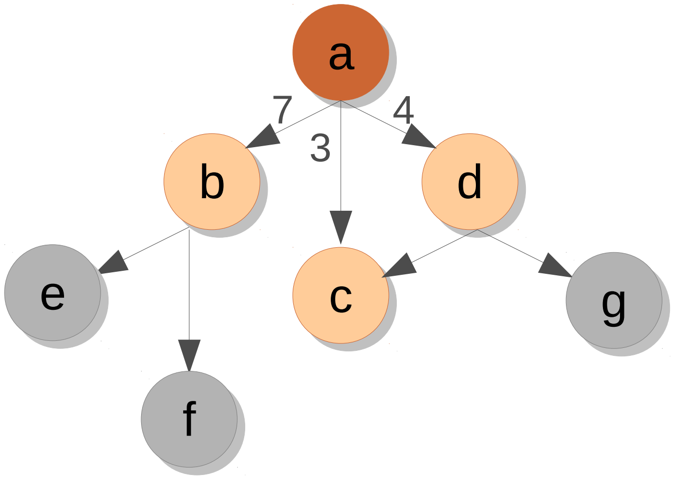
Synchronization

Recap

- You are given an input graph of India, and you want to compute the shortest path from Nagpur to every other city.
- Assume that you are given a GPU graph library and the associated routines.



```
__global__ void dsssp(Graph g, unsigned *dist) {
    unsigned id = ...
    for each n in g.allneighbors(id) {     // pseudo-code.
        unsigned altdist = dist[id] + weight(id, n);
        if (altdist < dist[n]) {
            dist[n] = altdist;
}   }   }
```

What is the error in this code?

Synchronization

- **Control + data flow**
- Atomics
- Barriers
- ...

Classwork: Implement mutual exclusion for two threads.

Initially, flag == false.

```

while (!flag) ;
S1;
```

```
S2;
flag = true;
```

Synchronization

- **Control + data flow**
- Atomics
- Barriers
- ...

Classwork: Implement mutual exclusion for two threads.

Classwork: Can we allow either **S1** or **S2** to happen first?

Initially, flag could be true or false.

```
while (!flag) ;
S1;
flag = false;
```

```
while (flag) ;
S2;
flag = true;
```

Assumptions:

- Reading of and writing to flag is atomic (seemingly one step).
- Both the threads execute their codes.
- flag is volatile.

Mutual Exclusion: 2 threads

- Let’s implement **lock()** and **unlock()** methods.
- The methods should be the same for both the threads (can have threadid == 0, etc.).
- Should use only control + data flow.

Mutual Exclusion: 2 threads

v1

- Thread ids are 0 and 1.
- Primitive type assignments are atomic.

```
lock:
    me = tid;
    other = 1 – me;
    flag[me] = true;
    while (flag[other])
        ;
unlock():
    flag[tid] = false;
```

- Mutual exclusion is guaranteed (if volatile).
- May lead to deadlock.
- If one thread runs before the other, all goes well.

Mutual Exclusion: 2 threads

v2

- Thread ids are 0 and 1.
- victim needs to be volatile.

```
volatile int victim;
lock:
    me = tid;
    victim = me;
    while (victim == me)
        ;
unlock():
    victim = me;
```

- Mutual exclusion is guaranteed.
- May lead to lack of progress.
- If threads repeatedly take locks, all goes well.

Peterson’s Lock

v3

```
volatile bool flag[2];
volatile int victim;
lock:
    me = tid;
    other = 1 – me;
    flag[me] = true;
    victim = me;
    while (flag[other] &&
             victim == me)
        ;
unlock():
        flag[tid] = false;
```

- Mutual exclusion is guaranteed.
- Does not lead to deadlock.
- The algorithm has progress.
- flag indicates if a thread is interested.
- victim = me is like saying “You before me”

What about N threads?