Programming Assignment 5

Implementing Graph Colouring Algorithm using locks

Report

CS19BTECH11010

Goal: This assignment aims to implement the graph colouring algorithm in parallel using threads and synchronises them with the help of locks.

Report:

In the program input is read from file stream which contains number of vertices, threads(partitions) and an adjacency matrix representation of the graph(undirected).

In the program the graph is created using structure

```
struct graph
{
    int vertices;
    list <int> *adjlist;
};
```

With vertices representing number of vertices and list <int> *adjlist to store the address of lists containing linked list of adjacent vertices to a vertex.

Partition array is created using

```
int part[k+1];
    part[0] = 0;
    part[k] = n;
    int i =1;
    int n1 = n;
    while(i < k)
    {
        int num = rand() % n;
        if(num > n1/2)
        {
            i--;
        }
        else
        {
           part[i] = num+ part[i-1];
           n1 = n1-num;
        }
        i++;
    }
```

Then an partition array is created with k+1 elements with two adjacent elements representing the extremity vertices of a partition
(suppose part[i] = a & part[i+1] = b then partition is vertices from a to b-1)
Here random partition lengths are created using srand() for seeding and rand() % n for random partition lengths. By (num > n1/2) condition we ensure an proper array is always created.

Now we take note of time using gettimeofday() and initialize semaphores(considering coarse lock a single mutex and for fine grain lock mutexlocks equal to no. vertices)

Then proceed to creating threads using thread library and pass it function and extremeties of a partition

Now in thread function we create map <int, bool>pos to differentiate between internal and external vertices with true and false respectively.

```
for(int i =start;i<=ending;i++)
    {
        pos[i] = true;
        for(auto j = g->adjlist[i].begin(); j != g->adjlist[i].end(); ++j)
        {
           int num = *j;
           if( (num < start) || (ending < num) )
           {
              pos[i] = false;
              break;
           }
        }
    }
```

Then the algorithm is run for internal vertices without any problem but in case of external vertices

Coarse lock

Before coloring every external vertex a mutex lock is aquired for whole algorithm(critical section) and released after completion that is during lock process only one boundary vertex is colored

Sem_wait(&lock) and sem_post(&lock)

Whereas in

Fine-grain lock

Before colouring every boundary vertex and its adjacent vertices are locked for whole algo and released after completion .the difference here is that more that one boundary vertex can be coloured at same time considering they don't share same adjacent vertices.

```
  vector<int>order;
for(auto q = g->adjlist[i].begin(); q != g->adjlist[i].end(); ++q)
{
   order.push_back(*q);
}
sort(order.begin(),order.end());
pthread_mutex_lock(&mutexLocks[i]);
while(!(order.empty()))
{
   int x = order.back();
   order.pop_back();
   pthread_mutex_lock(&mutexLocks[x]);
}
```

The algorithm is

1. Color first vertex with first color.
2. Do following for remaining V-1 vertices.
  Consider the currently picked vertex and color it with the lowest numbered color that has not been used on any previously colored vertices adjacent to it. If all previously used colors appear on vertices adjacent to v, assign a new color to it.

Finally the threads are joined together and time is again noted.

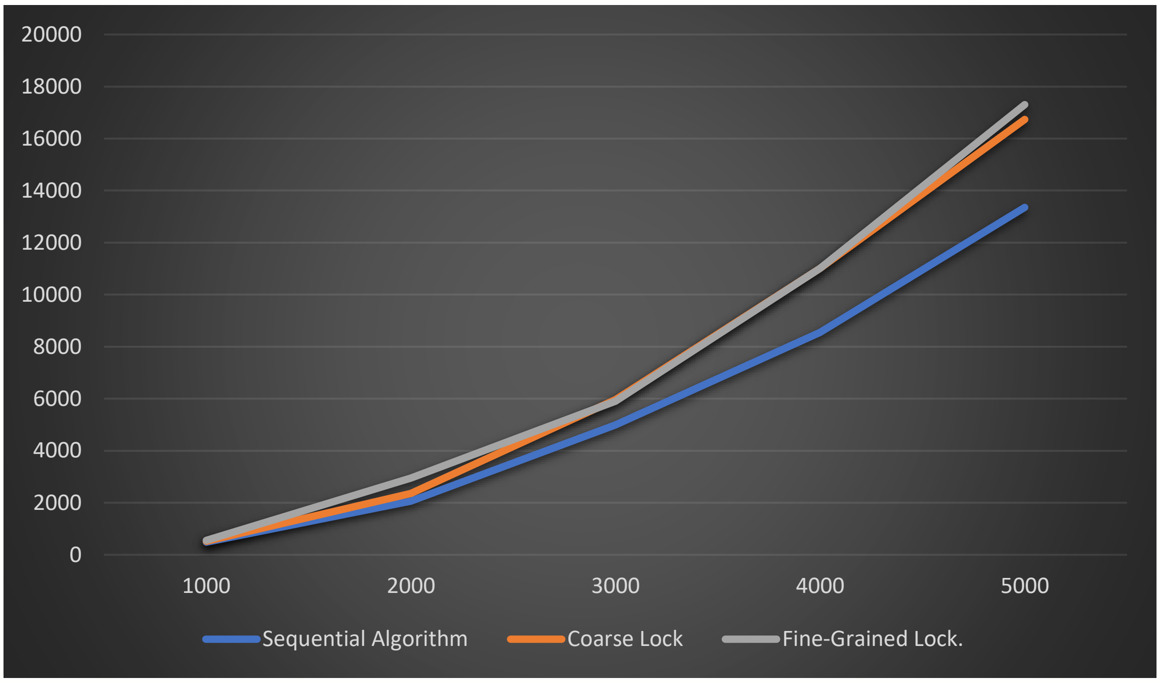
Then the respective log file is created using ofstream.

Plot1:

X-axis -> no. vertices

y-axis -> time taken in millisec

k = 20



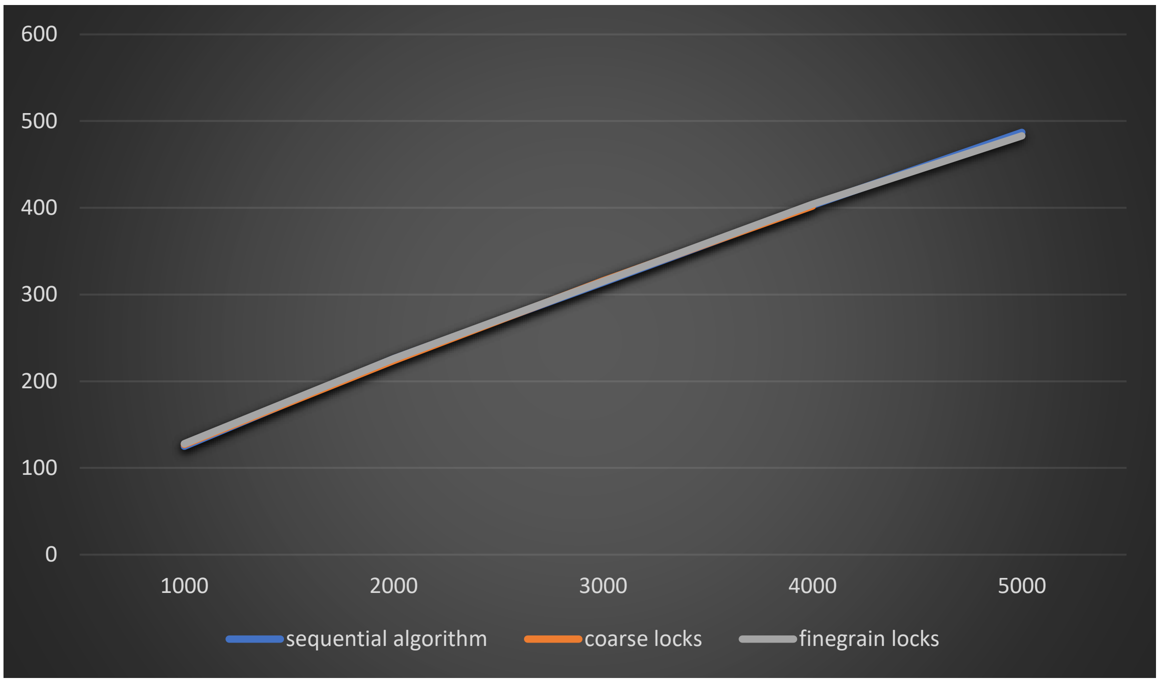
Here sequential takes least time due to thread creation overhead and internal,external division

Plot2:

X-axis -> no. vertices

y-axis -> no. colors used

k = 20

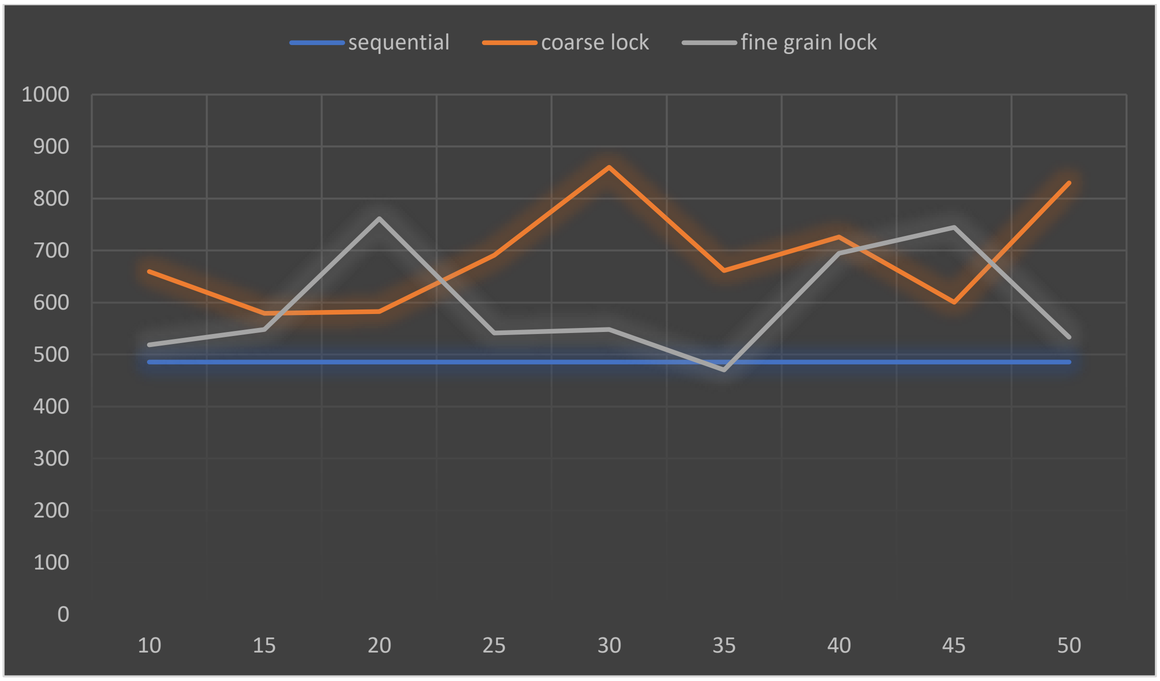


Plot3:

X-axis -> no. threads

y-axis -> time taken in millisec

n = 1000



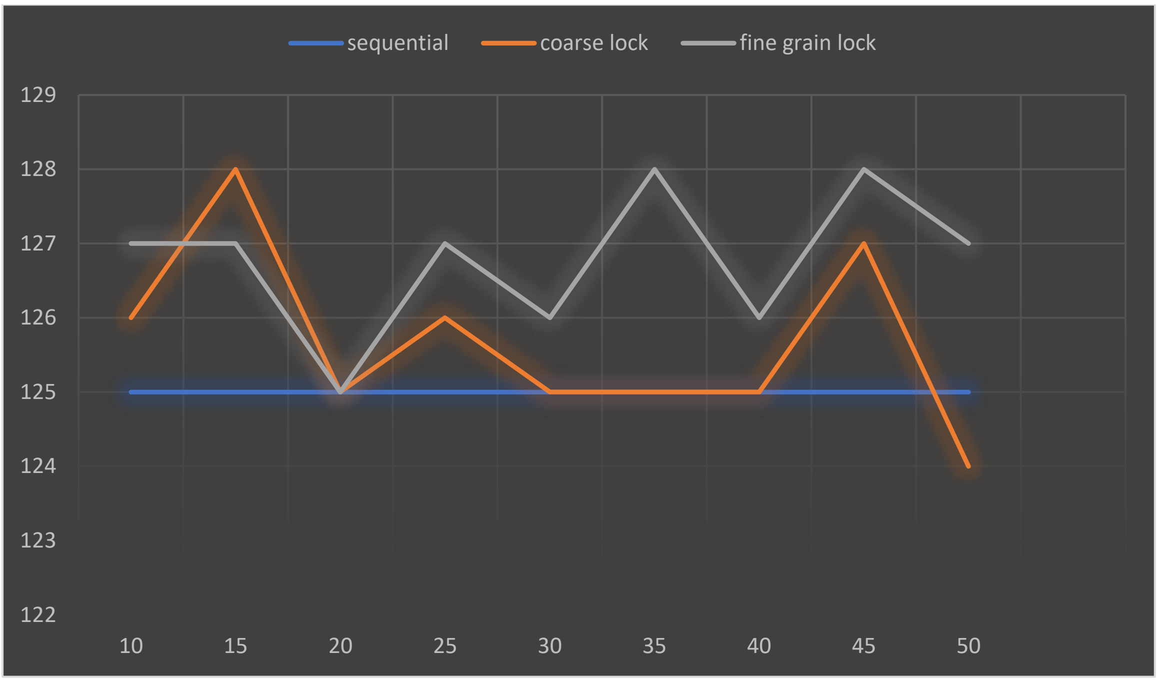
Sequential performs better then coarse lock and fine grain lock

Plot4:

X-axis -> no. threads

y-axis -> no. colors used

n = 1000


sequential
coarse lock
fine grain lock