Report

Source code implementation and algorithm working details:

n -> stores no of vertices

n_threads -> stores no of threads

Here 2 structures are used

1. node: it contains the data of each vertex like their neighbours,its colour etc.
   Vertex -> it is the identity of the node .
   colour -> it stores the colour of the vertex
   partition-> it tells us that in which partition/thread the vertex belongs to
   type -> it gives an idea of weather the vertex is internal or external .
   colour_array-> it is used while detecting the colours of the neighbour in greedy approach.
2. Pnode: it contains the data of each thread , i.e the vertices that are in the control of the thread
   vertex-> the vertices that are in the partition of that thread.

struct node**array : n array pointers are created dynamically , for the vertex representation and for each vertex their neighbours are linked using next.

struct pnode**parray : n_thread parray pointers are created dynamically for the thread representation and their vertices are linked to each of the parray node.

<u>main():</u>

Here for synchronization I used semaphore locks.

For coarse-grained case only one lock is created and initialized to 1 ,while in case of fine-grained lock array of n semaphore locks are created. And all of them are initialized to 1.

After reading input from the file. array,parray are dynamically allocated space using malloc(). After reading each neighboring vertex a function named “add” is called which inserts its adjacent node at the end.

Each vertex is randomly assigned to a thread. Using rand().

Similarly padd() is used to insert a pnode at the end of each thread pnode.

And then in the main function itself we calculated which vertices are internal and which are external , and this data is stored in array[i]->type .

And then using for loop all the threads are assigned their jobs simultaneously. And one more thing is the threads used are vector threads.

And after that the required values are printed by following the indentations.

colouring() function for coarse-grained lock:

Each thread must colour the vertices that are in its partition. So, go through the parray[i] thread and go through the vertices in that partition , if the vertex type(internal or external) decides what to follow.

If it is an internal vertex , then go through all of its neighbouring vertices and read each of their colours , and make colour_array[colour] =true, which helps us to know which colours are allocated to their neighbours . and assign the minimum unassigned colour(minimum value of index at which false occurs in the array).

If it is an external vertex, here comes the concept of locks,otherwise more than one thread may access the vertex at the same time . so we just need to obtain a lock before accessing the external vertex ,which ensures only one thread accessing the vertex at an instant. And remove the lock after assigning a colour to it

colouring() function for fine-grained lock:

Here the only difference with coarse-grained lock is that the way we are meeting the synchronization and locking. Here unlike the above case each vertex has its own lock. so total n locks are used here . when a thread wants to colour a boundary vertex , then it locks all the neighbouring vertices in increasing order. So here the increasing order is met by using a function sort() which uses an array array[i]->sort_array , which contains the sorted elements in ascending order for that particular vertex. So before applying lock all the neighbouring vertices including itself are called by sort function and the sorted values are stored in that particular vertex only.

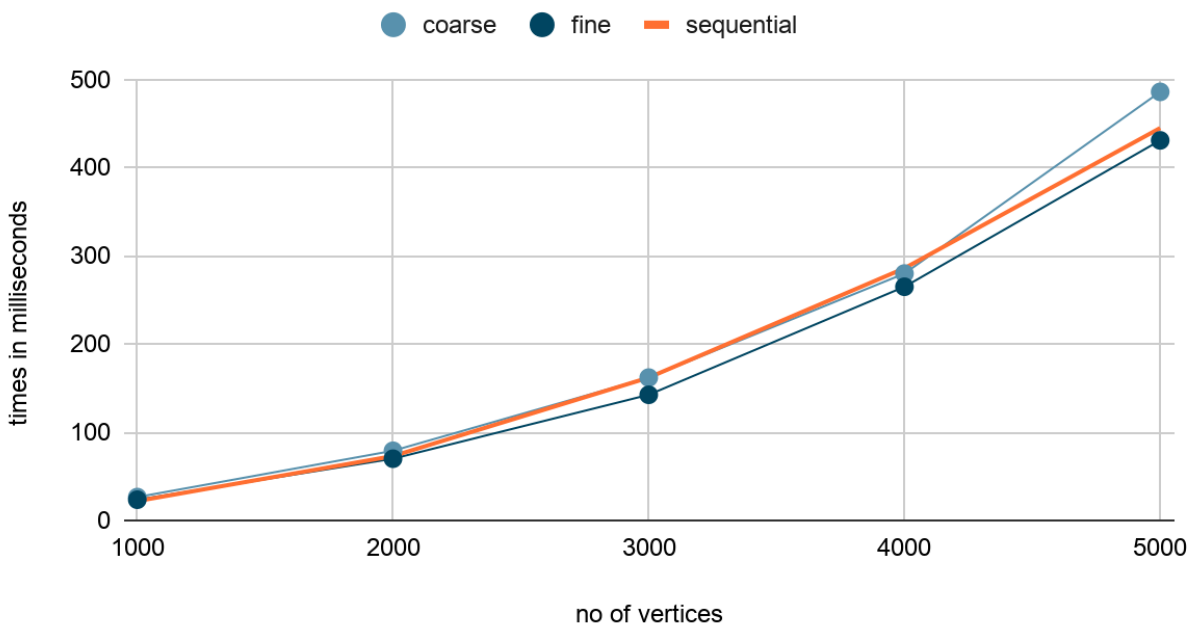
Now sem_wait() is called to the neighbouring vertices in increasing order of their vertex numbers. After assigning the colour to that vertex , using sem_post() unlock the neighbouring vertices in the same increasing order.

sort(): it is used for sorting the input and store in the array passed.

add(): it is like an insert function which insert node at last for the struct node.

padd(): it is like an insert function which insert node at last for the struct pnode.

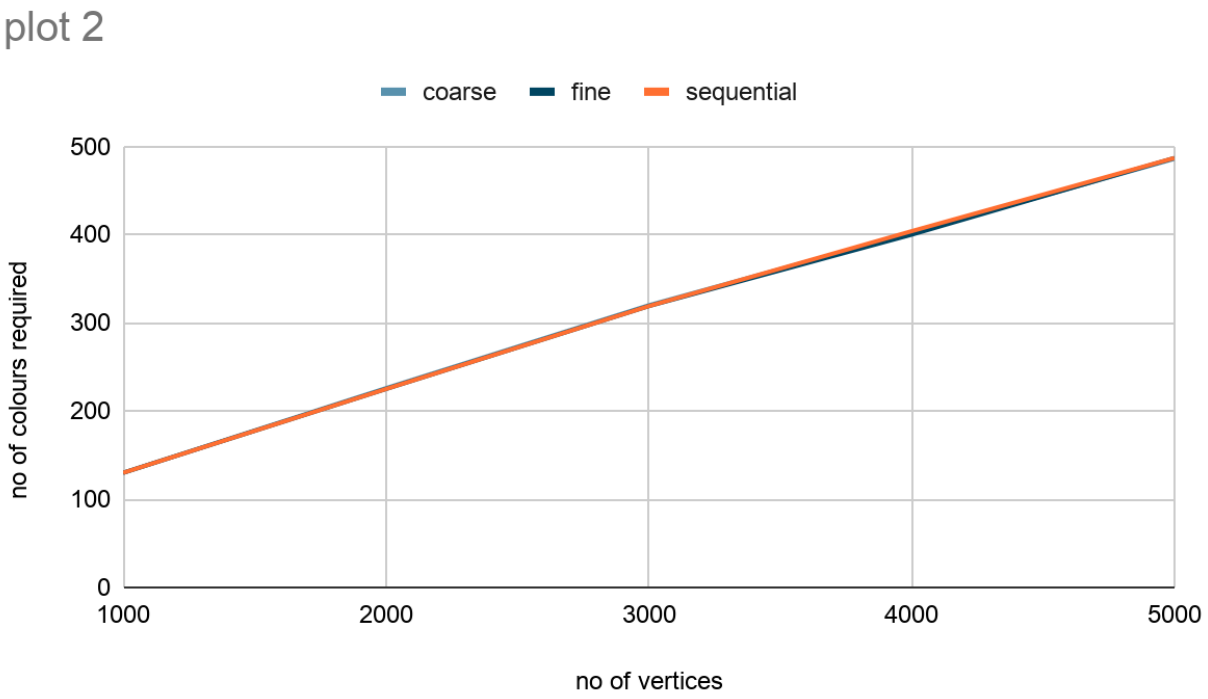
create_node(): it allocates space and initializes values.



No of threads:25

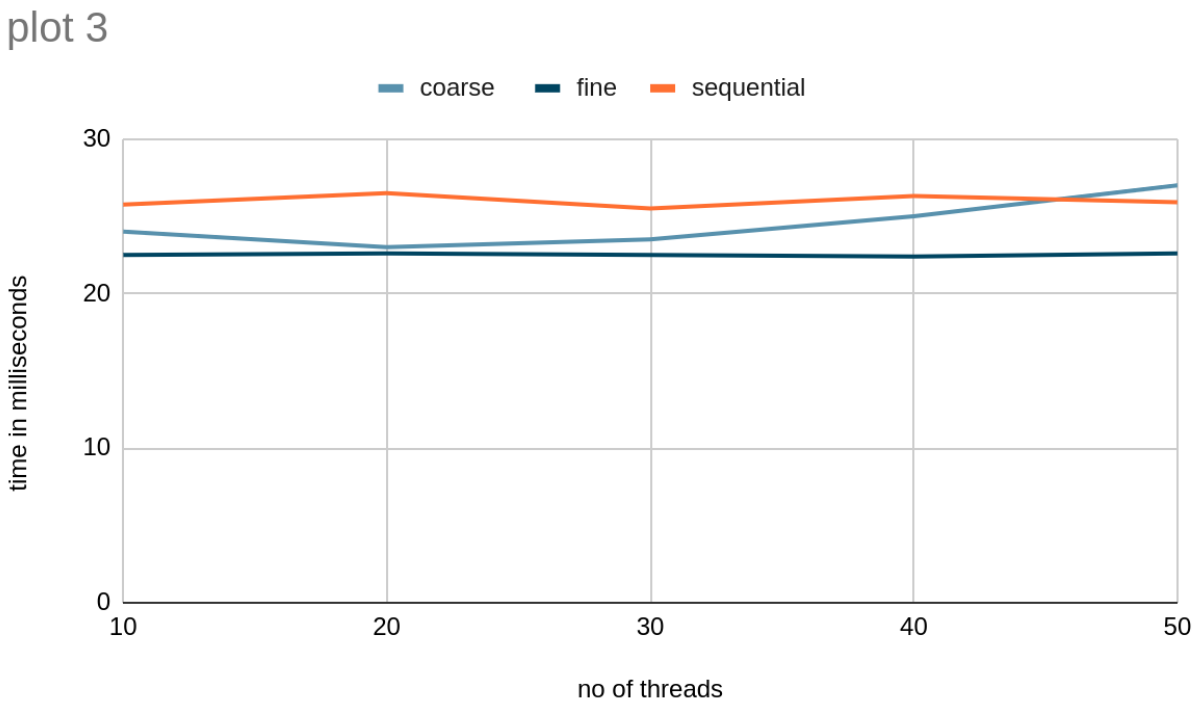
In case of fine grain only the neighbours are locked so it takes less time compared to coarse grain.

Sequential and remaining are almost same because of the locking .more no of times the locks are called so there id no much difference b/w sequential and other methods.



No of threads:25

Almost the colours are same for all the three .



No of vertices:1000

Sequential > coarse > fine. As the reason mentioned above.