a) In the parallel bfs thread, multiple threads may try relaxing the same vertex i.e. multiple threads may see d[v] = ∞ for the same v at the same time and try to modify d[v]. This is the race condition. However, the value of d[v] is not dependent on the previous value of d[v]. So the output will be unchanged in spite of the race condition.
Race condition could also arise when one thread is reading the value of d[v] while another thread is concurrently updating it. In this case, if read-write concurrency is not consistent, then the reader may read an inconsistent value.

b) A vertex v enters in $Q^{in}[i]$ (1 <= i <= p) only if it is adjacent to the current vertex and d[v] = INFINITY. This condition may be satisfied for more than one value of i concurrently and hence a vertex v may appear multiple times in Q^{in} for different values of i.
For any given thread, vertex v enters its queue only if it is adjacent to the current vertex and d[v] = INFINITY. Also, it enters the queue only after d[v] is set to a value less than INFINITY. Thereon, the value d[v] never gets reset to INFINITY. Therefore, for a given thread, the vertex v will enter the queue only once.
In order for any vertex to be in only one of the queues of Q^{in}, we need to acquire the lock on d[v] before we check its value and possibly update it and en-queue it. So that, any other thread waiting for the lock will see the value set to less than INFINITY once it acquires the lock. And hence, vertex v will not be present in more than one queue.

```
while ( True ) do
    a. while ( S[i] != 'empty' ) do
            i.  u  = S[i]:extract( )
            ii. for each v in adjacency(u) do
                    1. Lock(d[v])

                    2. if d[v]  = INFINITY then
                            a. d[v]   d[u] + 1
                            b. Qo:enque( v )

                    3. Unlock(d[v])

    b. t = 0
    c. Lock( i )
    d. while ( S[i] = 'empty' ) and ( t < Max-Steal-Attempts ) do
            i.  r = Rand( 1; p )
            ii. if Try-Lock( r ) then
                    1. if ( abs(S[r]) > Min-Steal-Size ) then
                            a. S[i] = second half of S[r]
            iii. Unlock( r )
            iv. t =  t + 1
    e. Unlock( i )
    f. if ( S[i] = 'empty' ) then break
```

For any given thread, vertex v enters its queue only if it is adjacent to the current vertex and d[v] = INFINITY. Also, it enters the queue only after d[v] is set to a value less than INFINITY.

c) Thereon, the value d[v] never gets reset to INFINITY even in consequent iterations . Therefore, same vertex cannot appear in Q^{in} in different iterations.

d) The total number of vertices in all the input queues is q_l.

In a given queue, the number of entries in the queue is halved after stealing.

But stealing can take place only if a queue has Min-Steal-Size number of vertices.

Therefore, the maximum number of iterations in which stealing can take place are:

Θ(log(ql/ Min-Steal-Size)).

The number of attempts made in every iteration is Max-Steal-Attempts.

The value of Max-Steal-Attempts is Θ(plogp) with high probability.(proved below in (e))

Therefore, the total number of attempts after which the stealing will start failing is:

the maximum number of iterations * the number of stealing attempts per iterations

= Θ(log(ql/ Min-Steal-Size))* Θ(plogp)

= Θ(plogp log(ql/ Min-Steal-Size))

with high probability (Proof by Triviality i.e. If E1 occurs with whp and E2 occurs when E1 occurs, then E2 also occurs whp)

e) The value of Max-Steal-Attempts should be such that the random number generated to decide the thread to steal from should cover all the threads. i.e. the minimum value required to generate all the numbers from 1 to p through the random number generator.

We know, the number of throws required such that all the bins contain at least one ball is Θ(nlogn) with high probability.

[

After, Θ(nlogn) throws, the probability that bin i gets 0 balls = $(1-1/n)^{c*n*\log n}$

By union bound, after Θ(nlogn)throws,

the probability[there exists a bin that has 0 balls] < $n * (1-1/n)^{c*n*\log n}$

= $n*(1/e)^{c*\log n}$

= $n*(1/n)^{c}$

= $1/n^{c-1}$

Thus, with high probability, there is no empty bin after Θ(nlogn) throws.

]

Similarly, the number of iterations required such that all the numbers from 1 to p are generated by the random number generator is Θ(plogp) i.e. cplogp where c >1.

f)

Approach 1: Span Approach

The time required by the serial algorithm to perform BFS is O(n+m)

$T_S = O(n+m)$

The time required by the parallel algorithm to perform BFS when there is only one processor

$T_1 = O(n+m)$

Span: $T_{infinity}$ = the time required when there are infinite number of processors

With infinite processors, time is spent in stealing the work till all qin[i]'s have minimum (O(1)) number of elements. This is done is log (n/D) time since we divide the queue into halves at each steal.

Where, D = diameter of the graph and hence n/D = average number of nodes at every level

With one element in each queue, the time required to process that node (i.e. to explore all its neighbours, set their distance and enqueue them if required), will be O(Δ).

Therefore, $T_{infinity} = O(\log(n/D)) + O(\Delta)$

Now, $T_p = T_1/p + T_\infty$ ….Graham Brent Law

Therefore, $pT_p = T_1 + pT_{infinity}$
$pT_p = O(n+m) + p * (O(\log(n/D)) + O(\Delta))$

$pT_p = O(n+m) + p * (O(\log(n/D)) + O(\Delta))$

For work optimality,
$pT_p = T_s$
$O(n+m) + p * (O(\log(n/D)) + O(\Delta)) = O(n+m)$

Therefore, $p * (O(\log(n/D)) + O(\Delta)) = O(n+m)$

size of the input graph (i.e. n+m) = $O(p*(\log(n/D) + \Delta)$

Approach 2:

At each BFS level each thread is either waiting to be launched, or exploring vertices, or trying to steal, or waiting at the sync point.

So total work done in parallel BFS = $T_{wait\_launch} + T_{explore\_vert} + T_{steal} + T_{wait\_sync}$

T_{wait_launch} = (Work done in for loop to spawn p-1 parallel bfs + Work done to launch parallel bfs from current thread) * number of levels)

= O (p*D).

$T_{explore\_vert} + T_{steal} = T_{pbfstotal}$ = total work done by all processing element in parallel BFS

$T_{pbfstotal}$ = Total work done in first for loop to explore vertices + Total work done in stealing in second

$= T_{total\_explore\_vert} + T_{total\_steal\_time}$

$T_{total\_explore\_vert}$ = Total work in exploring all edges of the graph = O (m+n)

$T_{total\_steal\_time}$ = (No of successful steal attempts before failing * Number of Max steal attempts done again + Number of unsuccessful steal attempts) * Number of levels

$= (\Theta (p \log p \log (q_i/\text{Min-Steal-Size}) * plogp + p* plogp) * D$
$= (\Theta (p \log p \log (q_i)* plogp + p* plogp) * D$……..(as Min-Steal-Size = $\Theta(1)$)

$T_{pbfstotal} = O(m+n) + (\Theta(p \log p \log(q_i)* plogp + p* plogp)] * D$

$T_{explore\_vert} + T_{steal} = O(m+n) + [(\Theta(p \log p \log(q_i)* plogp + p* plogp) * D]$
$T_{wait\_sync} = O (1) * D = O (D).$

So total work done in parallel BFS (pT_p)

$= O (p) + O (m+n) + [(\Theta (p \log p \log (q_i)* plogp + p* plogp) * D] + O(D)$

$= O(m+n) + [(\Theta(p \log p \log(q_i)* plogp + p* plogp) * D]$ (assuming $O(p) < O(n+m)$ and $O(D) < O(n+m)$.. as our graph contains edges which are much large in number than processing elements.. so ignoring it) …….. [1]

For serial BFS $T_s = O(n+m)$ ……[2]

Now for work optimality $pT_p = \Theta(T_s)$ ……..[3]

From [1], [2] and [3],

$O(m+n) + [(\Theta(p \log p \log(q_i)* plogp + p* plogp) * D] *p = \Theta(n+m)$

So ,

$[\Theta (p \log p \log(q_i)* plogp + p* plogp) * D] = O(n+m)$

So, $n+m = [\Theta (p \log p \log (q_i)* p logp+ p* plogp) * D] =$

$[p^2 * D * \log p \Theta(\log p \log(q_i) + 1)]$

m+n represents size of the graph and $[p^2 * \log p * D * \Theta(\log p \log(q_l))] =$

$\Theta(p^2 * (\log p)^2 * D * \log(q_l))$

represents the lower bound on it.

g)

In the given algorithm, for every node, we explore all its adjacent neighbours in a sequential loop.

In the above question, the term Δ comes because of the sequential loop that iterates O(Δ) times.

The exploration of the neighbours of a given vertex can be done in parallel and hence the running time of the loop can be reduced to O(1). Thus Δ disappears from the T_p.

With parallelized exploration of the adjacent vertices,

$T_p = T_1/p + T_\infty$ ….Graham Brent Law

$T_p = O(n+m)/p + (O(\log(n/D)) + O(1))$

$T_p = O((n+m)/p) + (O(\log(n/D))$

In scale free graphs, there are few nodes with large degrees. While exploring such nodes of such graphs, parallelizing the exploration of adjacent neighbours drastically improves the performance.

h)

The for loop in lines 10-13 can be executed parallely.

```
for i = 1 to p - 1 do
    spawn Parallel-BFS-Thread( I, G, Qout.q[i], d )
```

Here, the maximum time a thread waits to get launched is O(p) which now becomes O(1) because of concurrent forking of all the processes.

For question f earlier complexity was $O(p) + O(m+n) + [(\Theta(p \log p \log(q_l) * p \log p + p * \log p) * D] + O(D)$

It will become $O(m+n) + [(\Theta(p \log p \log(q_l) * p \log p + p * \log p) * D] + O(D)$.

For g, similarly contribution of O(p) during launch will become O(1).

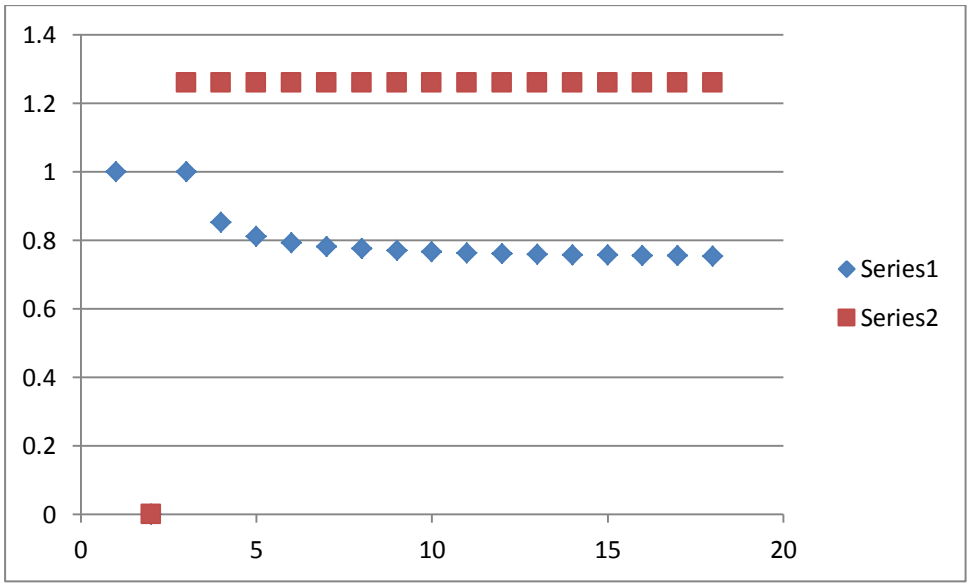
j)

The following is the table that compares the serial implementation with the parallel implementations in (f) and (g)

Data sets	Sequential (seconds)	Parallel (seconds)	Parallel_deltafree (seconds)	Speedup Factor for Parallel	Speedup Factor for Parallel_deltafree
cage15	2166	1073	1139	2.0186	1.9016
cage14	574	305	324	1.8819	1.7716
freescale	878	597	513	1.4706	1.711
wikipedia	1573	636	590	2.3473	2.66
kkt-power	16	14	16	1.142	1
rmat100M	6754	2032	2074	3.3238	3.2565
rmat1B	33644	6282	6010	5.355	5.598

k)

The CilkView scalability plot for the algorithm in (f)



The CilkView scalability plot for the algorithm in (g)