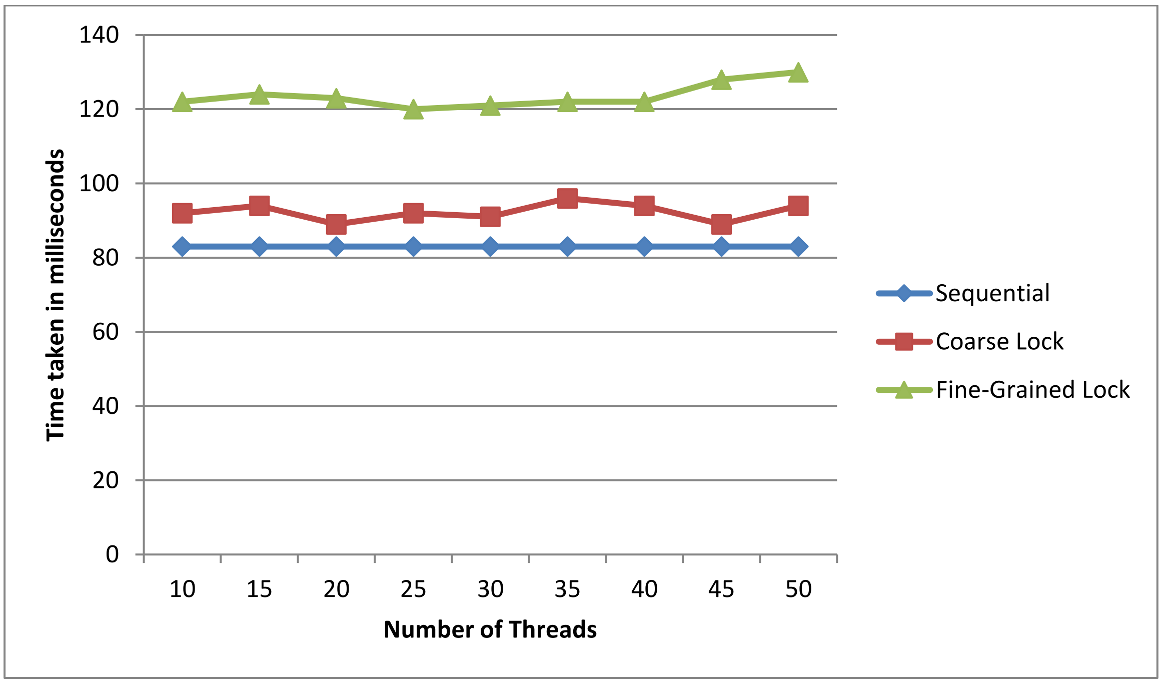


Analysis

1. All the 3 approaches uses almost same number of colours.
2. As number of vertices are increasing the total number of colours used are increasing.

Plot 3

Note: The number of vertices is fixed to 10^3.

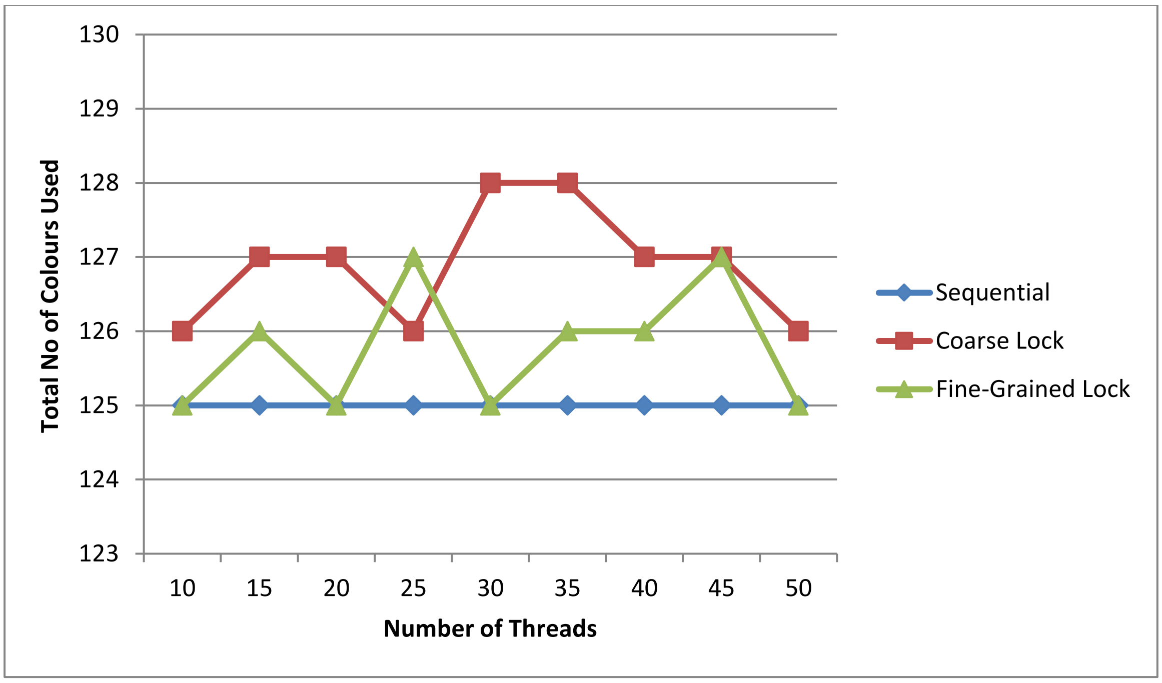


Analysis

1. The time taken by Coarse lock is almost same as number of threads are increasing.
2. Similarly time taken by Fine grained lock is also almost same as number of threads are increasing.
3. Sequential performs better than both Coarse Lock and Fine Grained Lock.
4. The increase in time due to overhead of thread creation dominates the decrease in time due to parallel execution using locks.

Plot 4

Note: The number of vertices is fixed to 10^3.



Analysis

1. All the 3 approaches use almost same number of colours.
2. As number of threads are increasing the total number of colours used are almost same for each approach.