Programming Assignment 5
Implementing Graph Coloring Algorithm using locks

Ananya Mantravadi
CS19B1004

CODE DESIGN & IMPLEMENTATION

Coarse-Grained Lock

```
mutex cg_lock;
int no_threads,v;
list <int> *adj;
int *colours;
int *partition;
bool *vertex_type;
int no_colours = 0;
ofstream output;
```

These are the global variables declared. `cg_lock` is the mutex lock that is used as the global lock whenever we colour an external vertex. `adj` list is for storing the graph information, `colours` contains the colour number assigned to each vertex, `partition` contains the partition number which each vertex belongs to, `vertex_type` holds the information of whether a vertex is internal or external. `no_colours` keeps track of the total number of colours used.

```
for(int i=0;i<v;i++){
    temp = rand() % no_threads;
    partition[i] = temp;
}
```

Partitions are created by assigning a random number from 0 to (number of threads-1). We then identify if each vertex is internal or external by iterating among its neighbour vertices and checking if all of them belong to the same partition. One observation is that while generating a graph with a huge number of vertices, the number of internal vertices found is almost zero.

We create an array of 'k' threads. Each thread is assigned to colouring vertices in its partition. The function `colour()` first recognizes the internal vertices of the partition assigned to that particular thread and colours them using a greedy approach by checking for internal conflicts with its neighbouring vertices, which are of the same partition. After internal vertices are coloured (this happens parallelly), external vertices need to be coloured one at a time, this is the critical section. For colouring any boundary vertex, in one of the partitions, the thread assigned to it obtains the lock on `cg_lock` and then assigns a colour to that vertex in a greedy manner.

```
//external vertices colouring - critical Section
    for (j=0; j<v; j++){
      //if vertex belongs to current partition and is a boundary vertex
        if(partition[j]==id)
          if(!vertex_type[j]){
            //acquire global lock
            cg_lock.lock();
            //apply greedy colouring
            for (it = adj[j].begin(); it!=adj[j].end(); ++it)
                if (colours[*it] != -1) //-1 indicates colour is not assigned
                    available[colours[*it]] = true;

            for (c = 0; c < v; c++)
                if (available[c] == false)//find lowest numbered coloured not used
                                          on prev coloured adj vertices
                    break;

            colours[j] = c;

            if(c>no_colours)
                no_colours = c; //update total number of colours

            for (it = adj[j].begin(); it != adj[j].end(); ++it)
                if (colours[*it] != -1)
                    available[colours[*it]] = false;  //reset for next iteration
            //release global lock
            cg_lock.unlock();
          }
    }
```

Fine-Grained Locks

The main design is similar to the above code but with the replacement of one global lock with an array of mutex locks of size the number of vertices.

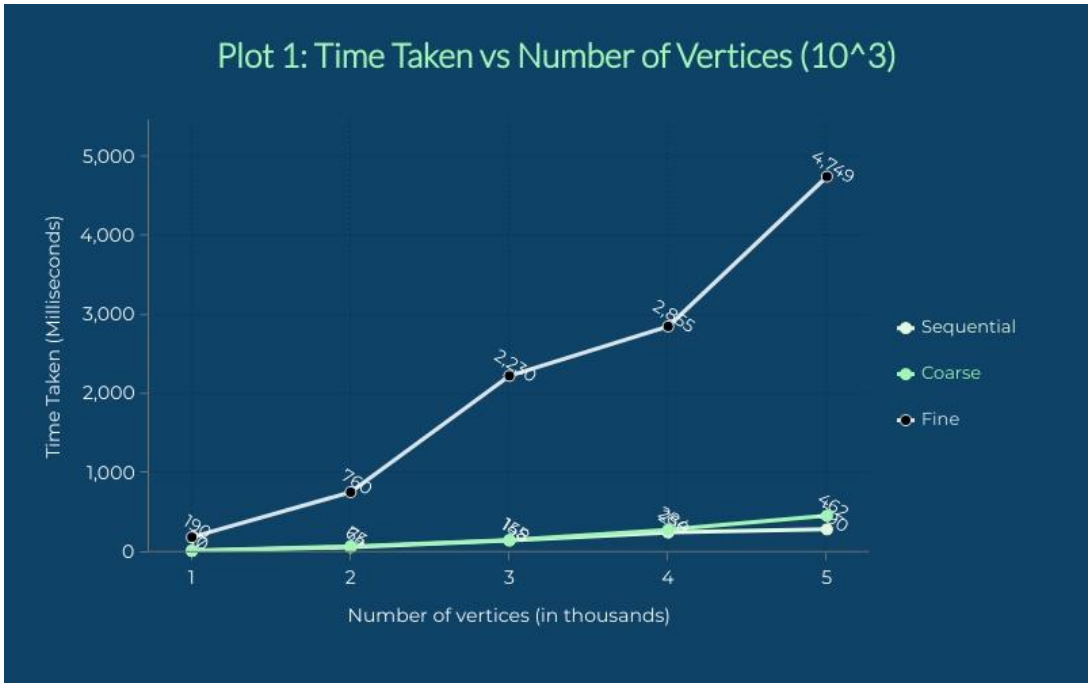
```
mutex *locks;    locks = new mutex[v];
```

Internal vertices are first coloured greedily as above. For each boundary vertex identified in the partition, the thread locks that boundary vertex along with all its neighbouring vertices. To prevent deadlocks, locking is done according to the increasing vertex ids. After obtaining the locks, it assigns a colour to the vertex after looking at the colours of the vertex's neighbours in a greedy manner as explained above.

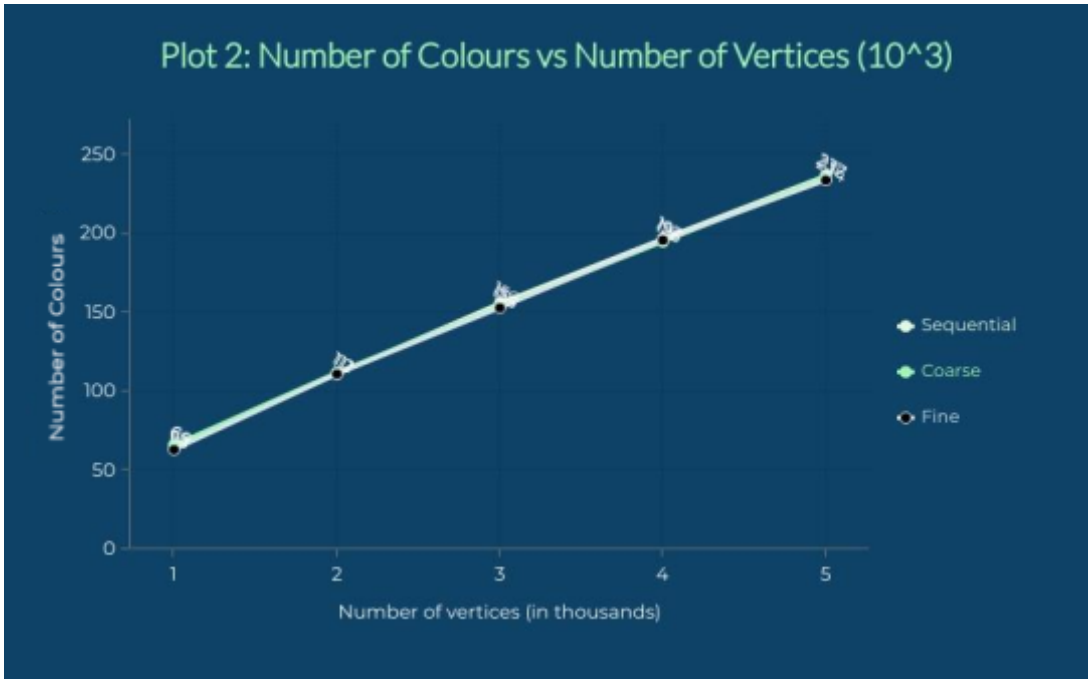
```
//external vertices colouring - critical section
  for (j=0; j<v; j++){
    if(partition[j]==id)
      if(!vertex_type[j]){
        //if vertex belongs to current partition and is a boundary vertex
        list <int> v_list;
        for(it=adj[j].begin();it!=adj[j].end();++it)
          v_list.push_back(*it); //add the vertex and its neighbours to a list

        v_list.sort(); //sort the list in increasing order

        for(it=v_list.begin();it!=v_list.end();++it)
          locks[*it].lock(); //acquire locks on vertices in order
        //apply greedy colouring
        for (it = adj[j].begin(); it!=adj[j].end(); ++it)
          if (colours[*it] != -1)
            available[colours[*it]] = true;

        for (c = 0; c < v; c++)
          if (available[c] == false)
            break;

        colours[j] = c;

        if(c>no_colours)
            no_colours = c;

        for (it = adj[j].begin(); it != adj[j].end(); ++it)
          if (colours[*it] != -1)
            available[colours[*it]] = false;
        //release locks on vertices
        for(it=v_list.begin();it!=v_list.end();++it)
          locks[*it].unlock();
        }
    }
```

The output of the above programs consisting of the number of colours used, the time taken by the algorithm, and the colours assigned to each vertex by the algorithm is generated in output-coarse.txt and output-fine.txt respectively.

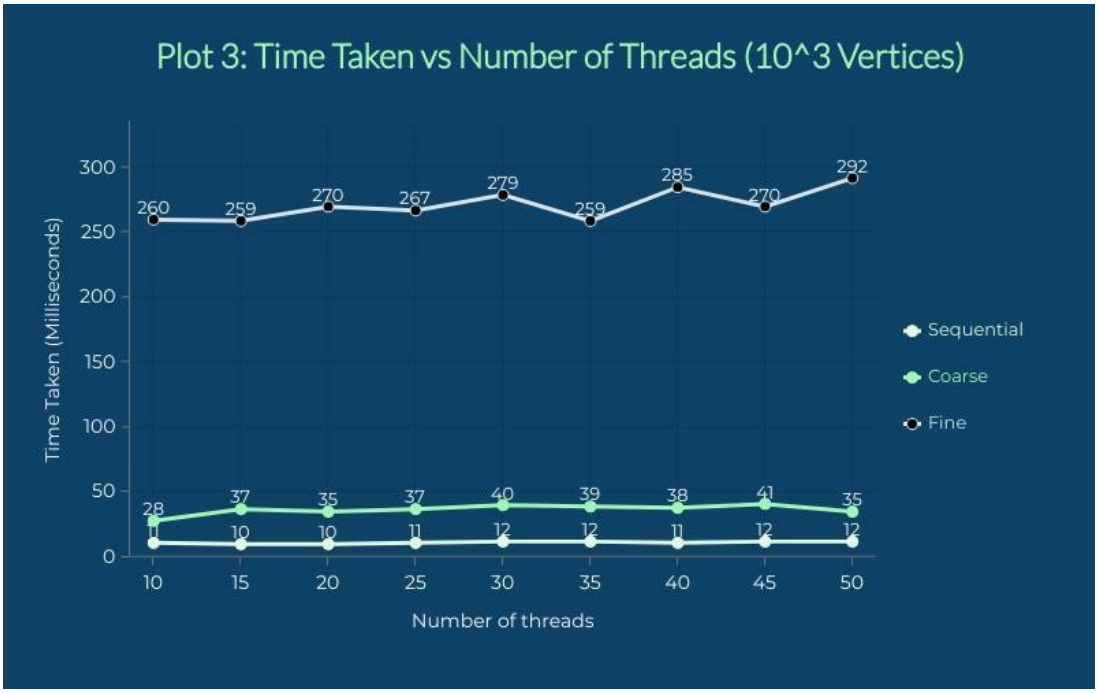
PLOT 1: Time Taken vs Number of Vertices



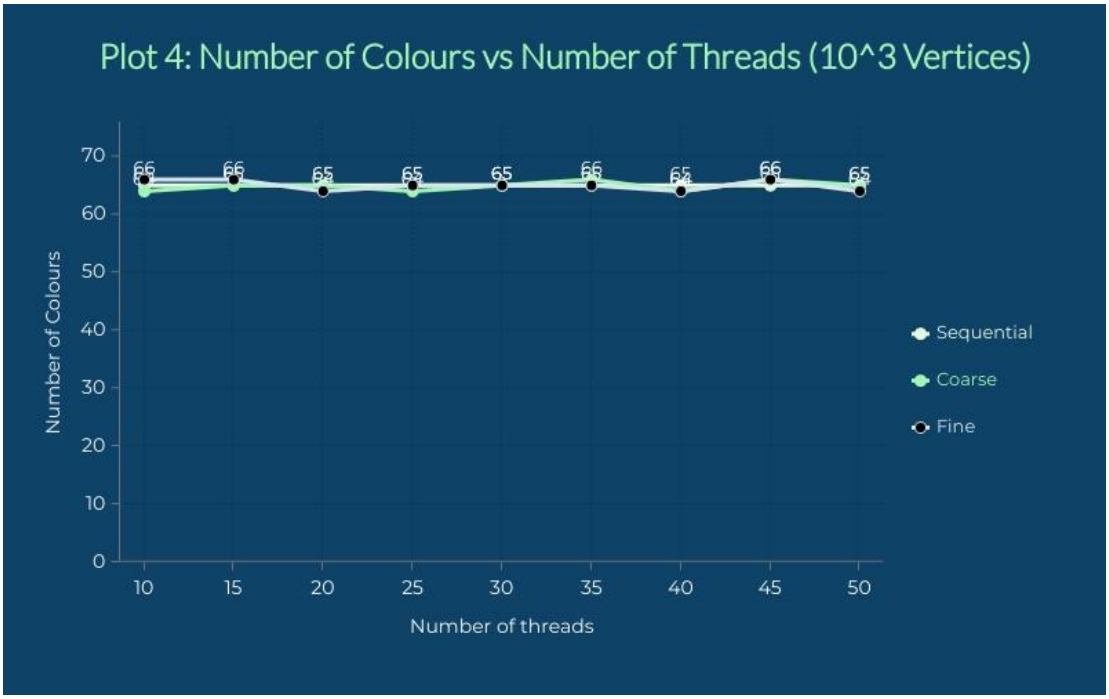
PLOT 2: Number of Colours vs Number of Vertices



PLOT 3: Time Taken vs Number of Threads



PLOT 4: Number of Colours vs Number of Threads



CONCLUSION

We observe that the time taken to execute the algorithms increases with the number of vertices, sequential being slightly faster than coarse-grained and fine-grained taking more time comparatively. This is for the inputs of the number of vertices in the order of 10^3 to $5*10^3$ with keeping the number of threads to two. Since almost all vertices are external, there is an overhead of obtaining the global lock each time in case of coarse-grained and multiple locks in case of fine-grained taking more time. The drawback we can see is if two vertices v_1 & v_2 from different partitions getting colored are adjacent to a vertex v_3, there is no need to lock v_3 if it is not getting colored. It is possible that with a larger number of vertices and an optimal number of threads, fine-grained algorithm will take a lesser amount of time. The number of colours used compared to each of the algorithms is also almost the same with varying numbers of vertices. We also notice that the time taken to compute does not change significantly with the varying number of threads in all three algorithms, since almost all vertices are external, and using multiple threads does not make much of a difference. The number of colours used in each of the algorithms is also almost the same with varying number of threads.