REPORT

Implementing Graph Coloring Algorithm using locks

1. Design and Algorithm implementation details

C++ is used for implementing graph colouring algorithm using threads and locks. The “semaphore.h” library is used for implementing binary semaphores/locks. Earlier, int mutex was being used but I was running into syhcronisation/deadlock issues, so a switch was made. “std::thread” is used for creating threads and implementing multithreading in the program. The representation chosen for the graph is “adjacency list” due to it ease of use and understanding. A list has been used to implement this. The “createGraph” function is used to resize the list after obtaining parameter “v”, which is the number of vertices. A temporary vector is used for taking input from input_params.txt, which is later copied into the list via “addEdge” function. Many vectors are used for the algorithmic implementation, an “intern” vector for keeping track of which vertices are internal and which are external, a “result” vector for keeping track of colours assigned to the v vertices, a “tempo” vector for checking what colours are being used by adjacent/neighbour vertices as well as a “partition” vector for randomly assigning vertices to different threads, this random assigning is done using “rand” function. The “srand(0)” function is used to make sure the randomly generated partitions are unique. Time is calculated using functions from the chrono library.

The entire implementation assumes vertices are numbered from 0 to v-1, and colours also start from 0, but in the final output, 1 is added to each vertex and colour for ease of understanding and to comply to output needs. For greedy colouring, first we check whether a vertex belongs to the

partition of the running thread or not. If it doesn't, the vertex is skipped and will be checked by another thread. If it is, we further check if the vertex is internal or not. If it is, then greedy colouring occurs without conflict. We traverse the list of each vertex and check if the neighbours are coloured. Note of the colours used is taken. After that, the smallest available colour is assigned to the vertex and the values are reset. If the vertex is external, we use one lock mutex(CL) to synchronise with other threads in the case of coarse locking. The lock is obtained for the vertex to be coloured only and steps similar to the ones used for internal vertex colouring are used. Its advantages include simple implementation and easy verification of mutual exclusion.

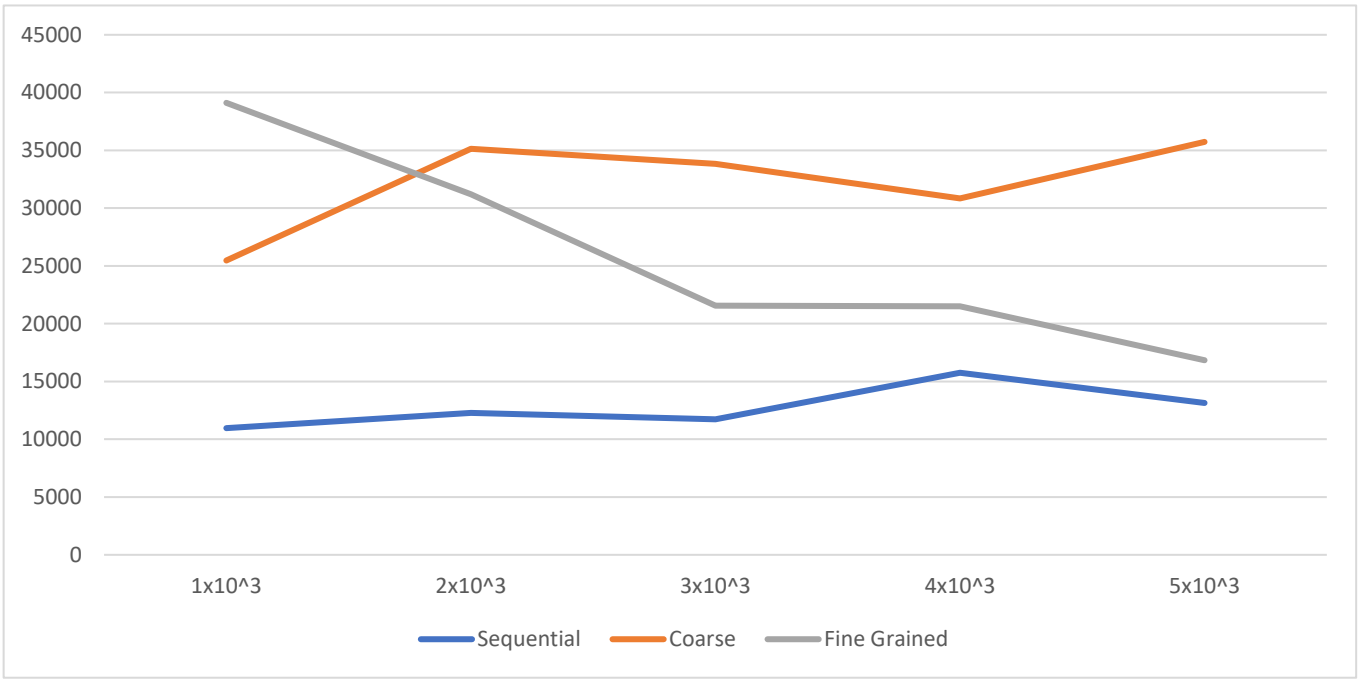
For fine-grained locking, the implementation was a lot more trickier. Since each vertex gets its own lock, an array of locks was created and initialised to 1. Sorting is also done on each list before greedy colouring begins. Since the list only contains the neighbours of vertex "i" and not vertex "i" itself, to ensure that locks are acquired in ascending order of ids for deadlock prevention, a "checker" variable is used for acquiring lock for the vertex to be coloured. This implementation proved to be a headache as improper implementation led to deadlocks, and program indefinitely pausing execution. However, with a lot of brainstorming, implementation was fixed, locks were acquired successfully without deadlocks and program was working fine. The colouring is done in a similar manner to coarse locking and in the end, the locks are released.

The number of vertices are varied from 1x10^3 to 5x10^3 for ease of use and for easy random generation of graph.

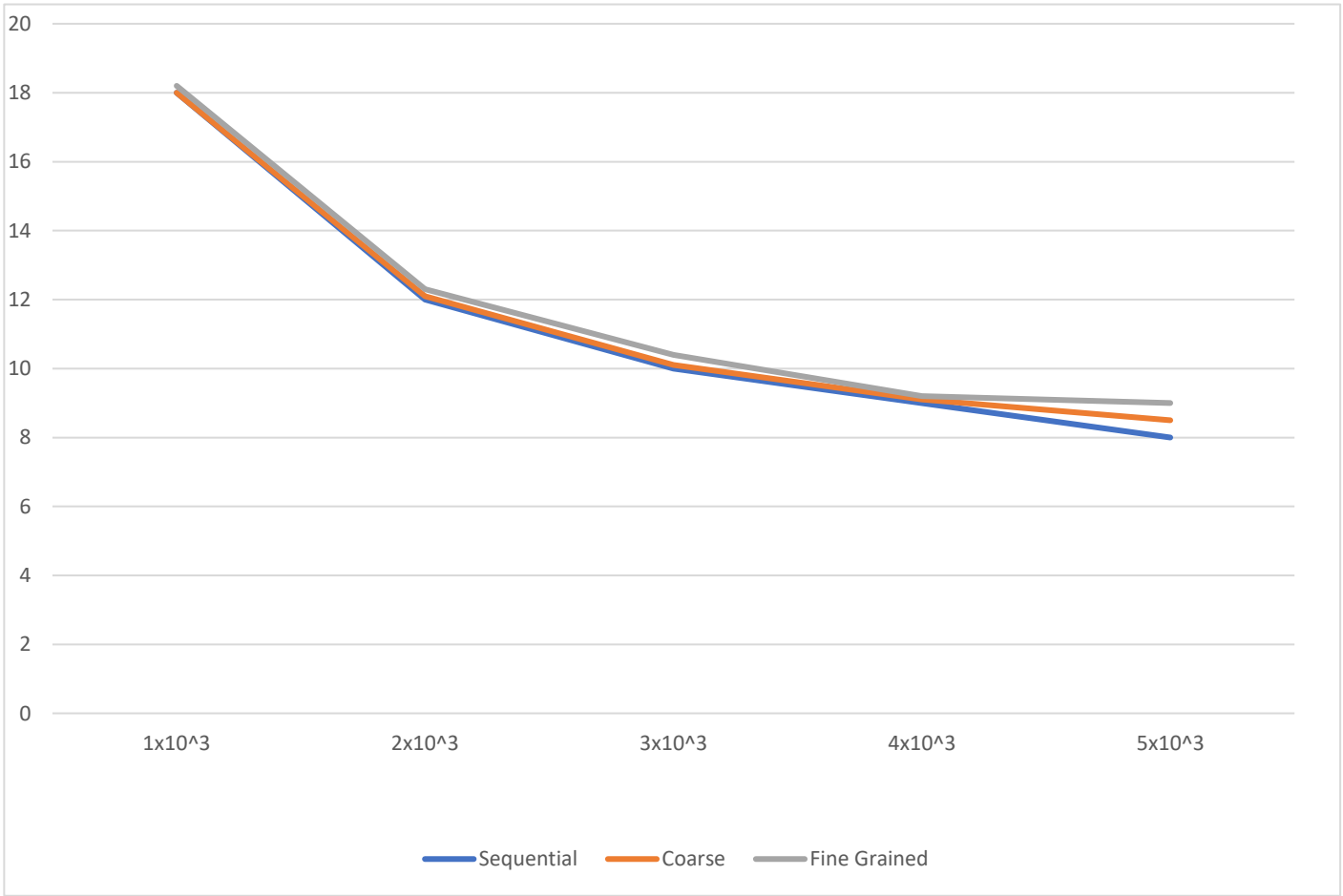
A sequential algorithm is used for comparisons as well, which is based on the same algorithm for colouring as the previous two, without the need for threads or locks.

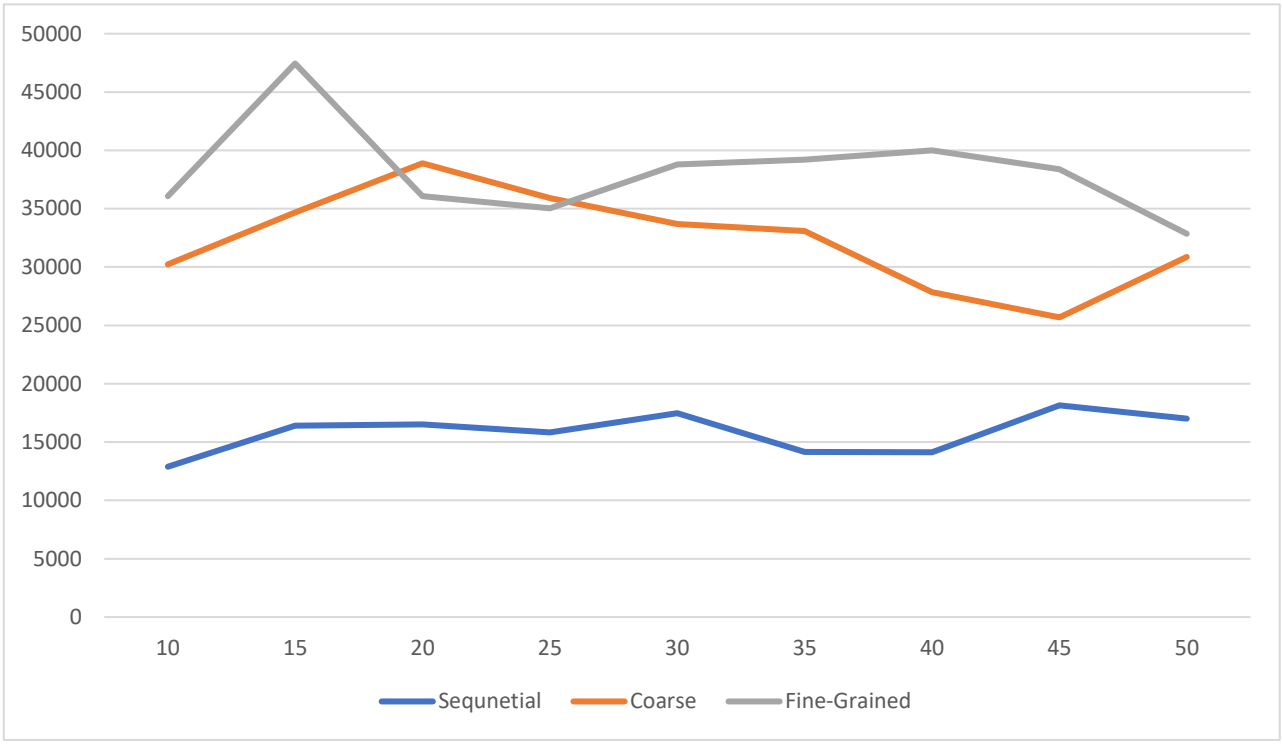
2. Graphs

2.1. Time Taken VS No. of Vertices

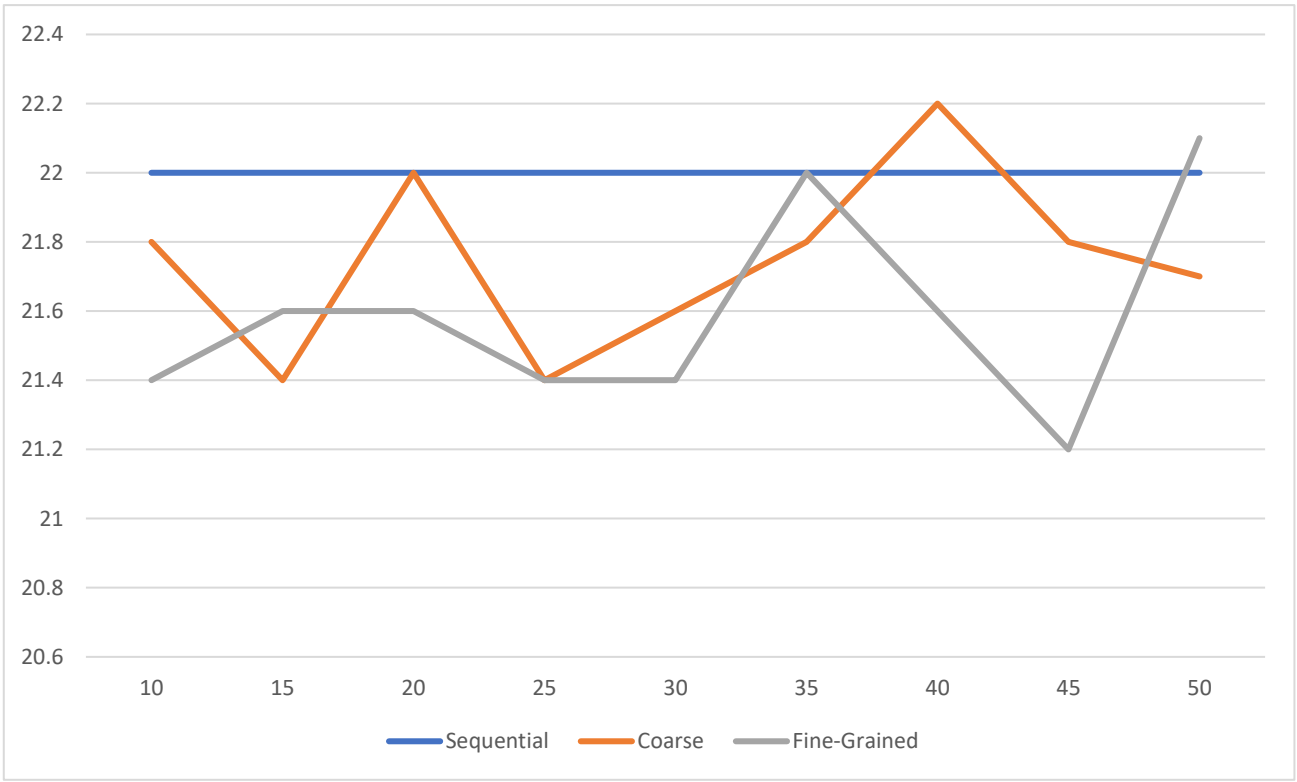


2.2. Colours used VS No. of vertices





2.4. Colours used VS No. of threads



ANALYSIS

In the first graph, as the number of vertices increase, the time taken by both the multi-threaded algorithms decreases, in general. Interestingly, fine-grained locking saves the most time as threads increase and it could be argued that for even more number of threads, it could take even lesser time than sequential execution. The sequential execution takes lesser time than both the locking/multithreading based algorithms because neither does it have the overhead of locking, nor does it have the need to go separate execution of several partitioned vertices into several threads. There is a sharp rise in coarse grained locking for 5x10^3, which could be argued to be an anomaly. Time in general also decreases for coarse grained execution. There was considerable variation between values sometimes. This is because the graphs are randomly generated and the colouring time depends heavily on the maximum degree of the graph, as well as on the way vertices are partitioned.

In the second graph, we see an interesting pattern. Firstly, colours used heavily depends on the graph which is randomly generated as well as the on its maximum degree. So looking at it as a continuous graph might not be the best course of action. But for each individual value of number of vertices, it can be argued that sequential algorithm gives the most consistent as well as the lowest number of colours used. Fine grained takes the most colours on average, but is close to coarse locking algorithm. This could be because of how the vertices are partitioned and since they are not coloured sequentially, they could take more colours when coloured in parallel.

In the third graph, as it should be, there is little to no variation in the time taken by sequential algorithm because it doesn't depend on the number of threads. In general, fine grained takes more time than coarse locking. This is probably because of the overhead of taking a lock at each traversal step in the case of fine-grained locking, which is not the case in coarse grained where only a single lock is used. However, parallelism is better preserved in fine-grained locking than in coarse-grained locking since contention for a global lock is reduced. Locks in general lead to overhead due to which both the

locking mechanisms take more time than sequential algorithm. A point can be made that for more threads and larger graphs, locking mechanisms prove to be supreme, but more data might be required to come to any satisfactory conclusions.
In the fourth graph, there is no variation in the number of colours used by the sequential algorithm, considering it has no relation to the number of threads in use. There is variation in the number of colours used by the multithreading locking algorithms. In general, fine-grained locking uses lesser number of colours than coarse locking. This is because of the randomly assigned partitions as well as how much parallelism is there in the execution. Not a lot can be said about the number of colours assigned because it highly depends on external factors(random graph, partitions, etc.). An interesting observation was that the more colours the multithreaded locking algorithm used, the lesser time it took. But there were variations. More data may be required.
So in conclusion, in general, for bigger graphs, fine-grained locking can provide faster graph colouring, since it can offer greater parallelism and reduce contention for a global lock. However, it requires a proper implementation to ensure mutual exclusion and prevent deadlocks. It takes more space and increased overhead for more locks, so proper implementation is key. Similarly, for greater number of threads, more parallelism allows for faster execution, although data is insufficient to confirm. Coarse grained locking is simpler to implement and it is easier to ensure mutual exclusion in its case. But for external vertices, the execution is essentially serialized. Variations can be observed because of the way partitions are randomly created, the maximum degree of the randomly generated graph, as well as other factors.