Implementing Multithreaded graph coloring

Topic - Coloring a graph in a Multithreaded manner such that no two adjacent vertices have the same color.

Submitted By: Kushagra Indurkhya
CS19B1017

Contents

Implementation design

Greedy Coloring Algorithm

To color a vertex such that it doesn't have the same color as any of its adjacent vertex the greedy approach is:

```
The color array is initialized to -1 for each vertex.
To color a vertex V:
    bool array, available_color[n] is initialized to false.
    For every vertex in the adjacency list of V :
        If it is already colored mark that color unavailable in the available_color array
    Color vertex V as the smallest color which is available (not marked false in available_Color array)
```

```
void color_vertex(int i)
{
    bool* available_color=new bool[n];
    for(int i=0;i<n;i++)
        available_color[i]=true;
    for(auto j:inp_graph->adj_list[i])
          if(color[j]!=-1) available_color[color[j]]=false;
    for(int k=0;k<n;k++)
        if(available_color[k])
        {
            color[i]=k;
            break;
        }
}
```

Sequential

This problem can be solved sequentially by iterating over each vertex in the graph and coloring it.

Multithreaded

For solving this problem in a multithreaded approach the problem can be subdivided into the following parts:

- ### Partitioning

  We randomly partition the vertices into k partitions where k is the number of threads, each thread takes a partition id and is responsible for coloring all its vertices.

  To randomly partition the vertices I have used two maps:

  map <int, vector<int>> partitions: Key is the partition id, value is the vector of vertices present in the partition
  map <int, int> indexing: the key is the vertex and the value is the partition it is present in.

  For partitioning:

  Shuffle the sequential vector of n values and assign each value in that sequence a partition

```
    /*Partitioning*/
    vector<int> vert;
    for (int i=0;i<n;i++)
        vert.push_back(i);
    random_shuffle(vert.begin(),vert.end());//Randomly shuffling the
vertices
    for (int i=0;i<n;)
    {
        for (int j=0;j<k;j++)
        {
            if(i<n)
            {
            //assigning partition j vertex at vert[i]
            partitions[j].push_back(vert[i]);
            indexing[vert[i]]=j;
            i++;
            }
            //all vertices have been partitioned
            else break;
        }
    }
```

Alternate Approach:
Assigning each vertex a random partition generated by rand()%k.
Drawback: won't guarantee that each partition gets some vertices, solution for this check Size of partitions[id].second() before creating a thread for that partition.

```
for(int i=0;i<n;i++)
{
int partition=rand()%k;
partitions[partition].push_back(i);
indexing[i]=partition;
}
```

Tried multiple inputs on both approaches, the performance of both was quite similar

- ## K-threads

  We create K threads each thread takes a partition id and is responsible for coloring all its vertices, the main thread waits on these k threads.
  The partition maps, graph, and color array are global.

- ## Identifying internal and boundary vertices

  Inside the thread function, the first thing to do is identify the Internal and boundary vertices, internal vertices are those whose all adjacent vertices are present in the same partition others are external/boundary vertices.
  For this, we use the indexing map as it gives O(1) access time,
  For every vertex **v** in the partition, we check if every vertex in the adjacency_list of v is in the same partition if yes then it's internal else is external which is pushed in a vector.

- ## Coloring Internal vertices

  An internal vertex can be colored easily by looking at its adjacent vertices and applying the mentioned greedy algorithm as all the vertices in its adjacency list are to be colored by the same thread with no need for synchronization with other threads.

```
    vector<int> external_vertices;

    for(int v:partitions[part])
    {
        bool flag=true;
        for(auto j:inp_graph->adj_list[v])
            if(indexing[j]!=part)
            {
                external_vertices.push_back(v);
                flag=false;
                break;
            }
        if(flag)
            color_vertex(v);
         }
```

- Coloring external vertices

Coloring external vertices is a bit tricky as we have to take care that when a thread **t** is coloring an external vertex V no other thread should be coloring a vertex V* such that V* is adjacent to V.To achieve this mutex locks can be used in ways described below:

  - Coarse Lock

    Here there is one common mutex lock **mtx** for all boundary vertices. For coloring any boundary vertex **V**, in one of its partitions, thread **T** obtains the lock on global mtx.
    Assigns a color to V in a greedy manner by looking at all its neighbors. Since T has obtained the lock on mtx, it does not have to worry about synchronization with any of the threads that are responsible for any of the V's adjacent vertices.

```
    for(int i:external_vertices)
    {
        mtx.lock();
        color_vertex(i);
        mtx.unlock();
    }
```

  -

- Fine-Grained Lock

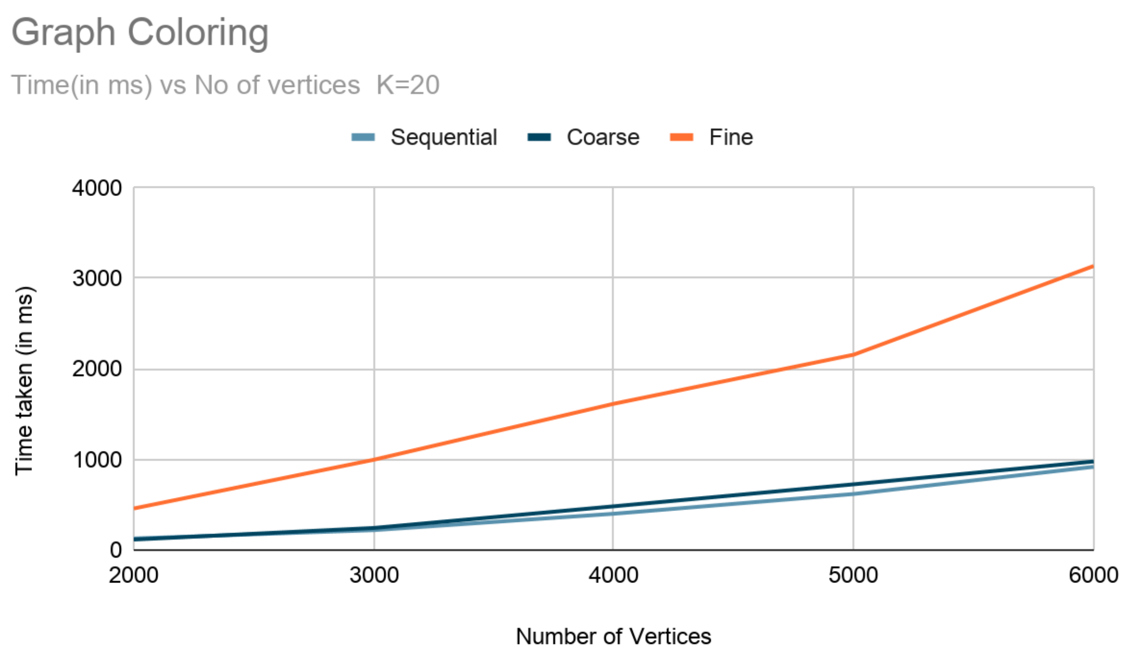
  Here we maintain a global array of mutexes mtx where mtx[i] is the lock for **i**th vertex.
  For coloring a external vertex **v** lock is obtained on all of its adjacent vertices, to avoid deadlock lock is obtained in the increasing order of vertex ids.Since in my implementation the input is taken as a adjacent matrix and stored in adjacent list formal the adjacency list of all the vertices are already sorted , we just have to obtain a lock on v in its sorted position

```
    for(int i:external_vertices)
    {
        bool inserted_flag=false;//is mtx[i] locked
        for (auto k:inp_graph->adj_list[i])
        {
            if(k>i && !inserted_flag)
            {
                mtx[i].lock();//lock mtx[i]
                inserted_flag=true;
            }
            mtx[k].lock();
        }
        //mtx[i] is not locked ...ie i is greater than the last element of its adjacency list
        if(!inserted_flag) mtx[i].lock() ;

        color_vertex(i);//color vertex i

        //Unlock all acquired locks
        mtx[i].unlock();
        for(auto k:inp_graph->adj_list[i])
            mtx[k].unlock();
    }
```

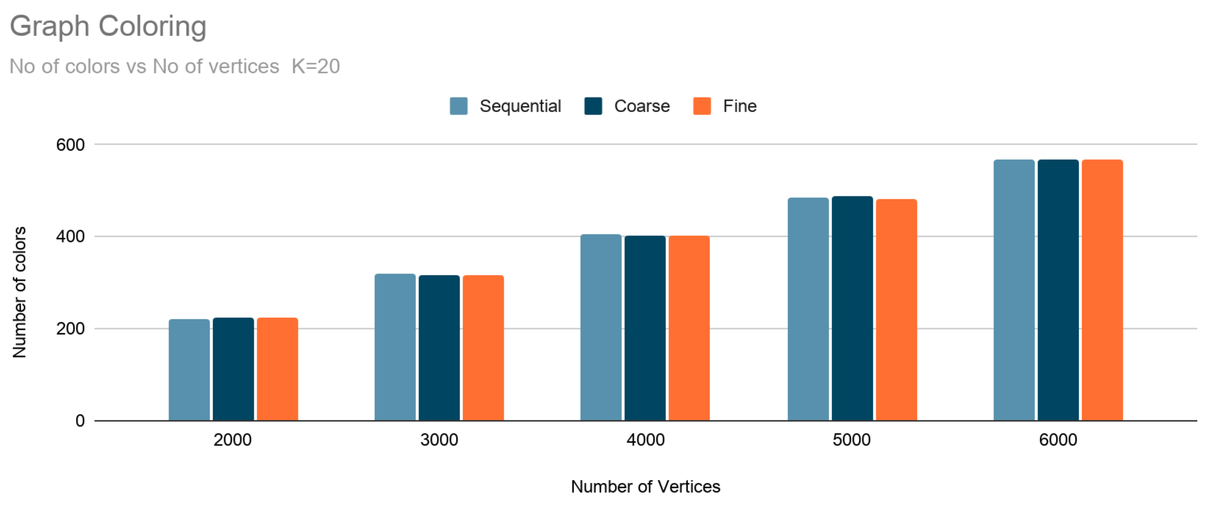
  After the vertex is colored we release all the acquired locks.

Comparisons

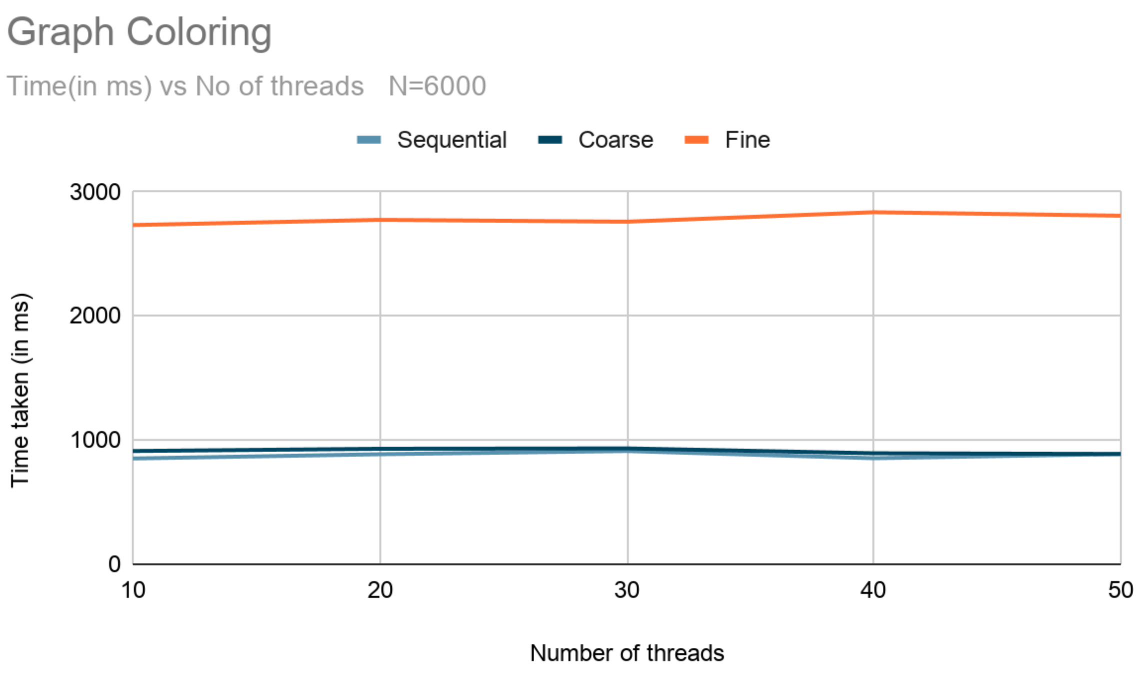
Graph-1



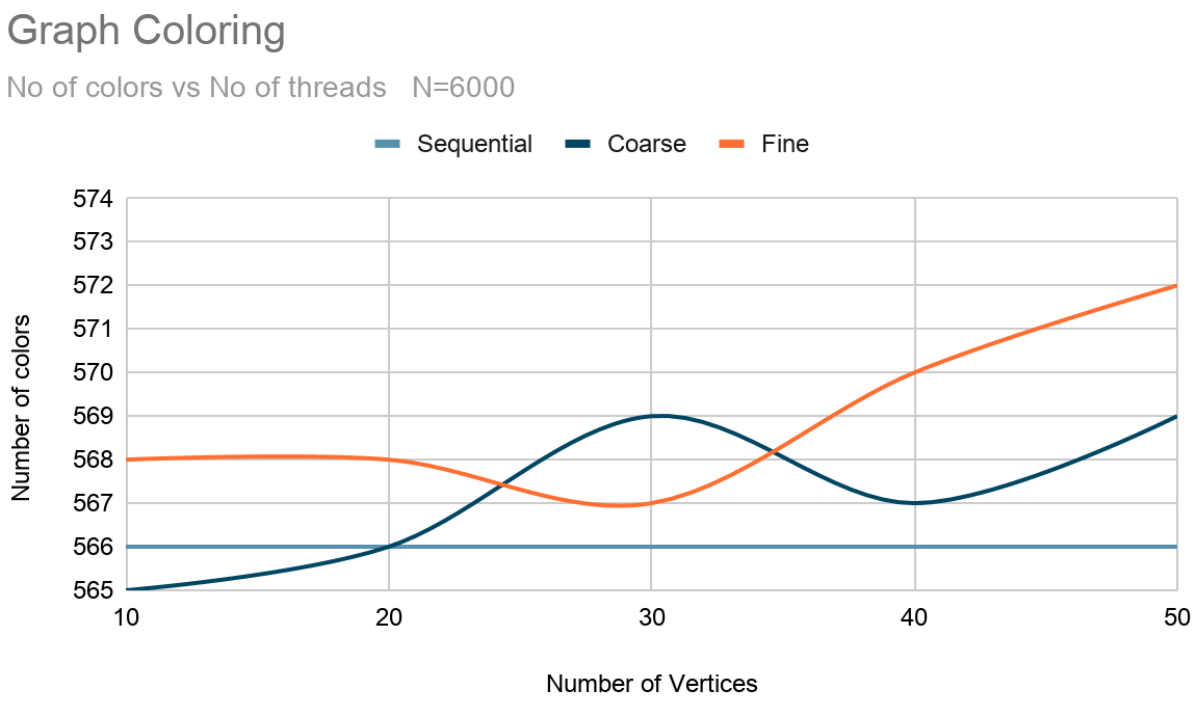
Graph-2



- Graph-3



- Graph-4



Analysis

For comparison of the 3 implementations i have plotted out the above graphs on analyzing the plots:

- Graph-1 and 3: as we increase the number of vertices the time taken by each method increases,but fine_grained takes much more time than others and coarse grained and sequential are comparable but coarse takes a little more time than sequential.
    - On analyzing i found out that in coarse grained and fine_grained the partitions almost always have negligible internal vertices hence in coarse grained graph when we acquire lock for coloring the vertices essentially its acting as sequential only as we are not able to utilize the multithreading in internal vertices + the overhead of thread creation and switching + lock acquiring hence time is not better than sequential
    - In fine-grained we should have seen better performance as at a time only some external vertices are locked other can be colored but since the input graph is relatively small the overhead caused by thread creation + locking and unlocking vertices must be much greater than the speedup provided by the multithreading also the edges in the random generated graph might be too entangled to provide much speedup in such a small graph.

  In graph-3 increase in the number of threads is not causing significant improvement as the graph is still very small so the above mentioned reasoning applies .
- Graph-2 and Graph-4: No of colors are erratic as the partitioning and scheduling of threads is random so no definite pattern can be observed , as seen in graph-2 in some cases fine-grained gives least colors in some coarse and in some sequential ,no definite pattern was observed.

Conclusion

Due to many constraints(Input files were becoming very large, taking too much time to write and computations were causing heating of the machine and the number of physical cpu threads on my system is very less), I was able to test the methods on small inputs (Order of thousands) only where sequential algorithm was performing better but on very large graphs(maybe practical graph data where clusters can be identified and some sort of smart partitioning can be done on basis of heuristics), high powered machines consisting of many physical threads on the cpu the fine-grained algorithm would perform much better than the sequential algorithm and there the power of multithreading would be realized.