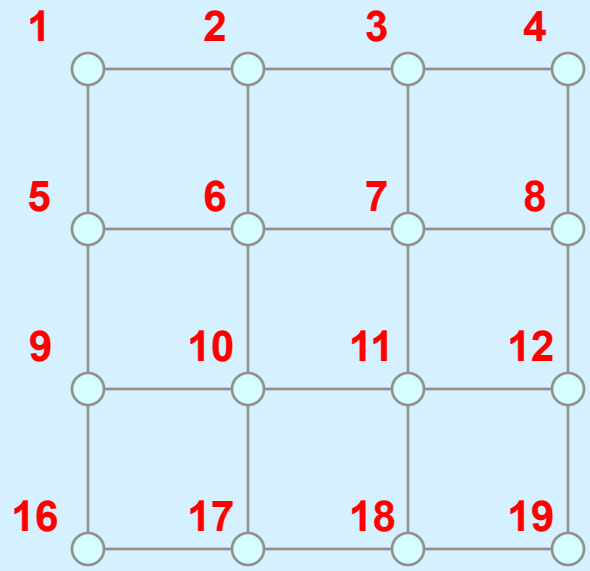
CS 240A : Breadth-first search in Cilk++

Thanks to Charles E. Leiserson for some of these slides

Breadth First Search

- Level-by-level graph traversal
- Serial complexity: $T_1 = \Theta(m+n)$



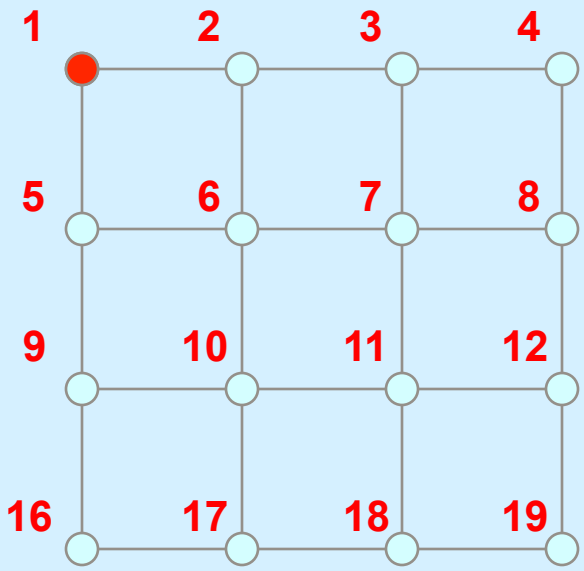
Graph: **G(E,V)**

E: Set of edges (size m)

V: Set of vertices (size n)

Breadth First Search

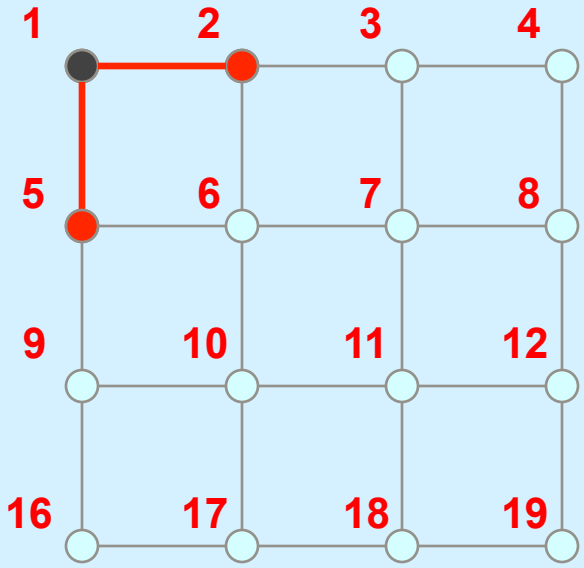
- Level-by-level graph traversal
- Serial complexity: $T_1 = \Theta(m+n)$

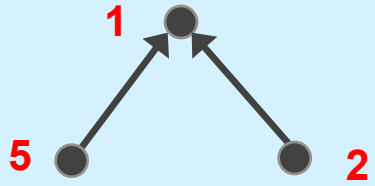


1 Level 1

Breadth First Search

- Level-by-level graph traversal
- Serial complexity: $T_1 = \Theta(m+n)$



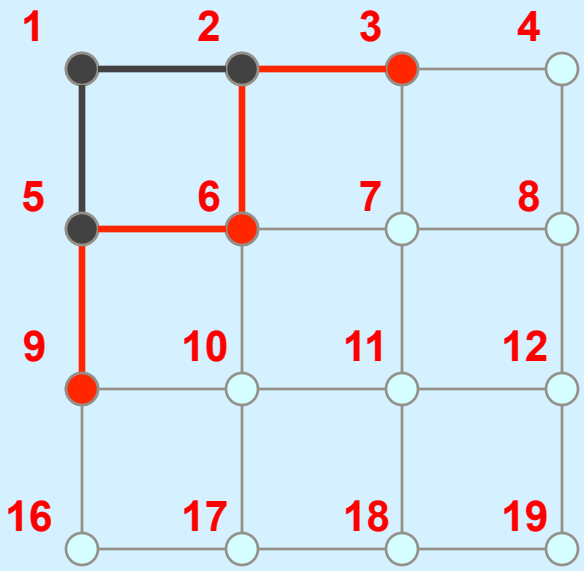


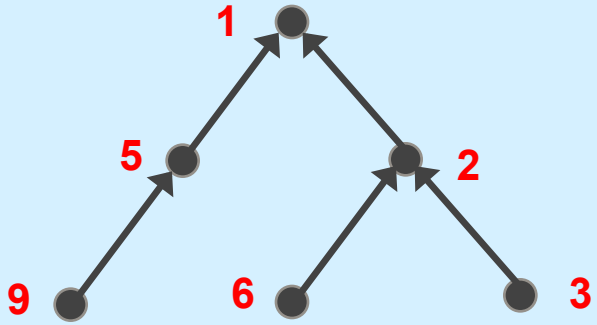
Level 1

Level 2

Breadth First Search

- Level-by-level graph traversal
- Serial complexity: $T_1 = \Theta(m+n)$





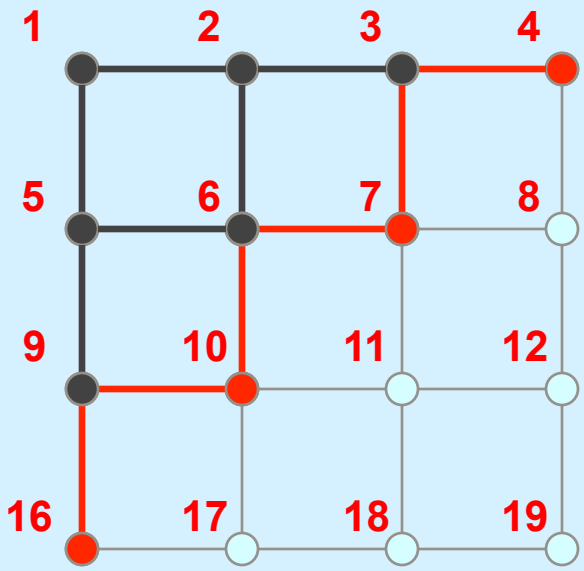
Level 1

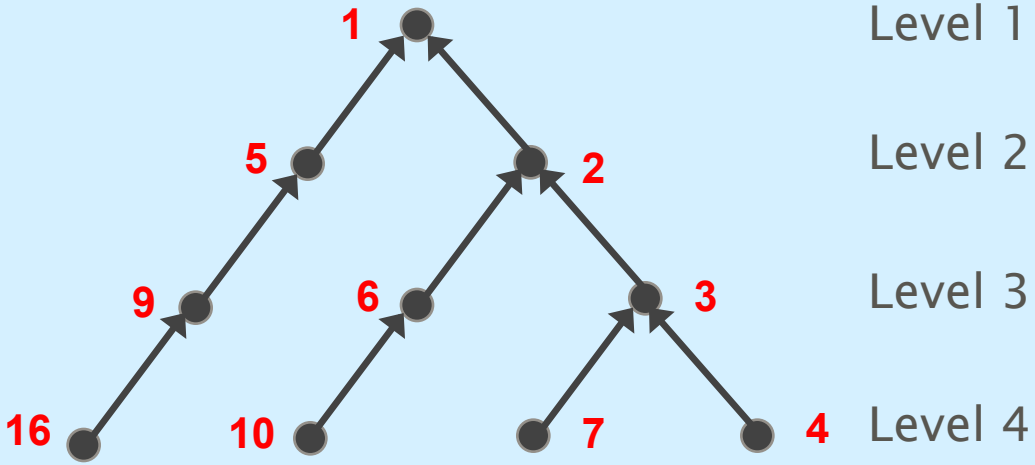
Level 2

Level 3

Breadth First Search

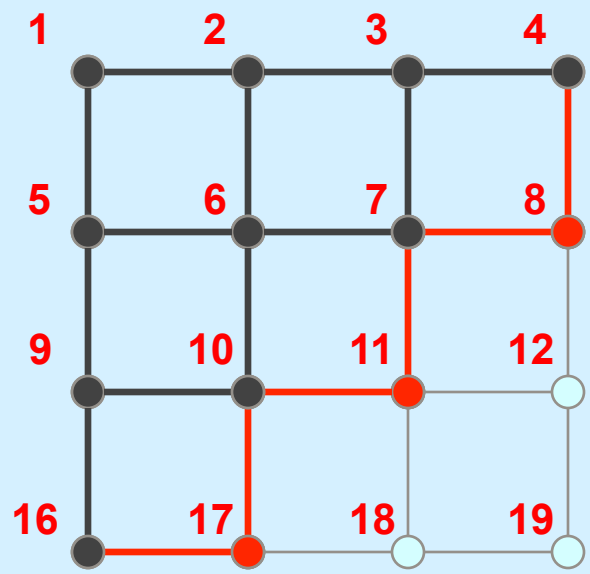
- Level-by-level graph traversal
- Serial complexity: $T_1 = \Theta(m+n)$

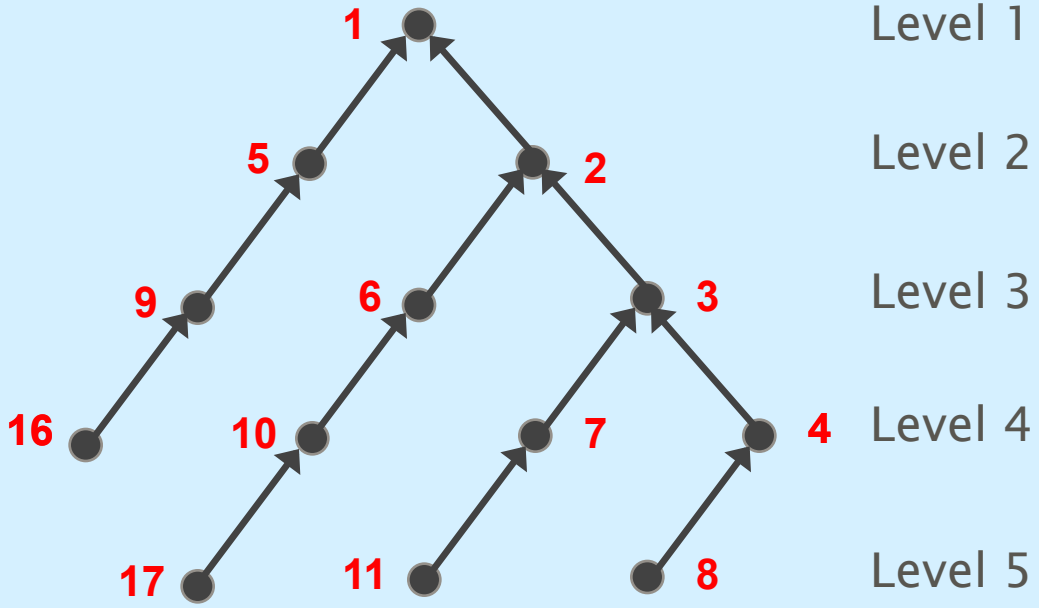




Breadth First Search

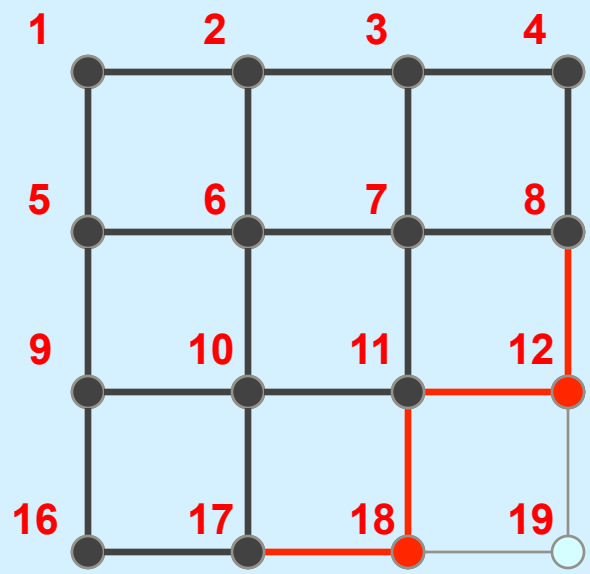
- Level-by-level graph traversal
- Serial complexity: $T_1 = \Theta(m+n)$

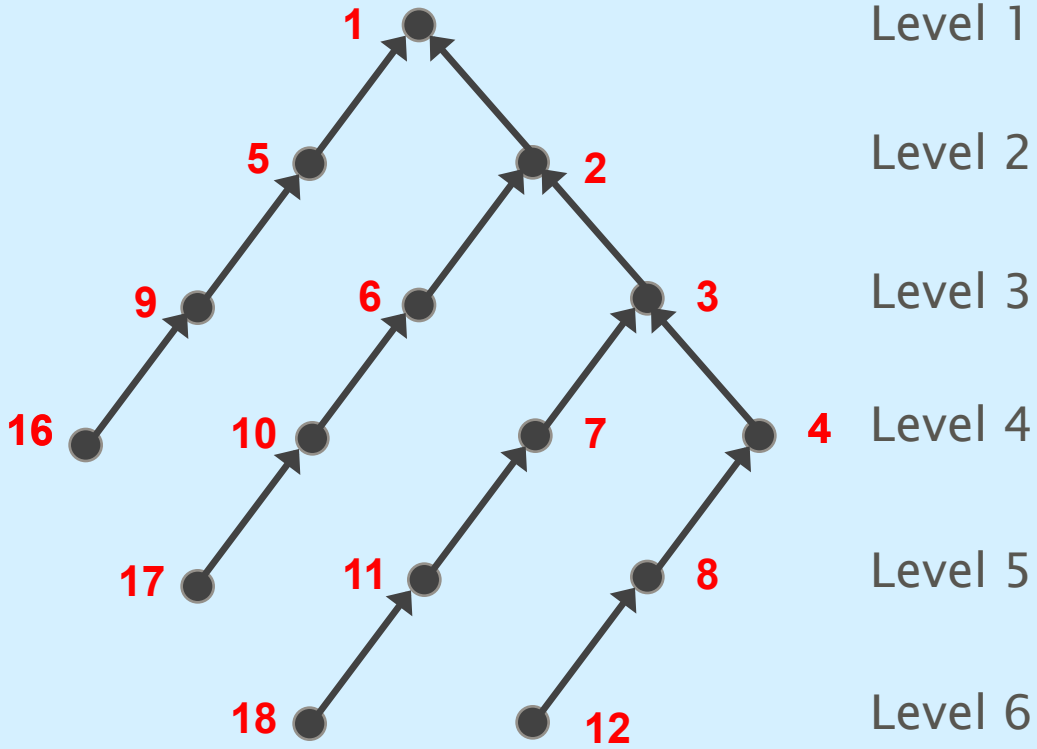




Breadth First Search

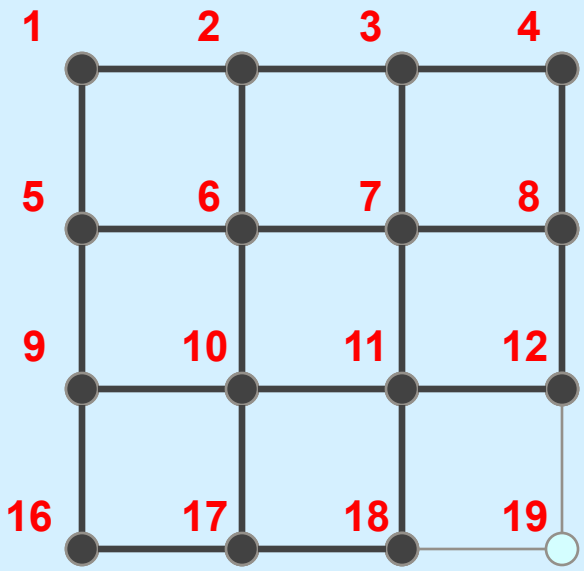
- Level-by-level graph traversal
- Serial complexity: $T_1 = \Theta(m+n)$

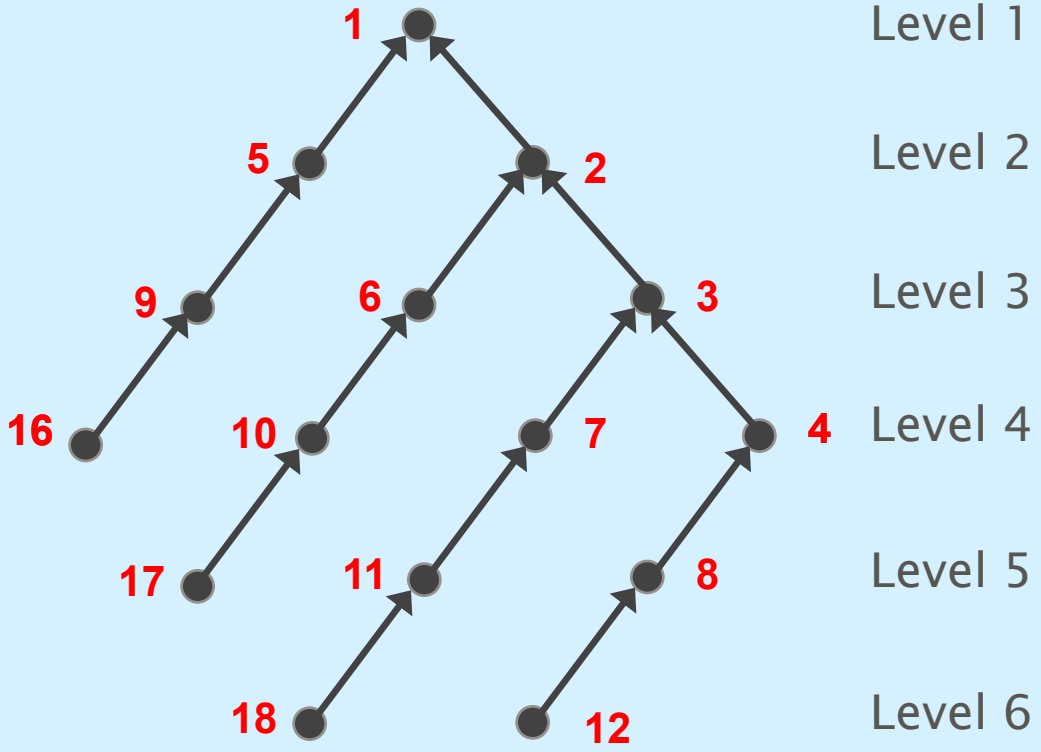




Breadth First Search

- Who is **parent(19)**?
  - If we use a queue for expanding the frontier?
  - Does it actually matter?





Parallel BFS

- Way #1: A custom

```
void BFS(Graph *G, Vertex root
{
    Bag<Vertex> frontier(roo
    while ( ! frontier.isEmpty() )
    {
        cilk::hyperobject< Bag<Vertex> > succbag();
        cilk_for (int i=0; i< frontier.size(); i++)
        {
            for( Vertex v in frontier[i].adjacency() )
            {
                if( ! v.unvisited() )
                    succbag() += v;
            }
        }
        frontier = succbag
    }
}
```

Bag<T> has an associative reduce function that merges two sets

operator+=(Vertex & rhs) also marks rhs "visited"

Parallel BFS

- Way #2: Concurrent writes + List reducer

```
void BFS(Graph *G, Vertex root)
{
    list<Vertex> frontier(root);
    Vertex * parent = new Vertex[n];
    while ( ! frontier.isEmpty() )
    {
        cilk_for (int i=0; i< frontier.size(); i++)
        {
            for( Vertex v in frontier[i].adjacency() )
            {
                if ( ! v.visited() )
                    parent[v] = f
            }
        }
        ...
    }
}
```

An intentional data race

How to generate the new frontier?

Parallel BFS

```
void BFS(Graph *G, Vertex root)
{
    ...
    while ( ! frontier.isEmpty() ) {
        ...
        hyperobject< reducer_list_append<Vertex> >
succlist();
        cilk_for (int i=0; i< frontier.size(); i++)
        {
            for( Vertex v in frontier[i].adjacency() ) {
                if ( parent[v] == frontier[i] )
                {
                    succlist.push_back(v);
                    v.visit();      // Mark "visited"
                }
            }
        }
        frontier = succlist.get
    }
}
```

Run cilk_for loop again

!v.visited() check is not necessary. Why?

Parallel BFS

- Each level is explored with $\Theta(1)$ span
- Graph G has at most d, at least d/2 levels
  - Depending on the location of root
  - d=diameter(G)

Work: $T_1(n) = \Theta(m+n)$

Span: $T_\infty(n) = \Theta(d)$

Parallelism: $$\frac{T_1(n)}{T_\infty(n)} = \Theta((m+n)/d)$$

Parallel BFS Caveats

- d is usually small
- d = lg(n) for scale-free graphs
  - But the degrees are not bounded ☹
- Parallel scaling will be memory-bound
- Lots of burdened parallelism,
  - Loops are skinny
  - Especially to the root and leaves of BFS-tree
- You are not "expected" to parallelize BFS part of Homework #4
  - You may do it for extra credit though ☺