OS-II

CS3523

CS19BTECH11039
Sri Hari

Programming Assignment 5
Report

Aim: To derive a conclusion for graph colouring by comparing the coarse-grained, fine-grained and sequential algorithms.

Randomising vertices: A vector of vertices is taken and randomised. Further the split method in the class splits the vector into a number of parts based on the number of threads.

Sequential(Greedy algorithm):
Design:

The algorithm works in a greedy manner, checks each vertex and explores its neighbours. If a neighbour is already coloured, we insert it into the set colours. By the end of the algorithm run, each vertex is explored, each neighbour of the vertex is explored sequentially and hence we colour all the graph vertices.

Implementation:

1. Iterate over the vertex, i.e extract a vertex from the vertices of the graph.

2. Create a set data structure for storing colours of neighbours.
3. Explore every neighbour of the vertex, if it is already coloured then insert it into the set. This ensures that the colours are stored in increasing order.
4. Once, all the neighbours are explored, now check in the set for the least possible colour to assign to the present vertex.
5. All the vertices get coloured after each vertex is extracted.

Coarse-Grained:

Design:

This method involves one common lock for the external vertices. A class method splits the vertices into random partitions and then the coarse-grained method colours the vertices in the following way. Split the vertices into 2 sets, internal and external. Internal need not have any thread synchronization but **Global variables** should be accessed and modified very carefully. For every global variable, we have a local copy and is modified carefully. For external vertices, add a lock before exploring the neighbours and unlock after colouring the neighbour. The internal colouring mechanism is the same as the Greedy colouring algorithm.

Implementation:

1. Divide the partition's vertices into 2 sets, namely internal and external vertices.
2. Internal vertices need not have any synchronization, follow the greedy colouring algorithm for the colouring of

internal vertices. All the threads simultaneously execute this.
3. External vertices need to have synchronization. Extract a vertex from external vertices and apply a lock.
4. Follow the greedy algorithm for colouring the vertex.
5. After colouring the vertex, unlock the previously applied lock such that other vertex or other vertices from other threads acquire the lock and colour the subsequent vertex.
6. Iterate over the vertices, until all are coloured.

Fine-Grained:

Design:

This method too involves the locking mechanism but unlike a common lock in the case of the coarse-grained method, here we have locks for every single vertex. A class method splits the vertices into random partitions. The internal vertices colouring happens to be the same as coarse-grained but the implementation of external vertices. Every time a vertex is extracted from external vertices, lock all the neighbours of the vertex including the vertex and colour the vertex. After colouring, unlock all the neighbour locks and the self-lock.

Implementation:

1. Divide the partition's vertices into 2 sets, namely internal and external vertices.
2. Internal vertices need not have any synchronization, follow the greedy colouring algorithm for the colouring of internal vertices. All the threads simultaneously execute this.
3. External vertices are to be properly synchronized.

4. Extract a vertex from external vertices and apply a lock to all its neighbours and itself.
5. Colour the vertex using the greedy algorithm as mentioned above.
6. After colouring the vertex, release all the neighbour locks and self-lock.
7. Iterate over the vertices until all are coloured.

Note: All the pthread attributes are properly initialised and properly destroyed before the termination of the program. Memory management is done, every allocated memory is deallocated at the end of the program.

Difficulties faced in implementation:

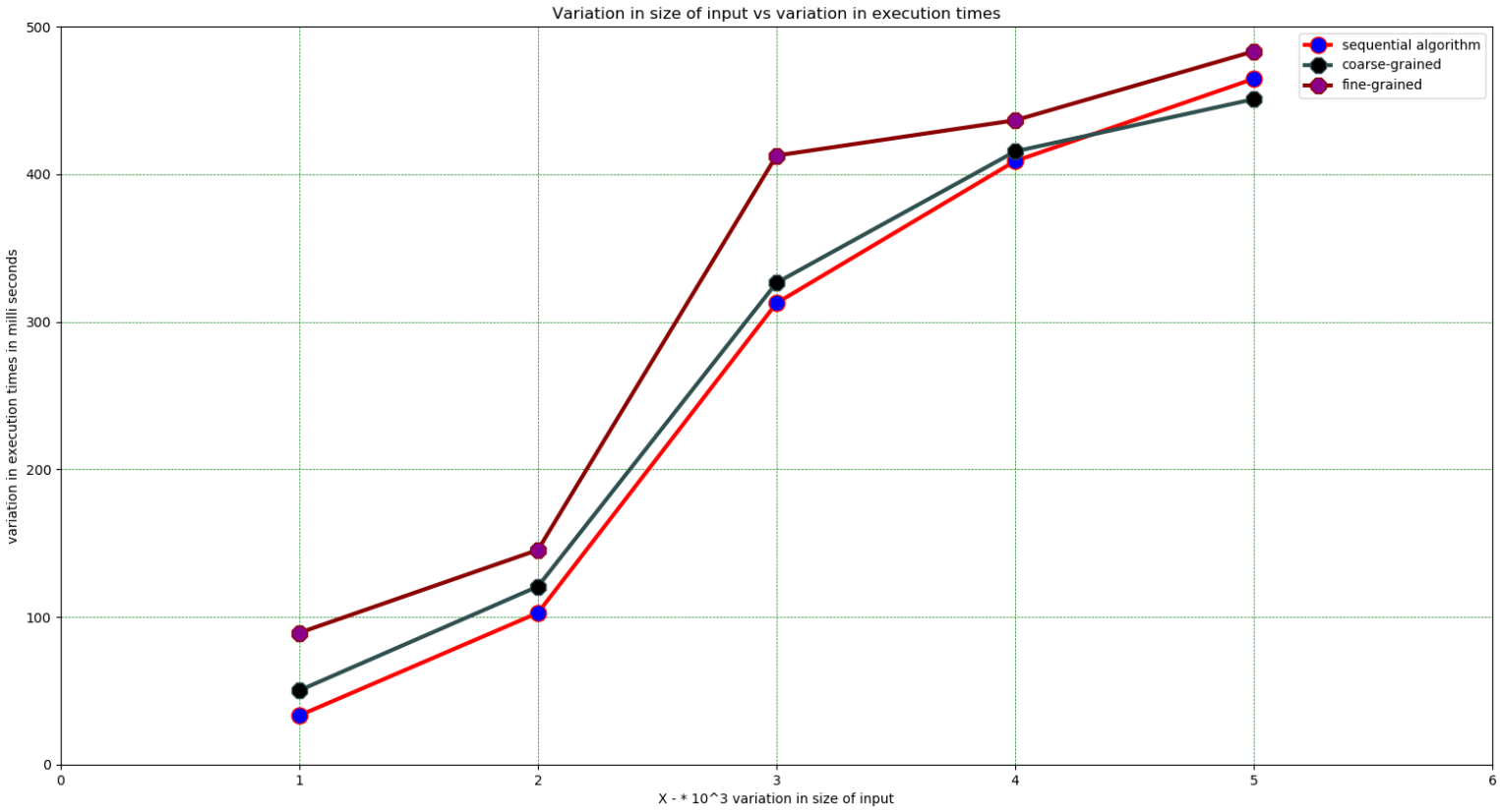
- There existed a few situations where the program ran into a deadlock situation.
- Debugging the deadlock situation.
- Improper synchronization of vertices leads to false colouring in fine-grained implementation.
- Improper modification of the global variables, i.e race conditions in some cases.

After thoroughly debugging the code, all the problems were corrected.

Note: The graphs were observed for the size in the order of 10^3 and for the order of 10^4 the machine showed the error "Killed". Moreover, each machine is giving different outputs for the same executable(checked 3 machines) and the results mentioned were optimal out of the 3. Every element of the graph was recorded 100 times and then averaged for getting a better result (used the shell script to achieve this).

Graphs and observations:

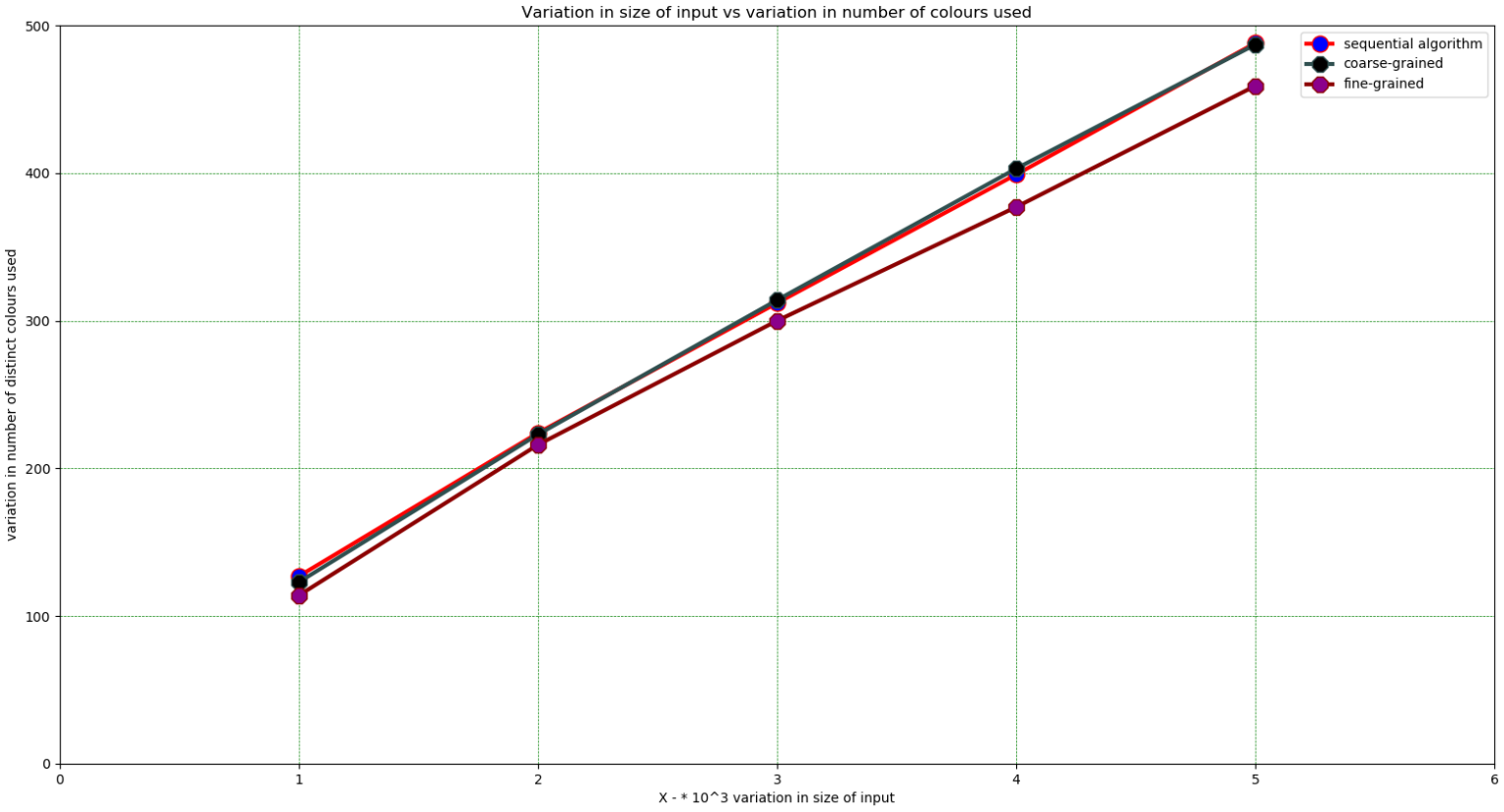
- **Variation in input vs variation in execution times:**



Clearly, in the graph, we can see that the fine-grained algorithm is taking some extra amount of time, than the sequential algorithm and coarse-grained algorithm which are almost equal. Initially, as the graph contains 1000 vertices, coarse-grained is taking more time to execute than the sequential and fine-grained is taking more time than the coarse-grained. Later as the size of the input increases, the **slopes** are clearly decreasing than that of the sequential algorithm. This implies that the coarse-grained algorithm and fine-grained algorithm are supposed to perform better than the sequential algorithm. But lack of computation ability of the machine led to this graph. However, it is clear that the

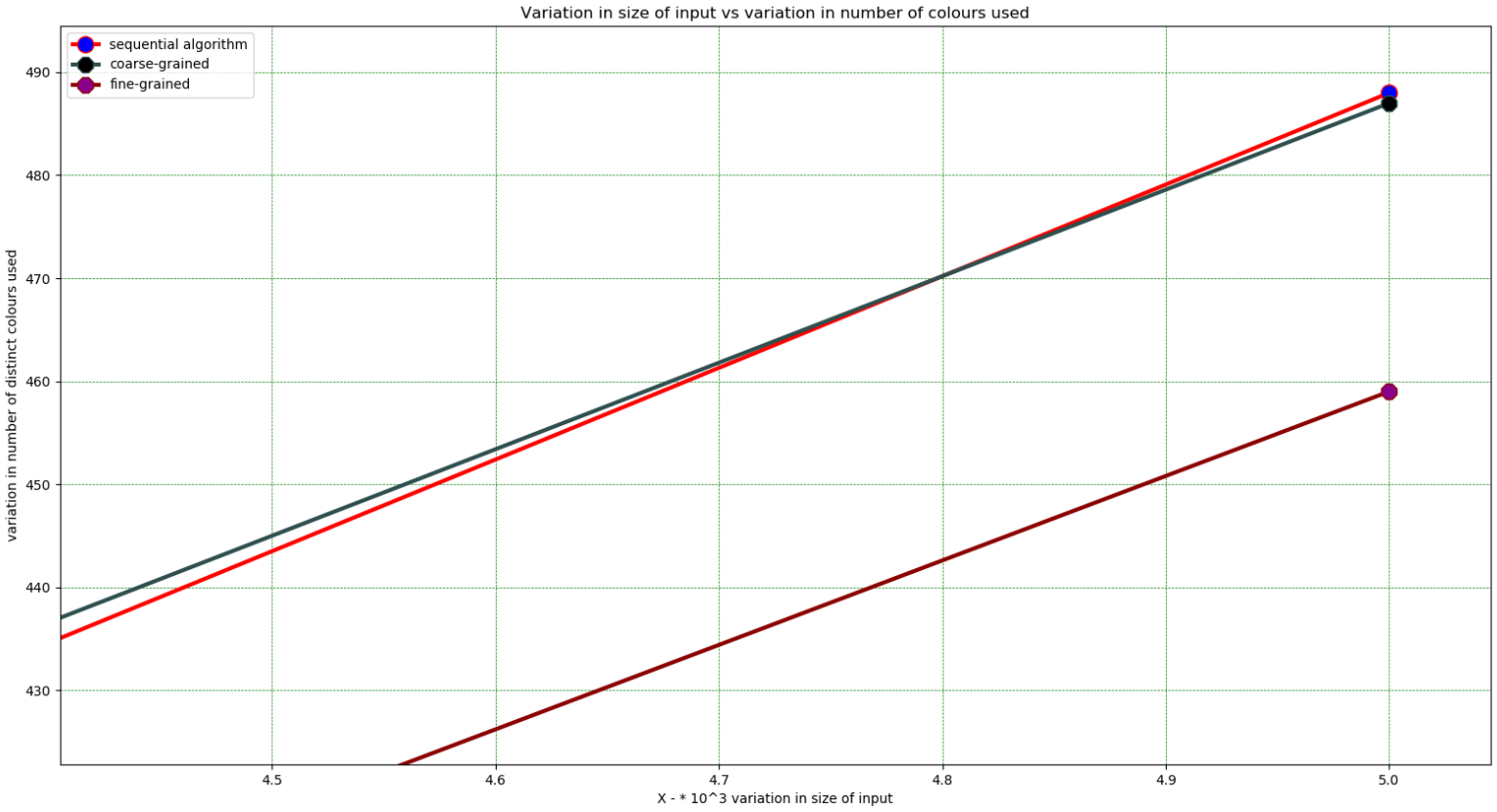
sequential algorithm is better for small inputs and the rest two are supposed to perform better than the sequential algorithm for large inputs.

- **Variation in size of input vs variation in the number of colours used:**



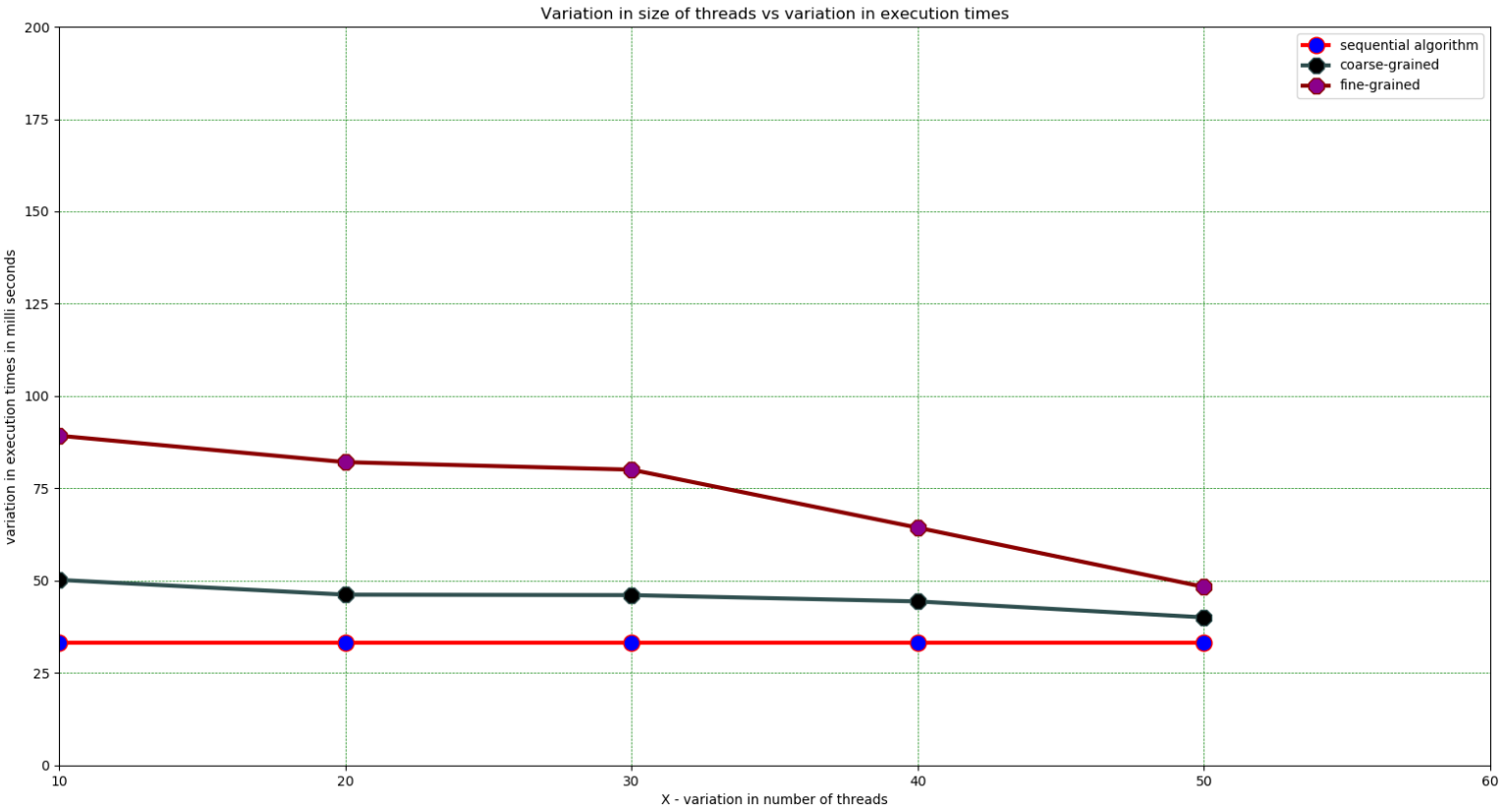
The graph shows that initially for the small inputs almost all the algorithms are taking an equal number of colours to colour the entire graph. But later on with the increase in the number of input vertices, the fine-grained algorithm is taking particularly less distinct colours to colour the very same graph. We can clearly see the deviation of the fine-grained algorithm is taking a diversion from the other two algorithms. So, fine-grained guarantees a less number of distinct colours almost all the time and now we need to

differentiate between the sequential algorithm and coarse-grained algorithm. For that let's see the zoomed pic of the above picture.



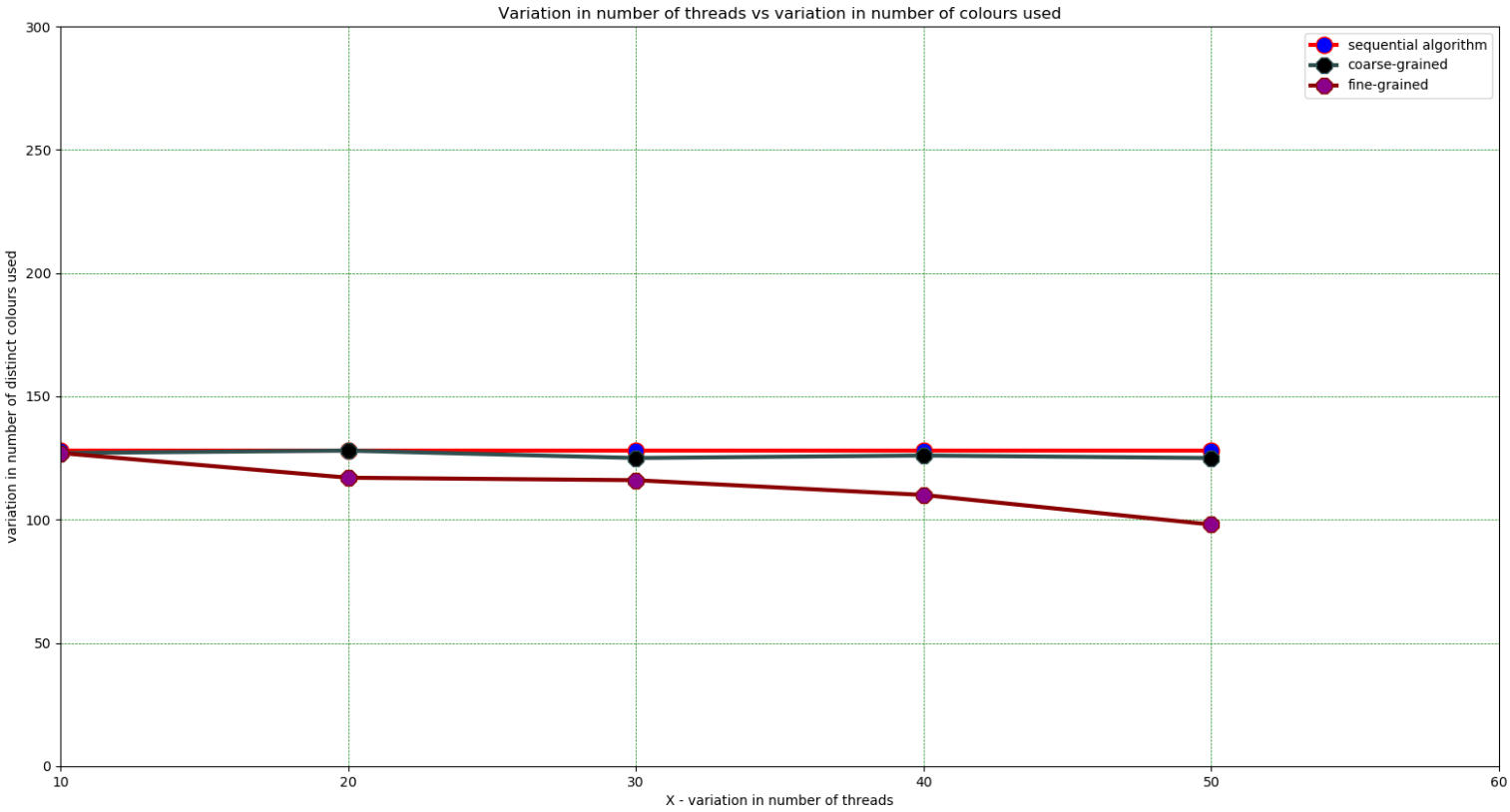
This clearly helps us to derive a conclusion that the coarse-grained algorithm curve is having less slope as the increase in the number of vertices. So in the end, for a reasonably large input size, the number of colours used follows the order fine-grained < coarse-grained < sequential algorithm. The computation of higher-order is a creating problem and is getting killed, thus creating a situation to conclude with the results obtained.

- **Variation in the number of threads vs variation in execution time:**

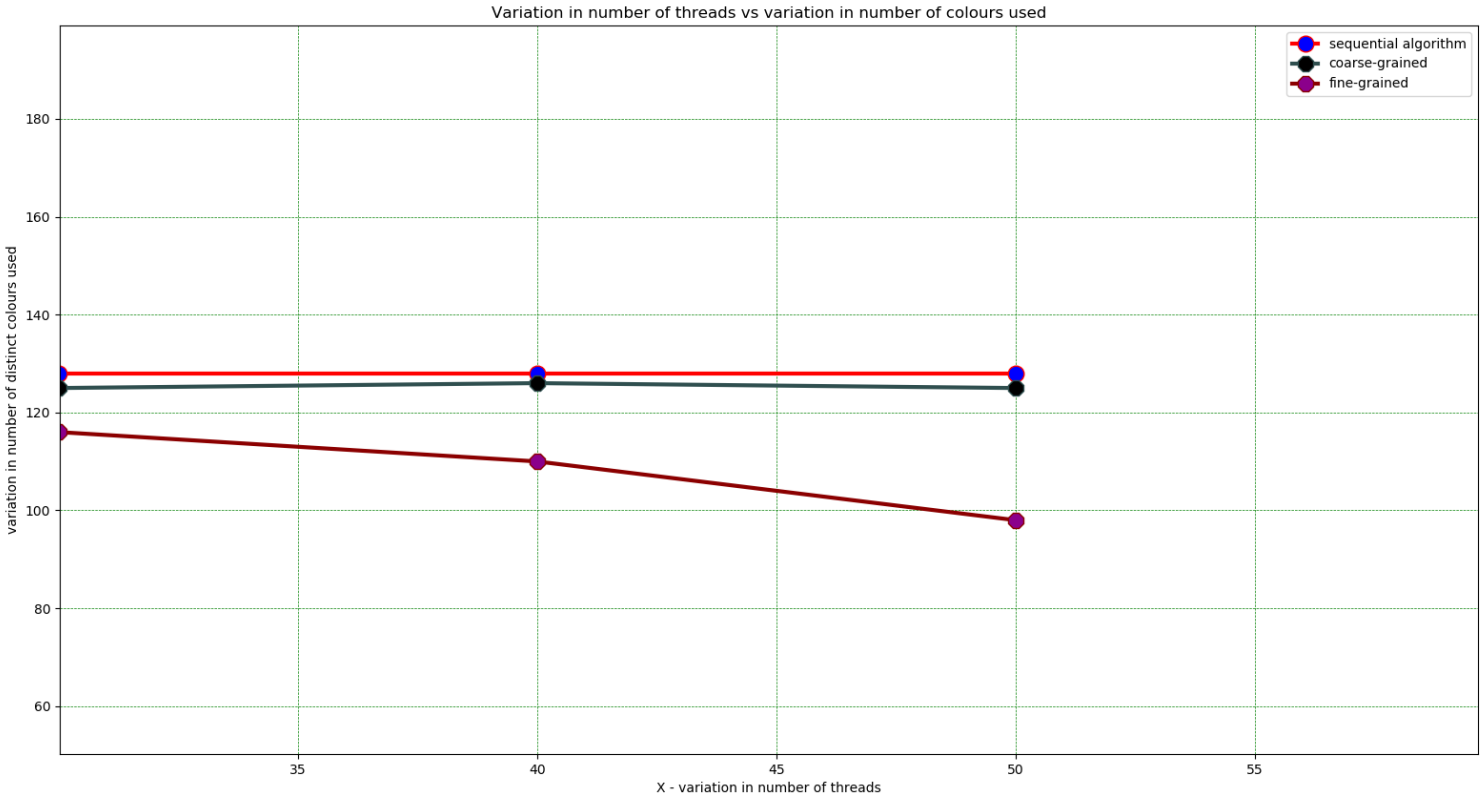


This graph shows that, with the increase in the number of threads for the fixed input of 1000, the time taken by coarse-grained and fine-grained are decreasing. Initially, the algorithms are taking more time than that of the sequential algorithm but with the increasing number of threads, the time taken by coarse-grained and fine-grained are decreasing in a rapid fashion that, for an increase in furthermore number of threads, the time taken by the coarse-grained and fine-grained algorithms are supposed to be less than the traditional sequential algorithm. The initial scenario might be because of the overhead of the threads created.

- **Variation in the number of threads vs variation in the number of colours used:**



From the graph, one can easily see that with an increase in the number of threads, the fine-grained algorithm is taking significantly less colours to colour all the graph vertices than the both sequential and coarse-grained algorithms. Initially for fewer numbers of threads, all the algorithms are almost starting with the same number of colours and then getting diverged. The divergence for the fine-grained algorithm is clearly evident but there is a neck to neck tie for the coarse-grained algorithm and sequential algorithm. Clearly the fine grained algorithm wins here and let's see the zoomed pic to get a proper understanding and conclusion.



This shows that with the increase in the number of threads, the coarse-grained algorithm is taking less number of colours than the sequential.

Final Conclusions:

As from the graphs, both coarse and fine grained algorithms are expected to outperform the sequential algorithm in large amounts of input vertices and large number of threads. Now in between the coarse-grained algorithm and fine-grained algorithm fine-grained algorithm is expected to perform better because of the extreme dips in the slopes when compared to the coarse-grained algorithms.

Note:

Input for the source codes:

The inputs for the source codes are followed exactly as in the pdf given.
Sample given:

2 5
1 2 3 4 5
1 0 1 1 0 0
2 1 0 1 1 0
3 1 1 0 1 0
4 0 1 1 0 1
5 0 0 0 1 0

The numbers 1 2 .. 5 are scanned and left.
The **generator** file is attached in the Zip file to create inputs in this format.

THE END!