****Report for 'IMPLEMENTING GRAPH COLORING ALGORITHM USING LOCKS':**

Design of the program:

1. After creating the cpp file, starting writing the code by including necessary header files such as "iostream", "fstream" etc.,

Here, we should include 'semaphore.h' to use semaphores for locks and 'vector' to use vectors for storing partitions of the graph.

2. Then declare necessary functions like

- present (function to search whether a integer is present in a vector (of int) or not),
- Type(function for knowing whether a vertex is internal or external in its partition),
- threadFunc (thread function for graph coloring with multithreading using coarse grain locking)
- threadFunc2 (thread function for graph coloring with multithreading using fine grain locking)

3. Also declare global variables:

- Mat(int**) – for Adjacency matrix
- type(bool*) – for knowing whether a vertex is internal or external in its partition

  ['0' implies internal; '1' implies external]

- colour(int*) – for knowing colours of each vertex ['0' implies uncoloured]
- Max – to know number of colours used
- V- Total number of vertices
- Also, semaphores mutex(sem_t), Mutex(sem_t*) are declared globally for mutex locks.

4. Now, in the main function, using a file pointer Infile declared with ifstream, open the file "input_params.txt" and read the input for P,V. [P is number of partitions]

["input_params.txt" contains values of P,V in first line, from next line onwards, adjacency matrix for the graph is present.

Ex: 2 5

0 0 0 1 0

0 0 0 1 1

0 0 0 0 1

1 1 0 0 1

0 1 1 1 0]

Also declare the file pointer 'Outfile' with ofstream to print the output into 'output.txt'.

5.Now, dynamically allocate memory for the globally declared variables.

6. Then, with the help of for loop, for every vertex randomly choose a partition, store the adjacency matrix from input file into ‘Mat’ and also uncolour all the vertices.

7. With the help of function ‘Type’, decide whether a vertex is internal or external in its corresponding partition.

8. Now, initialise the semaphores ‘mutex’ and array of semaphores ‘Mutex’ to ‘1’.

9. Then, with the help of for loop, create a thread for each partition, with ‘threadFunc’ and later, join the threads.

10. But make sure to note the time with the help of gettimeofday function, before thread creation and after threads execution.

(For COARSE GRAIN)

11.

In threadFunc:(Graph colouring implementation using coarse grain locking)

- Initially, declare an array of bool– Arr , and initialise each value to ‘true’.
  This Arr, helps to know, which colours are available and unavailable for a vertex basing on the colours of its adjacent vertices.
- Firstly, access each vertex in the partition with the help of for loop.
- If the vertex is internal vertex, then make the colours present in its adjacent vertices in Arr to unavailable (i.e, false).
  Then, colour the present vertex with least available colour ID.
  Later, again, make the colours present in its adjacent vertices in Arr to available (i.e, true).
- But if the vertex is external vertex, the vertex will be having adjacent vertices outside its partition, so, we use locks preventing other threads to access at the same time.
  Hence, the same procedure is followed like internal vertex, unlike now, initially sem_wait(&mutex) is used for locking and at the end, sem_post(&mutex) is used for unlocking.

The threadFunc ends if all the vertices in the partition are coloured.

12. Now, printing the output into ‘output.txt’ for coarse grain locking algorithm i.e, number of colours used, timetaken(in milliseconds) by algorithm and corresponding colours used for each vertex.

(For FINE GRAIN)

13. Now, in ‘threadFunc2’:(Graph colouring implementation using fine grain locking)

It follows same as coarse grain locking, unlike now, instead of using a single lock (if vertex is external), now, we use multiple locks.

That is, if a vertex is external, then initially, with the help of for loop, all the locks of the adjacent vertices including itself are locked(sem_wait(&Mutex[i])) in increasing order of vertex IDs so as to prevent any deadlock.

Then after colouring the vertex, the locks of adjacent vertices including itself are unlocked(sem_post(&Mutex[i])) in the same order i.e, increasing order of vertex IDs.

14. Now, again, printing the output into 'output.txt' for fine grain locking algorithm i.e, number of colours used, timetaken by algorithm(in milliseconds) and corresponding colours used for each vertex.

15. Finally, define the above declared functions i.e, present,Type,threadFunc or threadFunc2.

GRAPHS:

The following below are the four graphs which are generated:

Graph1: showing the "Number of Vertices Vs Time taken by the algorithm(in millisec)".

Here, the number of Partitions is fixed to 25.

Graph2: showing the "Number of Vertices Vs Number of colours used".

Here, the number of Partitions is fixed to 25.

Graph3: showing the "Number of Partitions Vs Time taken by the algorithm(in millisec)".

Here, the number of Vertices is fixed to 1000.

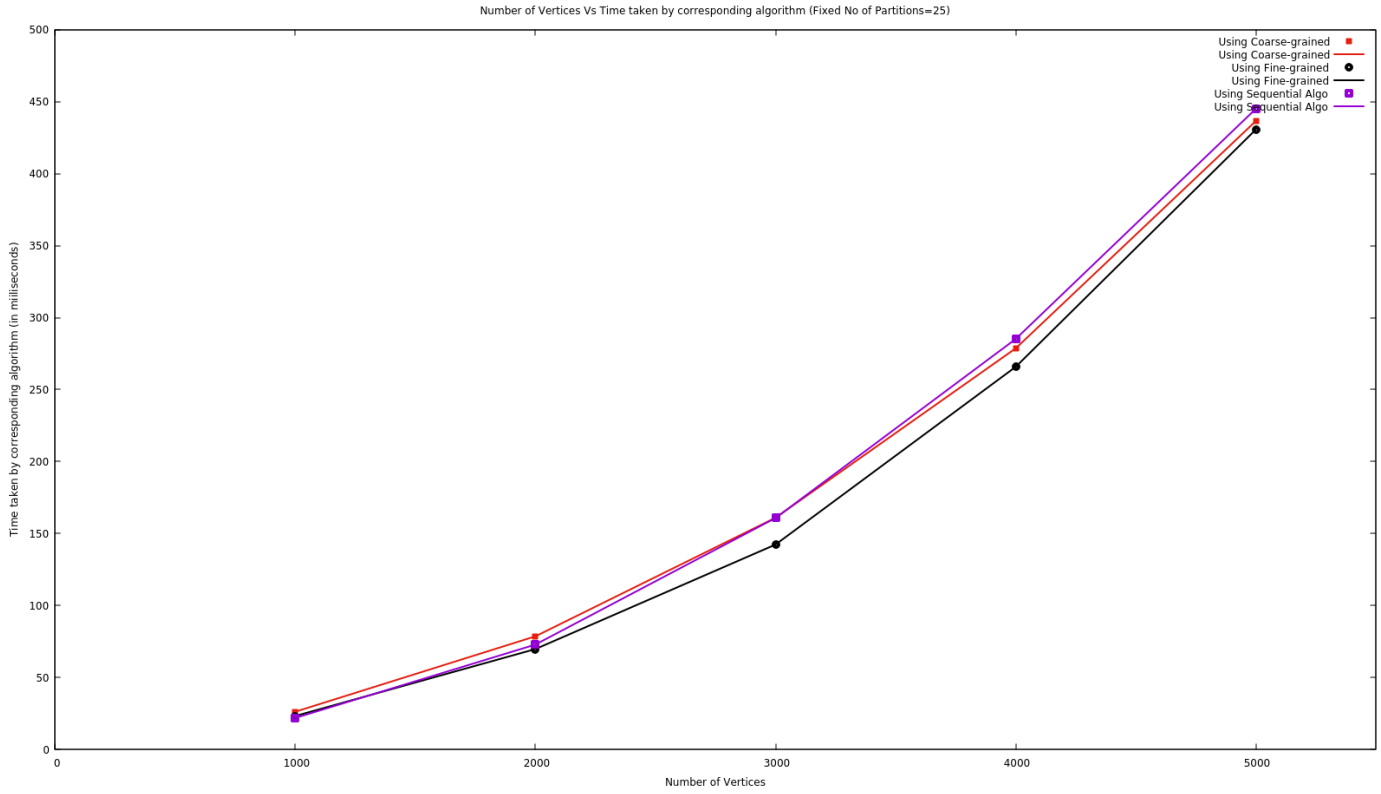
Graph4: showing the "Number of Partitions Vs Number of colours used".

Here, the number of Vertices is fixed to 1000.

Basing on the results displayed in the graphs, a small analysis is made on the output.

(Here, sometimes the output values are varying for different laptops)

GRAPH1: "Number of Vertices Vs Time taken by the corresponding algorithm (in millisec)" (No of partitions =25 (fixed))



For generating this graph, **number of partitions are fixed to 25** and Number of Vertices are varying from 1*10^3 to 5*10^3 are taken on x-axis and corresponding time taken (in milliseconds) are taken on y-axis. The graph is plotted using the three algorithms (coarse-grained, fine-grained, sequential algo).

Analysis:

Clearly from the graph, as the number of vertices increase, time taken by the algorithms also increase.

More importantly,

For number of vertices below 2500:

Time taken by Fine-grain < Time taken by Sequential algo < Time-taken by Coarse-grain.

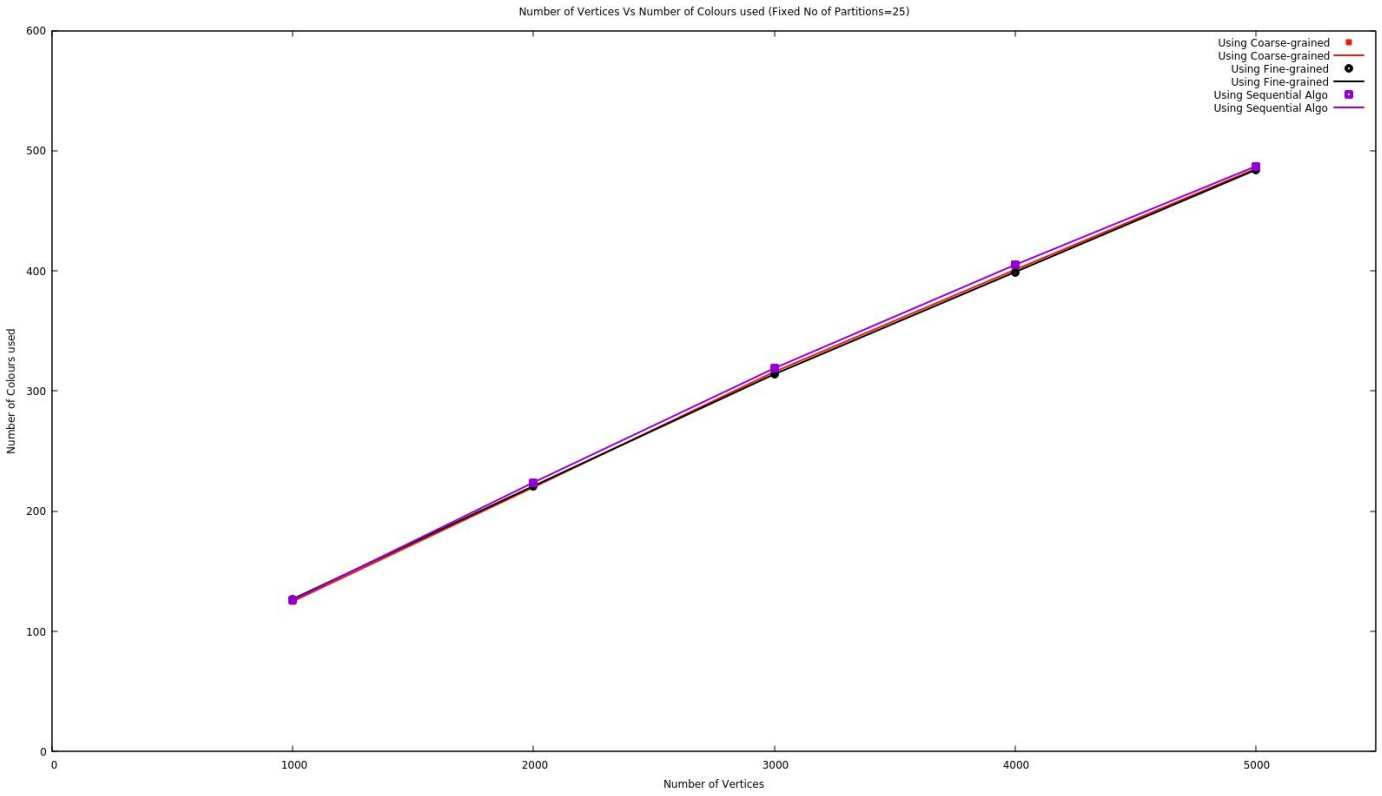
For number of vertices above 2500:

Time taken by Fine-grain < Time-taken by Coarse-grain < Time taken by Sequential algo.

Hence**, Performance**(as number of vertices increase):

Fine-grain > Coarse-grain > Sequential-algo.

GRAPH2: "Number of Vertices Vs Number of colours used".(No of partitions=25(fixed)).



For generating this graph, **number of partitions are fixed to 25** and Number of Vertices are varying from 1*10^3 to 5*10^3 are taken on x-axis and corresponding Number of colours used are taken on y-axis. The graph is plotted using the three algorithms (coarse-grained, fine-grained, sequential algo).

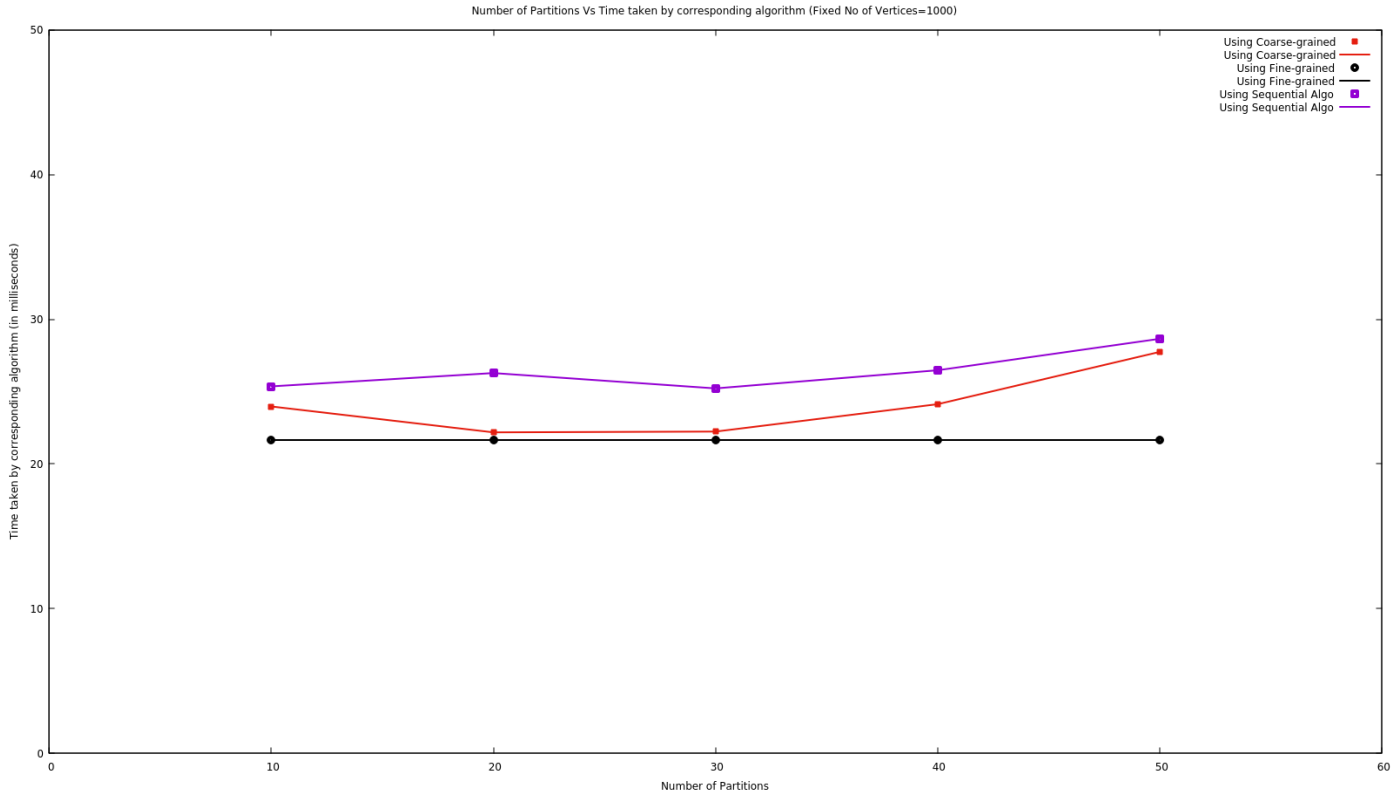
Analysis:

Clearly, from the above graph, as the number of vertices increase, number of colours used also increase for the three algorithms and moreover, they are almost overlapping each other which implies, the number of colours used by the three algorithms is almost same.

- Number of colours used by the 3 algorithms increase with in increase in number of Vertices.
- Number of colours used by the 3 algorithms for same number of vertices at any time:

  Coarse-grained ~ Fine-grained ~ Sequential Algorithm (almost same)

GRAPH3: "Number of Partitions Vs Time taken by the algorithm (in millisec)" (No of threads = 1000(fixed)).



For generating this graph, **number of vertices are fixed to 1000** and Number of Partitions are varying from 10 to 50 are taken on x-axis and corresponding time taken (in milliseconds) is taken on y-axis. The graph is plotted using the three algorithms (coarse-grained, fine-grained, sequential algo).

Analysis:

Clearly from the graph, as the number of Partitions increase, time taken by the algorithms is almost same.

More importantly,

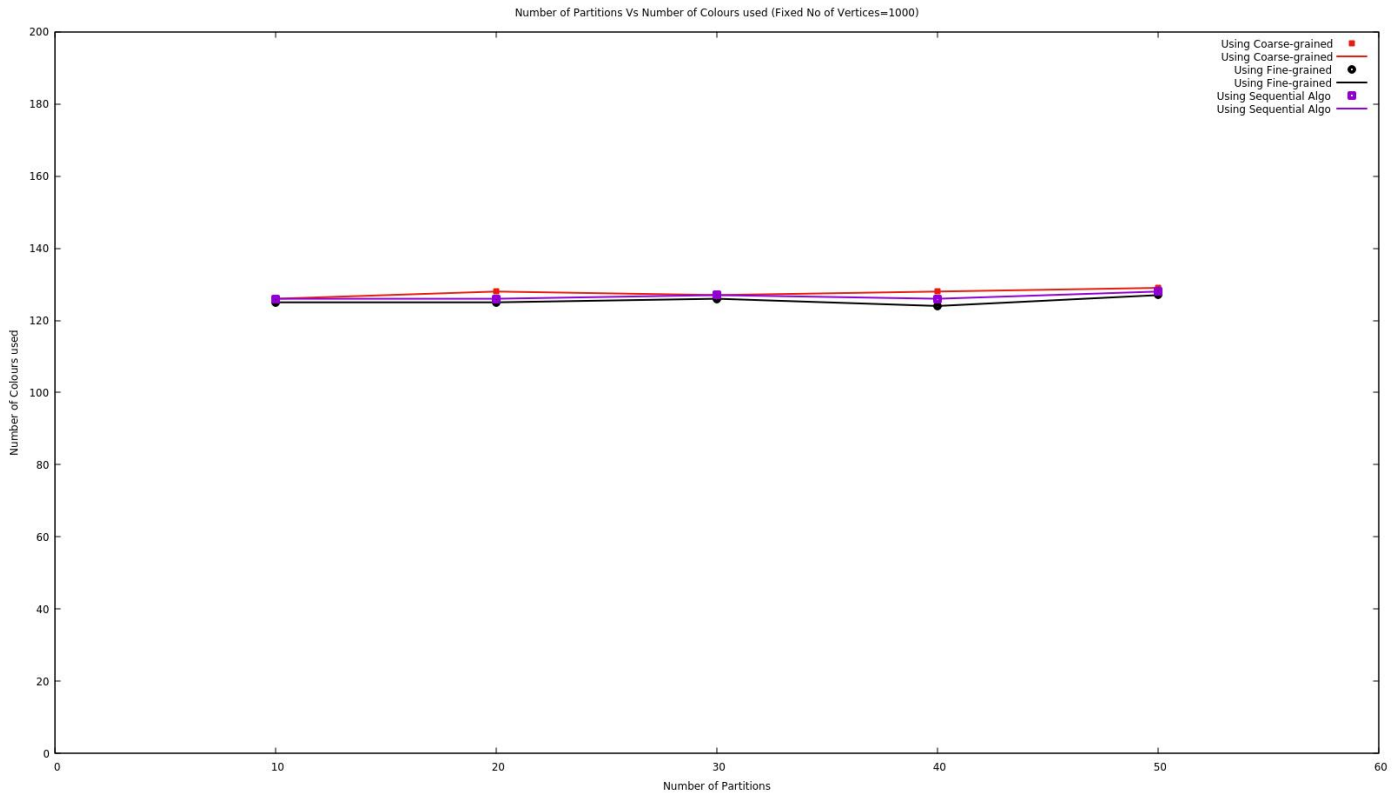
For same number of partitions :

Time taken by Fine-grain < Time-taken by Coarse-grain < Time taken by Sequential algo.

Hence, **Performance** (as number of Partitions increase):

Fine-grain > Coarse-grain > Sequential-algo.

GRAPH4: "Number of Partitions Vs Number of colours used" (No of threads = 1000(fixed)).



For generating this graph, **number of vertices are fixed to 1000** and Number of Partitions are varying from 10 to 50 are taken on x-axis and corresponding Number of colours used are taken on y-axis. The graph is plotted using the three algorithms (coarse-grained, fine-grained, sequential algo).

Analysis:

Clearly, from the above graph, as the number of partitions increase, number of colours used are always approximately same which implies, that with increase in number of partitions, number of colours used won't change and moreover, they are almost overlapping each other which implies, the number of colours used by the three algorithms is almost same.

- Number of colours used by the 3 algorithms is almost same with increase in number of Threads.
- Number of colours used by the 3 algorithms for same number of partitions at any time:

  Coarse-grained ~ Fine-grained ~ Sequential Algorithm (almost same)