OPERATING SYSTEMS

ASSIGNMENT 5 REPORT

K.SRIVATSAN

ME18BTECH11016

Graph Colouring Problem:

Given a graph with 'N' vertices and 'M' edges, we have to colour all the vertices using multiple colours such that no two adjacent vertices have same colour.

There are many techniques present in order to make sure that lesser number of colours are used. However, here we are going to take the approach of Greedy Algorithm which will colour the graph. Although Greedy algorithm cannot guarantee the best solution(Least number of colours) but it can give a good solution.

Graph Colouring Greedy Algorithm:

This is an algorithm of Sequential (No parallel programming) Greedy algorithm for graph colouring.

- Initially chose a vertex say vertex 0, and give it a colour , say 0. (Numbers are used to represent various colours)
- Now repeat this for all remaining vertices:
    - Consider the current vertex 'v' and colour it with the lowest numbered colour that isn't used on any coloured vertices adjacent to it.
    - Suppose if all previously used colours appear on vertices adjacent to v, then assign it a new colour.

Pseudocode:

Colour(v_0) = 0.

Total_Colours = {0,1,2,……., m} // m is the degree of graph

For remaining vertices say v:

Colour(v) = min (Total_Colours – colour(adjacent(v))

Parallel Graph Colouring:

Here we use the power of parallel programming using threads to perform graph colouring.

Each thread is randomly assigned a few vertices(by random partitioning) to perform colouring. As long as there is no vertex which has its adjacent vertex in another thread's group, it can be parallelly coloured. Otherwise, we need to impose lock on such vertices whose at least one adjacent vertex lies in another thread's partition.

The first case vertices are called **Internal Vertices**.

The second case vertices are called **External/Boundary Vertices.**

There are two ways of implementing lock procedure, which gives rise to 2 different algorithms.

Coarse Grained Locking:

Here a global lock is applied on the section when one thread has to colour a boundary vertex. Once a boundary vertex is coloured by that thread, it releases this single global lock.

PseudoCode:

Let p denote number of threads.

Randomly perform partition of set of vertices 'V' into $V_1, V_2 \ldots\ldots, V_p$

Let 'm' denote maximum degree of graph

Let List 'B' contain all the boundary vertices in the graph.

Perform the following for each thread T_i

Initialize an m+1 dimension array, 'TotalColors' : {0,1,2,3……,m}

For each vertex 'v' within the partition of V_i such that 'v' is an internal vertex in this partition

Colour(v) = min (Total_Colours – colour(adjacent(v))

For each vertex 'v' within the partition of V_i such that 'v' is a boundary vertex in this partition

Lock the following section

Colour(v) = min (Total_Colours – colour(adjacent(v)) //CRITICAL SECTION

Release the Lock

CODE DESCRIPTION:

Global Variables:

adj_mat_ptr : This is an integer pointer to adjacency matrix.

part_mat_ptr : This is an integer pointer to a matrix which stores how vertices are partitioned.

mutex_lock : A semaphore which will be used to lock the critical section

Structs Used:

Vertices:

Stores the id of vertex and the colour assigned to it.

Timeval :

Used to gettimeofday and globally start_time and end_time are declared as its two instances.

User Defined Class:

compare:
This is used to pass to Priority Queue STL library compare function. It assigns the vertex with smaller id, higher priority.

User Defined Functions:

exist:

Takes priority queue as input and an integer variable 'id', It returns True if a vertex with its id equal to 'id' resides in the priority queue. If doesn't exists, then it returns false.

coloring:

This is the main function which performs the entire coloring algorithm as discussed earlier. It takes a Priority Queue, Number of vertices, Max degree of graph and Array of vertex informations. It also takes an index which will be used as we are partitioning vertices.

Main Code:

Input Processing:

- We initialize the semaphore 'mutex_lock' to 1.
- A file pointer *iptr is maintained to read the contents of file "input_params.txt"
- Initially n (number of vertices) is read from the file. Also, the number of threads to be used in read into a varible 'p'.
- Now the adjacency matrix is dynamically allocated and given a space of n x n. Adjacency matrix is also read from input file. Also, another matrix 'part_mat' is initialized with space p x n dynamically.

- Now we define an ‘n’ dimensional array of struct ‘vertices’ and assign vertex ids to each vertex. Currently the colour of each vertex is set to be -1.
- Using thread, ‘p’ threaded array is created named ‘threads’.
- We have an array ‘partition’ of size ‘n’.
- Initially a random vertex is assigned to each partition, so that each partition has atleast one vertex.
- For remaining vertices, we randomly assign a partition to a vertex. ‘partition[i]’ represents the partition into which i^{th} vertex has gone into.
- Now we count the maximum number of ‘1’ over the rows of adjacency matrix to get the degree of graph. We store it in ‘m’.
- A priority queue ‘PQ’ is initialized using STL and the compare class is passed as the comparing function to decide on priority.
- For every pair of vertices, if any one of the vertices is not in ‘PQ’, and within the pair both of them lie in different partitions and if an edge exists between them (‘Boundary Vertices’), they are pushed to PQ.
- In the ‘part_mat’ matrix, we store the information of which thread accesses which vertices. So, part_mat[i][j] = 1 implies that j^{th} vertex belongs to i^{th} partition, 0 implies otherwise.

Main Colouring Algorithm:

- Now we store time in start_time.
- Each thread is passed the function ‘coloring’ with suitable parameters.
- An array ‘arr’ is formed, which contains the $(index)^{th}$ row of ‘part_mat’ matrix, using the ‘index’ parameter.
- An array {0,1,2,…m} of size m+1 named ‘totalcolors’ is initialized.
- From vertex 1 to n,

  If a vertex is an internal vertex (that is, it doesn’t exist in the Priority Queue):

  We maintain a local copy of ‘totalcolors’ array and save it as ‘colors’

  Now we loop on all vertices to see which are adjacent to the current vertex.

  If a vertex is adjacent to it’s vertex:

  In the local copy ‘colors’

  If this adjacent vertex was already coloured, then it’s color in ‘colors’ array is set to INT_MAX.

  When we do min(colors) array, the adjacent vertex colour will never be set to current vertex as it is INT_MAX.

  Now we simply assign a colour based on min(‘colors’) array.

If a vertex is a boundary vertex (that is, it does exist in the Priority Queue):

A lock is Acquired by a process that comes here first using 'mutex_lock'.

We maintain a local copy of 'totalcolors' array and save it as 'colors'

Now we loop on all vertices to see which are adjacent to the current vertex. If a vertex is adjacent to it's vertex:

In the local copy 'colors'

If this adjacent vertex was already coloured, then it's color in 'colors' array is set to INT_MAX.

When we do min(colors) array, the adjacent vertex colour will never be set to current vertex as it is INT_MAX.

Now we simply assign a colour based on min('colors') array.

The Acquired lock is now released.

- Algorithm ends here and now we store current time in end_time.

Output Processing:

- We have an ofstream output which sends output to a file 'output.txt'.
- The colour of each vertex is stored in an 'n' dimensional array 'color_array'. The max of this array +1 denotes the number of colours used in this algorithm.
- Time taken is also printed in milliseconds. Finally, the colour of each vertex is printed out.

Fine Grained Locking:

Here 'n' locks are used. One for each vertex. So, in a fine manner, access to each vertex can be controlled.

PseudoCode:

Let p denote number of threads.

Randomly perform partition of set of vertices 'V' into V_1,V_2……,V_p

Let 'm' denote maximum degree of graph

Let List 'B' contain all the boundary vertices in the graph.

Perform the following for each thread T_i

Initialize an m+1 dimension array, 'TotalColors' : {0,1,2,3……,m}

For each vertex 'v' within the partition of V_i such that 'v' is an internal vertex in this partition

Colour(v) = min (Total_Colours – colour(adjacent(v))

For each vertex ‘v’ within the partition of V_i such that ‘v’ is a boundary vertex in this partition

Initialize a list ‘A_i’ which contains all the adjacent vertices to ‘v’ and ‘v’ itself.

Lock all vertices within ‘A_i’ in increasing order of their vertices (To prevent Deadlock)

Colour(v) = min (Total_Colours – colour(adjacent(v)) //CRITICAL SECTION

Unlock all vertices in ‘A_i’

CODE DESCRIPTION:

Global Variables:

adj_mat_ptr : This is an integer pointer to adjacency matrix.

part_mat_ptr : This is an integer pointer to a matrix which stores how vertices are partitioned.

mutex_lock_ptr : A global semaphore pointer which points to the array of ‘n’ semaphores.

Structs Used:

Vertices:

Stores the id of vertex and the colour assigned to it.

Timeval :

Used to gettimeofday and globally start_time and end_time are declared as its two instances.

User Defined Class:

compare:
This is used to pass to Priority Queue STL library compare function. It assigns the vertex with smaller id, higher priority.

User Defined Functions:

exist:

Takes priority queue as input and an integer variable ‘id’, It returns True if a vertex with its id equal to ‘id’ resides in the priority queue. If doesn’t exists, then it returns false.

coloring:

This is the main function which performs the entire coloring algorithm as discussed earlier. It takes a Priority Queue, Number of vertices, Max degree of graph and Array of vertex informations. It also takes an index which will be used as we are partitioning vertices.

Main Code:

Input Processing:

- A file pointer *iptr is maintained to read the contents of file "input_params.txt"
- Initially n (number of vertices) is read from the file. Also, the number of threads to be used in read into a varible 'p'.
- Now the adjacency matrix is dynamically allocated and given a space of n x n. Adjacency matrix is also read from input file. Also, another matrix 'part_mat' is initialized with space p x n dynamically.
- Now we define an 'n' dimensional array of struct 'vertices' and assign vertex ids to each vertex. Currently the colour of each vertex is set to be -1.
- We initialize the semaphore array of size 'n' as 'mutex_locks'. Initialize all entries to 1.
- Using thread, 'p' threaded array is created named 'threads'.
- We have an array 'partition' of size 'n'.
- Initially a random vertex is assigned to each partition, so that each partition has atleast one vertex.
- For remaining vertices, we randomly assign a partition to a vertex. 'partition[i]' represents the partition into which i^{th} vertex has gone into.
- Now we count the maximum number of '1' over the rows of adjacency matrix to get the degree of graph. We store it in 'm'.
- A priority queue 'PQ' is initialized using STL and the compare class is passed as the comparing function to decide on priority.
- For every pair of vertices, if any one of the vertices is not in 'PQ', and within the pair both of them lie in different partitions and if an edge exists between them ('Boundary Vertices'), they are pushed to PQ.
- In the 'part_mat' matrix, we store the information of which which thread accesses which vertices. So, part_mat[i][j] = 1 implies that j^{th} vertex belongs to i^{th} partition, 0 implies otherwise.

Main Colouring Algorithm:

- Now we store time in start_time.
- Each thread is passed the function 'coloring' with suitable parameters.
- An array 'arr' is formed, which contains the $(index)^{th}$ row of 'part_mat' matrix, using the 'index' parameter.
- An array {0,1,2,…m} of size m+1 named 'totalcolors' is initialized.
- From vertex 1 to n,

  If a vertex is an internal vertex (that is, it doesn't exist in the Priority Queue):

  We maintain a local copy of 'totalcolors' array and save it as 'colors'

  Now we loop over all vertices to find which all are adjacent to the current vertex.

If a vertex is adjacent to it's vertex:

In the local copy 'colors'

If this adjacent vertex was already coloured, then it's color in 'colors' array is set to INT_MAX.

When we do min(colors) array, the adjacent vertex colour will never be set to current vertex as it is INT_MAX.

Now we simply assign a colour based on min('colors') array.

If a vertex is a boundary vertex (that is, it does exist in the Priority Queue):

Another priority Queue 'A' is created where we push all the neighbours of the vertex and the vertex itself.

A copy of this 'A' is made into 'A_temp'. We get the id of all the vertices that are in this 'A_temp' and put lock on them. Since this is already a priority queue, locking is done in ascending order of vertex id.

We maintain a local copy of 'totalcolors' array and save it as 'colors'

Now we loop over all vertices to find which all are adjacent to the current vertex.

If a vertex is adjacent to it's vertex:

In the local copy 'colors'

If this adjacent vertex was already coloured, then it's color in 'colors' array is set to INT_MAX.

When we do min(colors) array, the adjacent vertex colour will never be set to current vertex as it is INT_MAX.

Now we simply assign a colour based on min('colors') array.

Now we again release the locks using the same way we acquired the locks.

- Algorithm ends here and now we store current time in end_time.

Output Processing:

- We have an ofstream output which sends output to a file 'output.txt'.
- The colour of each vertex is stored in an 'n' dimensional array 'color_array'. The max of this array +1 denotes the number of colours used in this algorithm.
- Time taken is also printed out in milliseconds.
- Finally, the colour of each vertex is printed out.

TASK 1:

Time Taken vs Number of vertices (Across various algorithms)

Parameters Used :

Number of Vertices : Varied from 6000 to 10,000 in steps of 1000

Number of threads : 20

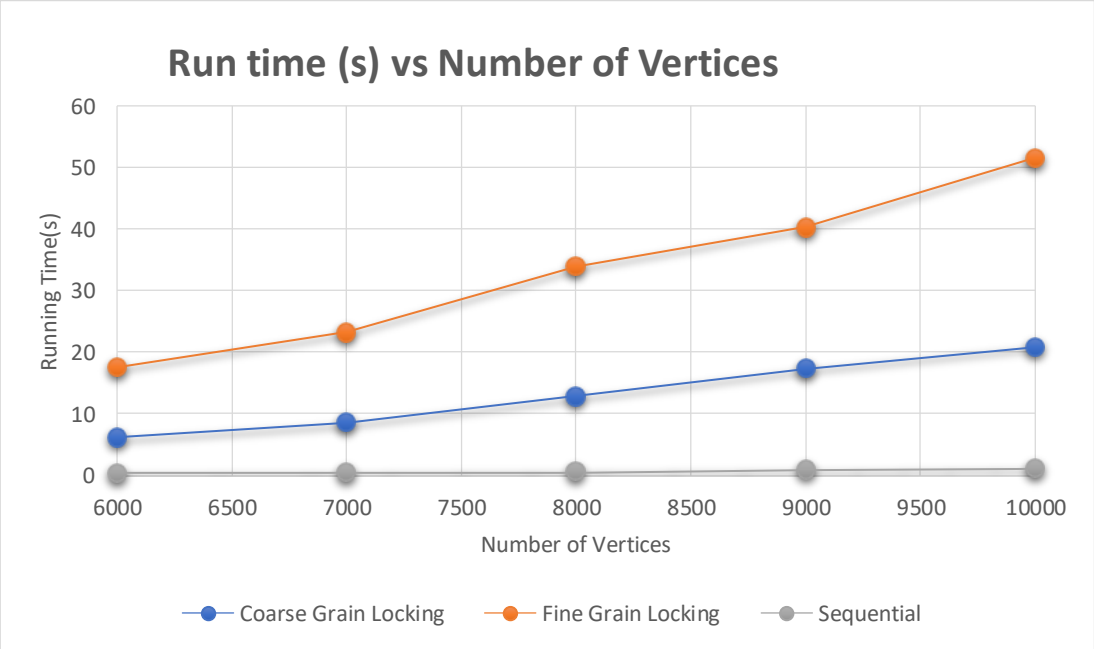
Expectation:

- As number of vertices increases, we expect more time taken for a particular algorithm.
- For given number of vertices, Fine grain must take least time while Sequential must take the most.

TABLE:

Time Taken (s) vs No. of threads					
	6000	7000	8000	9000	10000
Coarse Grain Locking	6.204	8.61	12.829	17.306	20.881
Fine Grain Locking	17.564	23.292	33.895	40.31	51.478
Sequential	0.419	0.547	0.713	0.884	1.137

GRAPH:



OBSERVATIONS AND THEIR EXPLANATIONS:

1) As the number of vertices increases, time taken by a specific algorithm increases. The increase is larger for Fine Grain and Coarse Grain as compared to the sequential algorithm.
   Explanation:
   As vertices increases, the computation time eventually increases due to more work. As Fine and Coarse grain algorithms involves in more complexity than a simple Sequential algorithm without any threads, the rise is higher for the Thread related algorithms.
2) For a given number of vertices:
   a) For low vertices, all the three algorithms almost have similar runtimes.

   Explanation:

   Any algorithm will take less time as vertices become low. It will be hard to differentiate between algorithms as all will take quicker times to complete.

   b) For moderate or large vertices, Fine grain seems to perform worst while the sequential performs best.

   Explanation:

   - This is clearly an anomaly. The reasons are as follows:
   - Since we use one lock per vertex in Fine grain algorithm, there is a lot of time utilized in properly acquiring lock for specified vertices and also for releasing them. On a high-speed system, it might give a better performance. However, on personal laptops, there is a large delay due to wait and signal signals.
   - Fine Grain algorithm makes use of a list to maintain the neighbour vertices of a vertex which lacks in the other 2 algorithms. Inserting elements into the list and later retrieving them to provide locks in proper ascending order takes a very significant amount of time.
   - Both the above reasons apply for a Boundary vertex. As the partitions are random, most of the vertices are expected to be boundary vertices which implies there are large number of vertices which undergo the process of above two points.
   - The reason why there are many Boundary vertices is because, random partitioning may not ensure uniform load on each partition.
   - Another important point is that in Sequential algorithms, there no concept of boundary or internal vertices, hence this algorithm works rapidly.
   - Lock related reasons can be applied to justify why Coarse grain taken larger time than sequential. We can clearly see that more locks, more wait and post signals, more the time is taken by the algorithm. Hence Fine grain takes more time than Coarse grain which in turn takes a slightly more time than the Sequential.

- Since most of the parameters are randomized, random effects can also cause anomalies.

TASK 2:

Number of Colours used vs Number of vertices (Across various algorithms)

Parameters Used :

Number of Vertices : Varied from 6000 to 10,000 in steps of 1000

Number of threads : 20

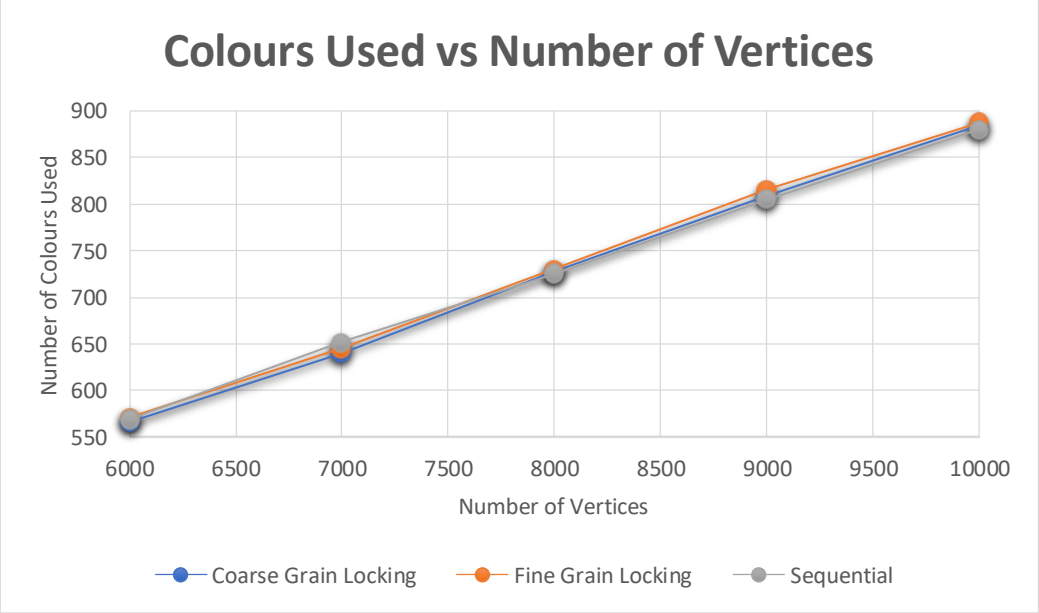
Expectation:

- As number of vertices increases, we expect more colours used for a particular algorithm.
- For given number of vertices, it is difficult to expect on number of colours used but however it is expected to stay somewhat nearer to each other.

TABLE:

Colours Used vs No. of threads					
	6000	7000	8000	9000	10000
Coarse Grain Locking	566	640	727	809	884
Fine Grain Locking	571	645	730	815	887
Sequential	569	652	725	805	880

GRAPH:



OBSERVATIONS AND THEIR EXPLANATIONS:

1) As the number of vertices increases, number of colours used by a specific algorithm increases. The increase is pretty much the same for all three algorithms.
   Explanation:
   As vertices increases, in order to satisfy the colouring constraints, we may require newer colours. This increases the total colours used. Majorly the total colours is dependent on vertices only, hence there is minimal change due to change in algorithm.
2) For given number of vertices, number of colours used by different algorithms is nearly the same.
   Explanation:
   The major difference between the algorithms is only parallelism. Colouring boundary vertices is very much sequential as they are part of critical sections. Hence there is hardly any difference between the colours used. There are slight differences which could have occurred due to the randomization involved in the algorithms.

TASK 3:

Time Taken vs Number of threads (Across various algorithms)

Parameters Used :

Number of threads : Varied from 10 to 50 in steps of 10

Number of vertices : 10000

Expectation:

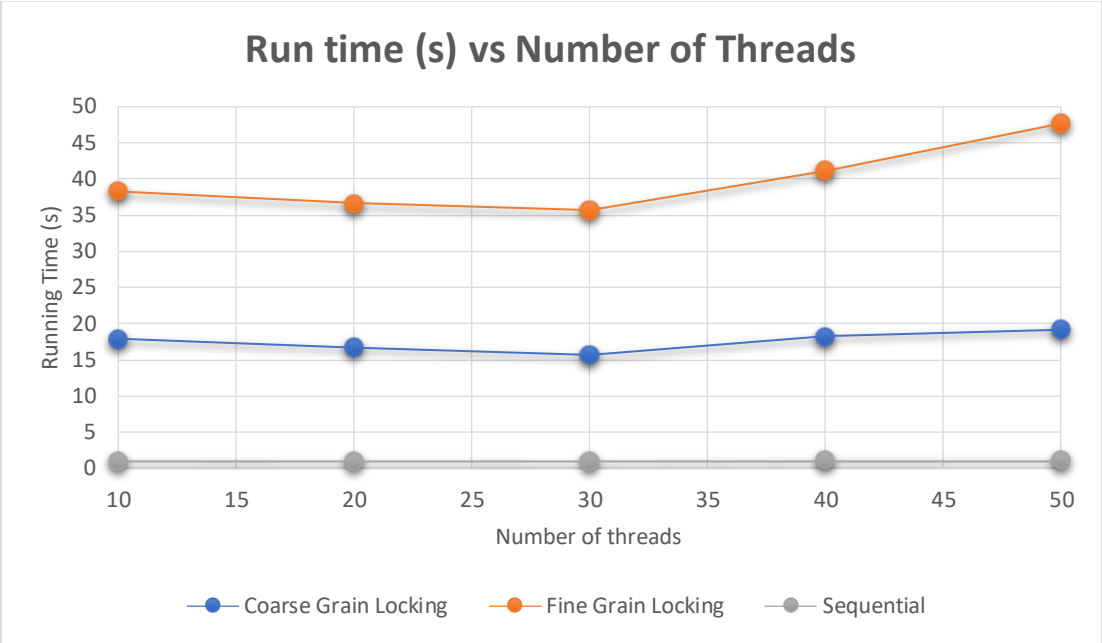
As number of threads increases, we expect less time taken for a particular algorithm due to parallelism. However Sequential is independent of time.

For given number of threads, Fine grain must take least time while Sequential must take the most.

TABLE:

Time Taken (s) vs No. of threads					
	10	20	30	40	50
Coarse Grain Locking	17.78713	16.69657	15.60893	18.20432	19.16955
Fine Grain Locking	38.25407	36.6073	35.62331	41.05537	47.58624
Sequential	0.965509	0.931443	0.887786	1.078261	1.103221

GRAPH:



OBSERVATIONS AND THEIR EXPLANATIONS:

1) As the number of threads increases, time taken by a specific algorithm decreases for some time and we see a larger increase. Clearly the sequential algorithm has a more constant profile as it involves no concept of threads.
   Explanation:
   For lower threads:
   As the threads increases slightly, we see that time taken reduces which shows signs of parallelism within an algorithm.
   For more threads:
   The time taken seems to increase which is an anomaly.
   Reasons:
   - The major reason is computational inefficiency, due to lack of high CPU core parallelism.
   - After a certain number of threads, the CPU is unable to produce parallelism, which has resulted in increase on running time.
2) For a given number of threads:
   Fine grain seems to perform worst while the sequential performs best.

   Explanation:

   - This is clearly an anomaly. The reasons are as follows:
   - Since we use one lock per vertex in Fine grain algorithm, there is a lot of time utilized in properly acquiring lock for specified vertices and also for releasing them. On a high-speed system, it might give a better performance. However, on personal laptops, there is a large delay due to wait and signal signals.

- Fine Grain algorithm makes use of a list to maintain the neighbour vertices of a vertex which lacks in the other 2 algorithms. Inserting elements into the list and later retrieving them to provide locks in proper ascending order takes a very significant amount of time.
- Both the above reasons apply for a Boundary vertex. As the partitions are random, most of the vertices are expected to be boundary vertices which implies there are large number of vertices which undergo the process of above two points.
- The reason why there are many Boundary vertices is because, random partitioning may not ensure uniform load on each partition.
- Another important point is that in Sequential algorithms, there no concept of boundary or internal vertices, hence this algorithm works rapidly.
- Lock related reasons can be applied to justify why Coarse grain takes larger time than sequential. We can clearly see that more locks, more wait and post signals, more the time is taken by the algorithm. Hence Fine grain takes more time than Coarse grain which in turn takes a slightly more time than the Sequential.
- Since most of the parameters are randomized, random effects can also cause anomalies.

TASK 4:

Number of Colours Used vs Number of threads (Across various algorithms)

Parameters Used :

Number of threads : Varied from 10 to 50 in steps of 10

Number of vertices : 10000

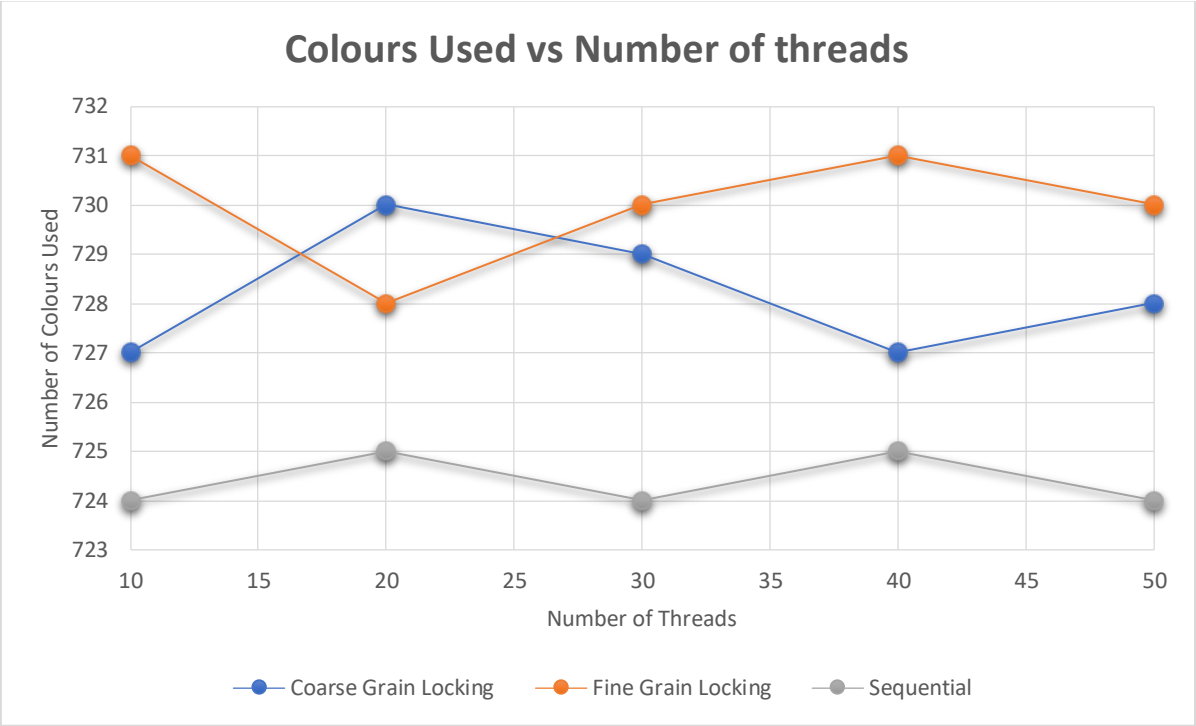
Expectation:

- As number of threads increases, it is difficult to expect on number of colours used but however it is expected to stay somewhat nearer to each other.
- For given number of vertices, it is difficult to expect on number of colours used but however it is expected to stay somewhat nearer to each other.

TABLE:

Colours Used vs No. of threads					
	10	20	30	40	50
Coarse Grain Locking	727	730	729	727	728
Fine Grain Locking	731	728	730	731	730
Sequential	724	725	724	725	724

GRAPH:



OBSERVATIONS AND THEIR EXPLANATIONS:

1) As the number of threads increases, number of colours used by a specific algorithm fluctuates. The fluctuation is somewhat within a similar range for an algorithm.
   Explanation:
   - Majorly the total colours is dependent on vertices only, hence there are fluctuations while we test it with a lesser independent feature such as threads.
   - The fluctuations can also be caused by randomizations involved between successive iterations.
2) For given number of threads, number of colours used by different algorithms is nearly the same. However sequential seems to take lesser colours, just slightly lesser.
   - Explanation:
     The major difference between the algorithms is only parallelism. Colouring boundary vertices is very much sequential as they are part of critical sections. Hence there is hardly any difference between the colours used. There are slight differences which could have occurred due to the randomization involved in the algorithms.
   - As Sequential algorithms colour without any real thread manipulations, they might have got a near optimal result quickly than other algorithms.