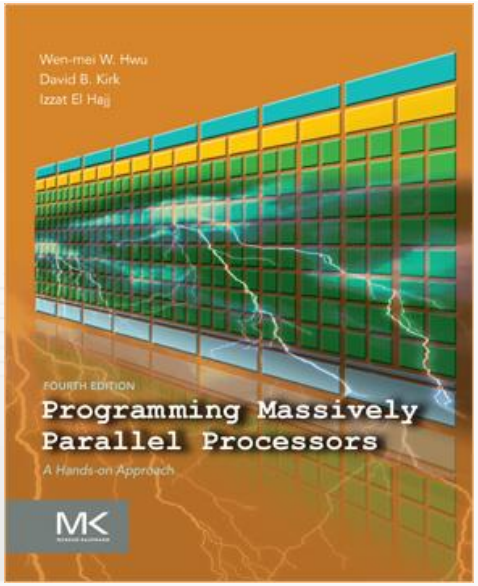
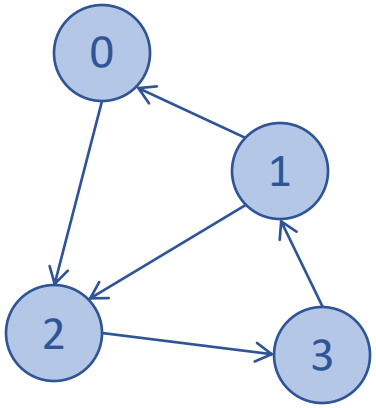
Programming Massively Parallel Processors

A Hands-on Approach

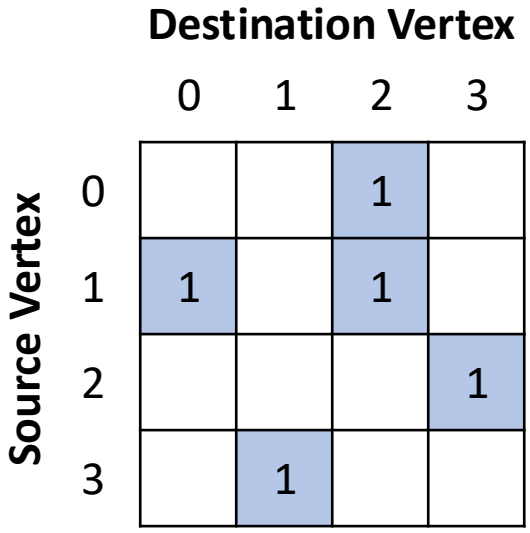
CHAPTER 15 (PART 1) › Graph Traversal



Representing Graphs



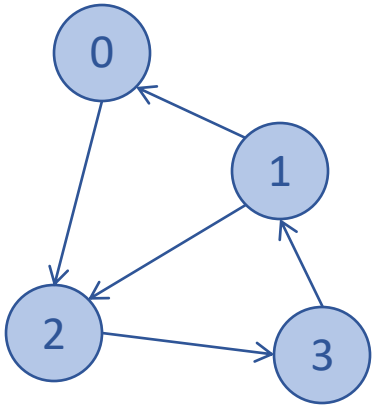
Graph



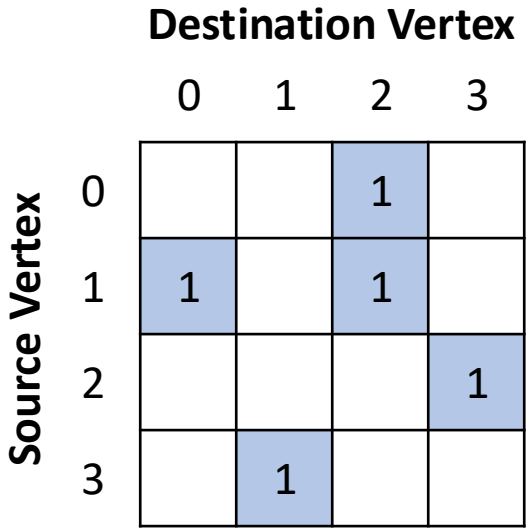
Adjacency Matrix

- Graphs can be represented with adjacency matrices
  - Adjacency matrix is usually very sparse
    - Can use the same storage formats for sparse matrices
  - We will assume unweighted graphs
    - Nonzeros all ones so no need to store their values

COO, CSR, and CSC Representation



Graph



Adjacency Matrix

CSR

srcPtrs	0	1	3	4	5
dst	2	0	2	3	1

CSC

dstPtrs	0	1	2	4	5
src	1	3	0	1	2

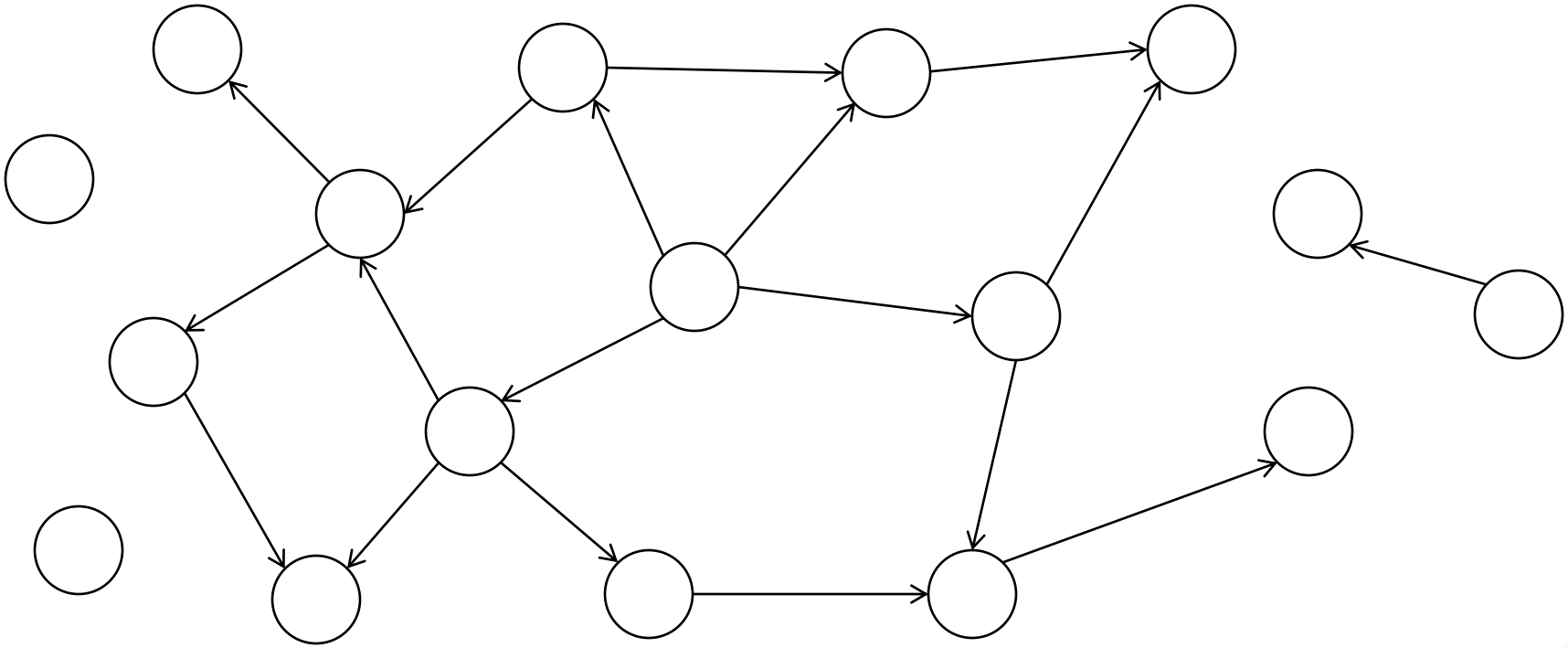
COO

src	0	1	1	2	3
dst	2	0	2	3	1

Approaches to Parallelizing Graph Traversal

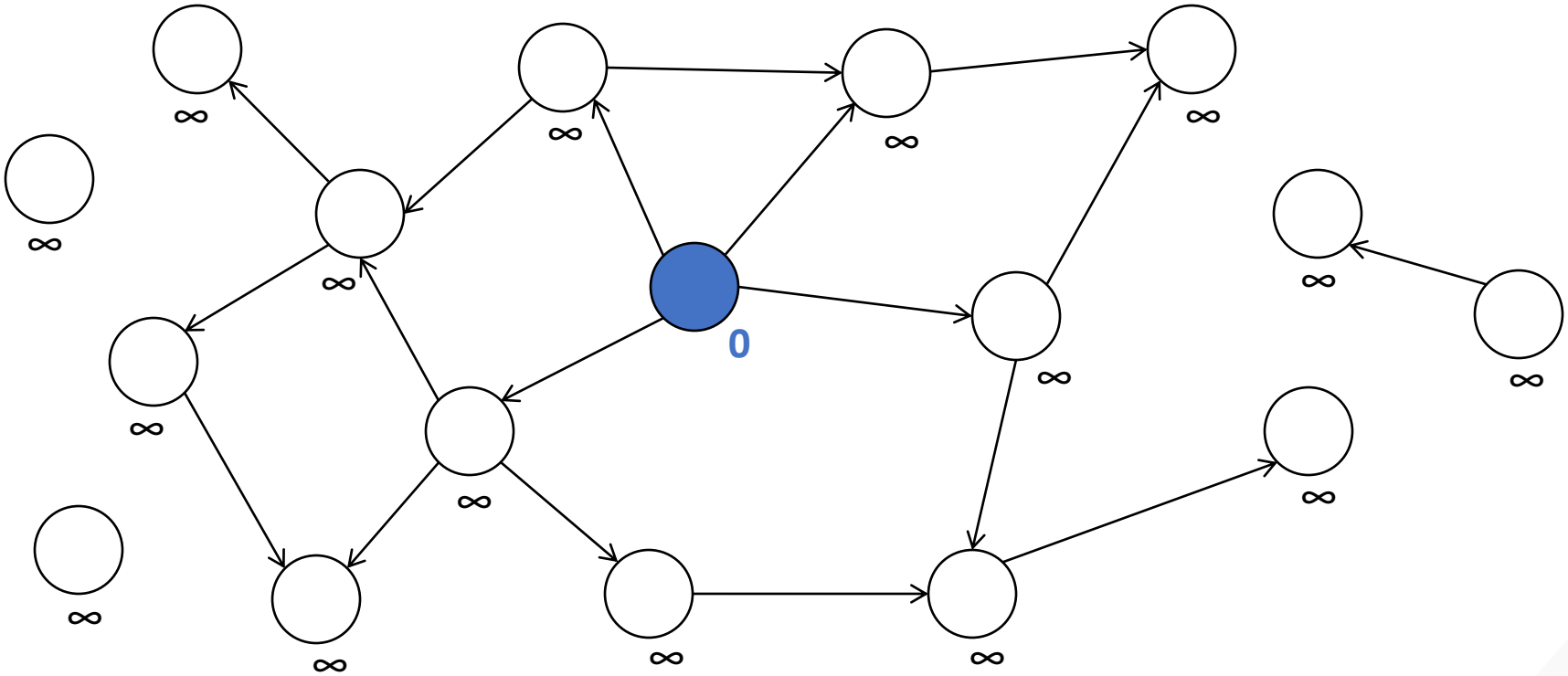
- Vertex-centric
  - Assign a thread to do something for each vertex
  - Typically use CSR/CSC
    - Given a vertex, easy to find all its incoming or outgoing edges
    - Might also use other formats (e.g., ELL, JDS) as optimization
- Edge-centric
  - Assign a thread to do something for each edge
  - Typically use COO
    - Given an edge, easy to find its source and destination vertices
- Hybrid
  - Example: Given an edge, need neighbors of the source vertices and neighbors of the destination vertices
  - Use both COO and CSR/CSC

Breadth First Search (BFS)



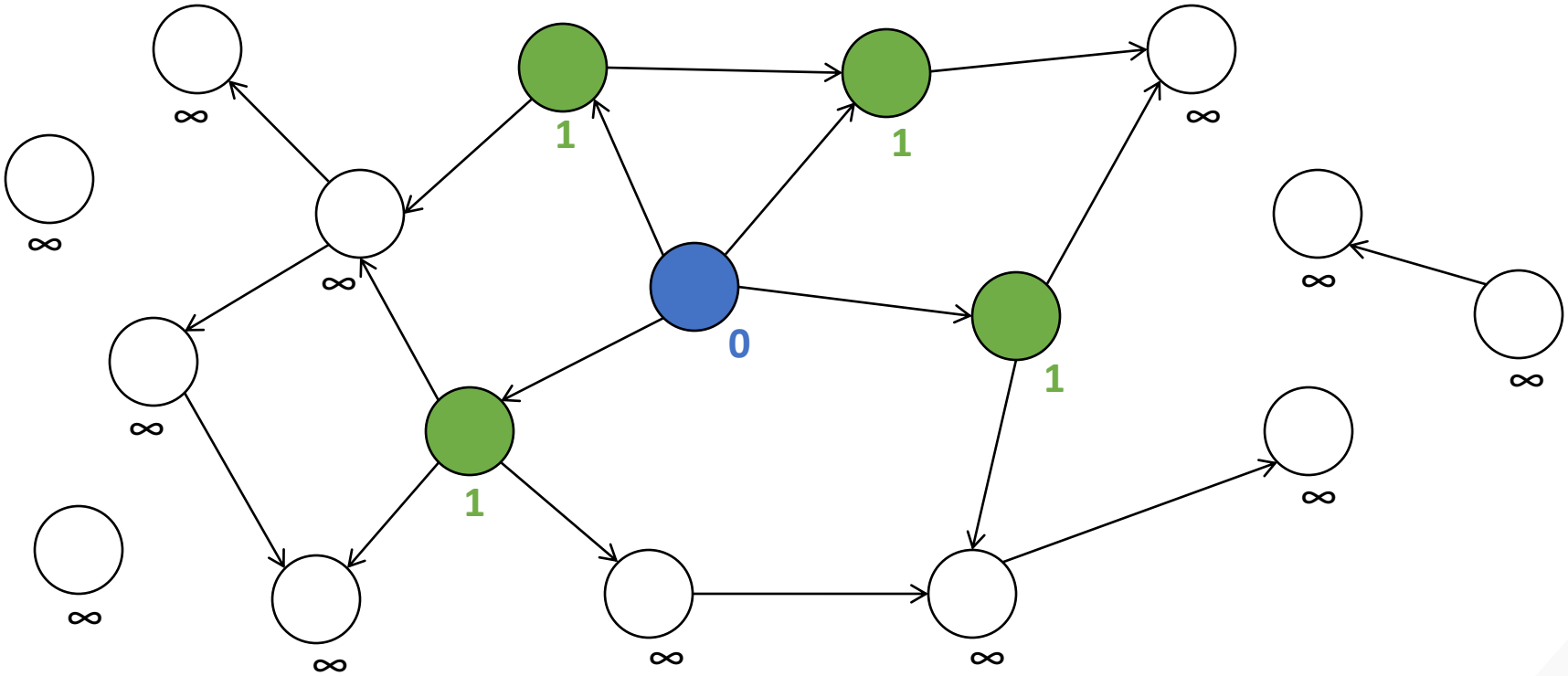
Objective: find the distance (level) of each vertex from some source vertex

Breadth First Search (BFS)



Objective: find the distance (level) of each vertex from some source vertex

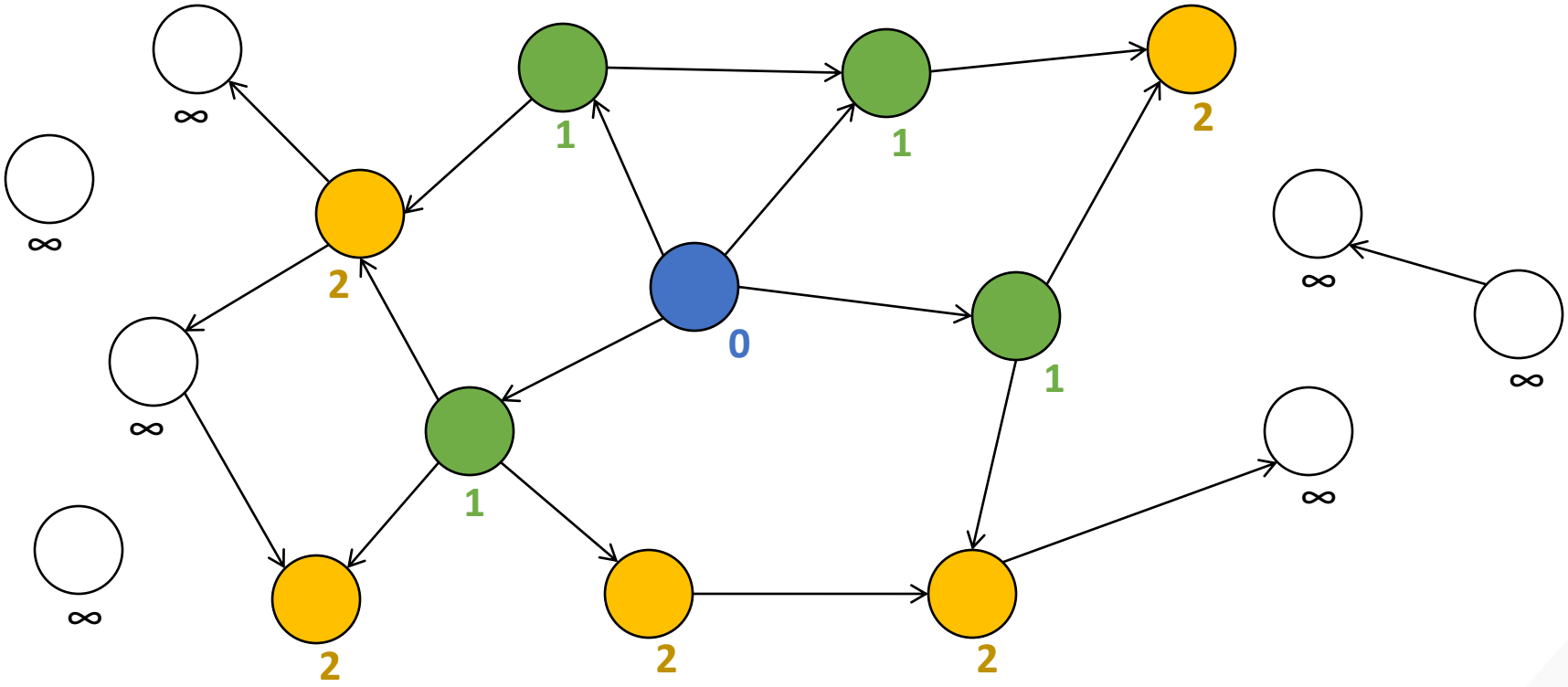
Breadth First Search (BFS)



Level 1

Objective: find the distance (level) of each vertex from some source vertex

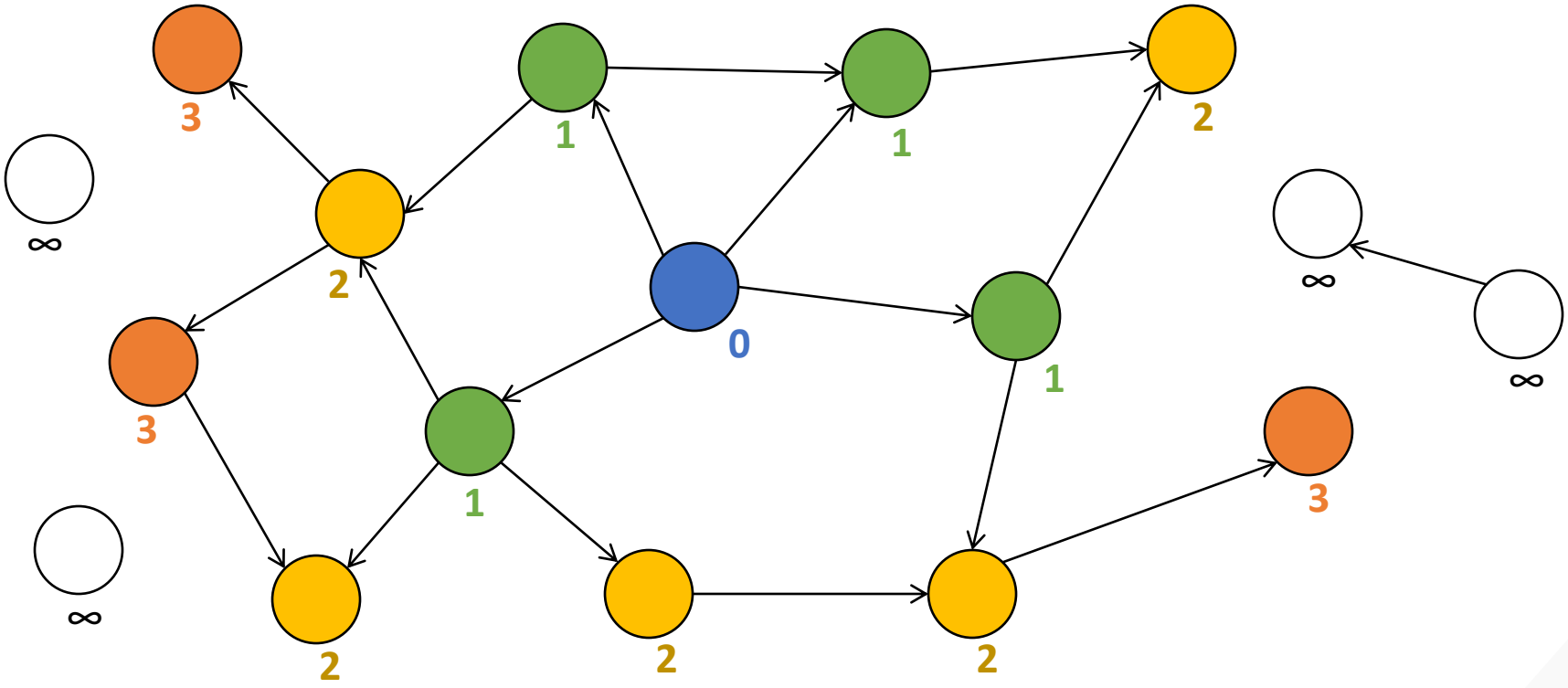
Breadth First Search (BFS)



Level 2

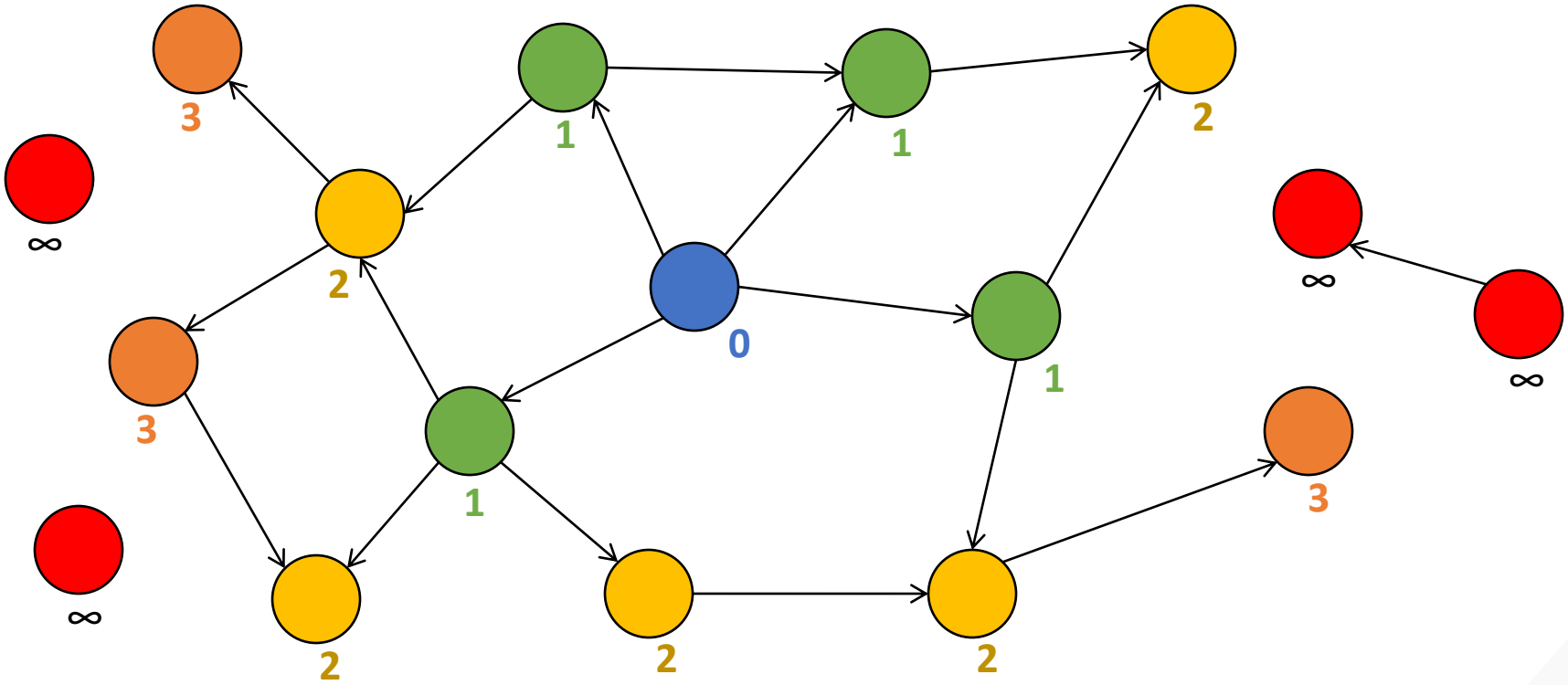
Objective: find the distance (level) of each vertex from some source vertex

Breadth First Search (BFS)



Objective: find the distance (level) of each vertex from some source vertex

Breadth First Search (BFS)



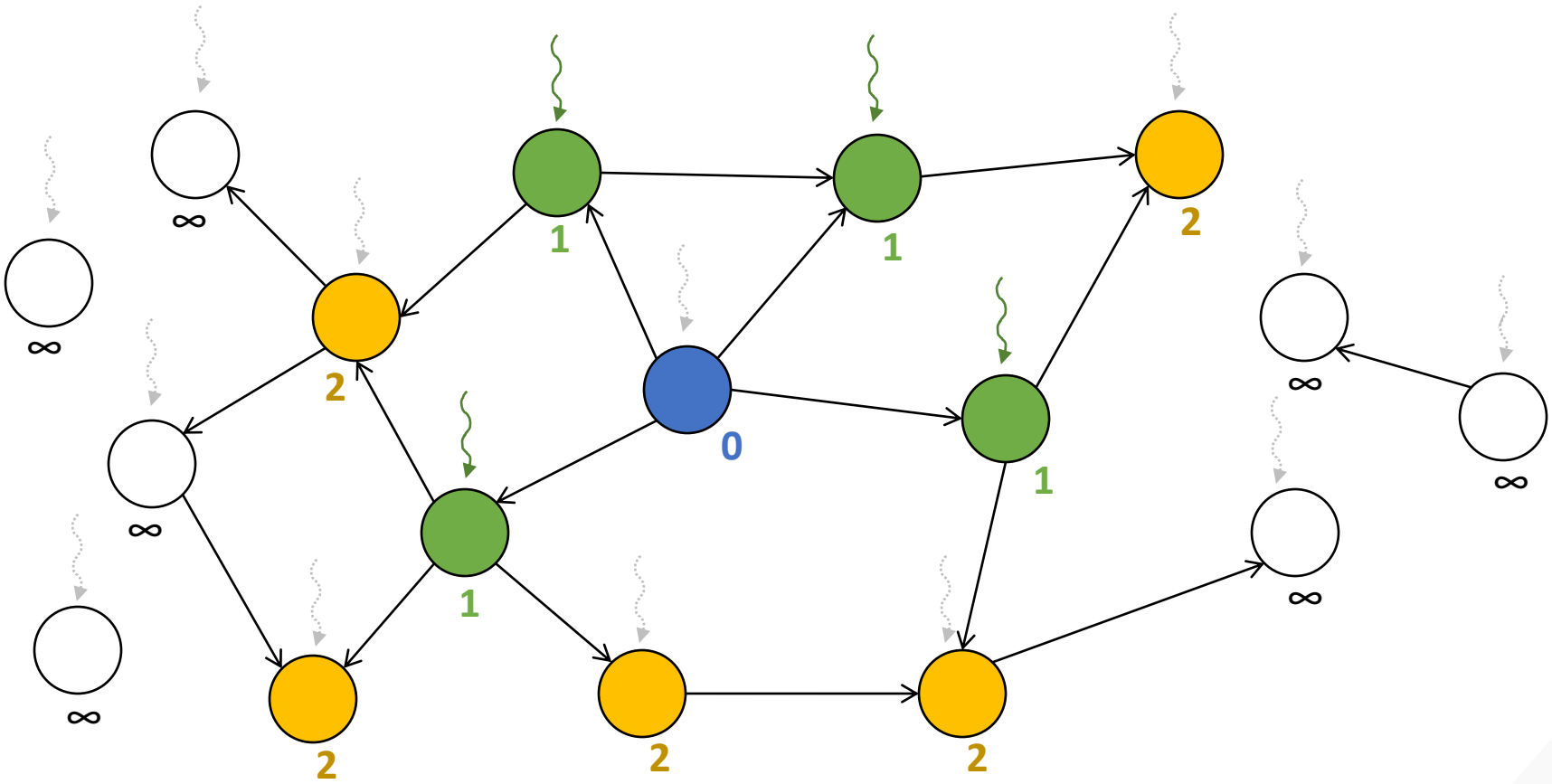
Unreachable

Objective: find the distance (level) of each vertex from some source vertex

Approaches to Parallelizing BFS

- Vertex-centric (two versions)
  - **Push:** For every vertex, if it was in the previous level, add all its unvisited outgoing neighbors to the current level
    - i.e., a vertex pushes information to its outgoing neighbors

Vertex-Centric BFS (Push)



Level 1 => Level 2

Threads whose vertices are in **level 1** mark their unvisited neighbors as being in **level 2**

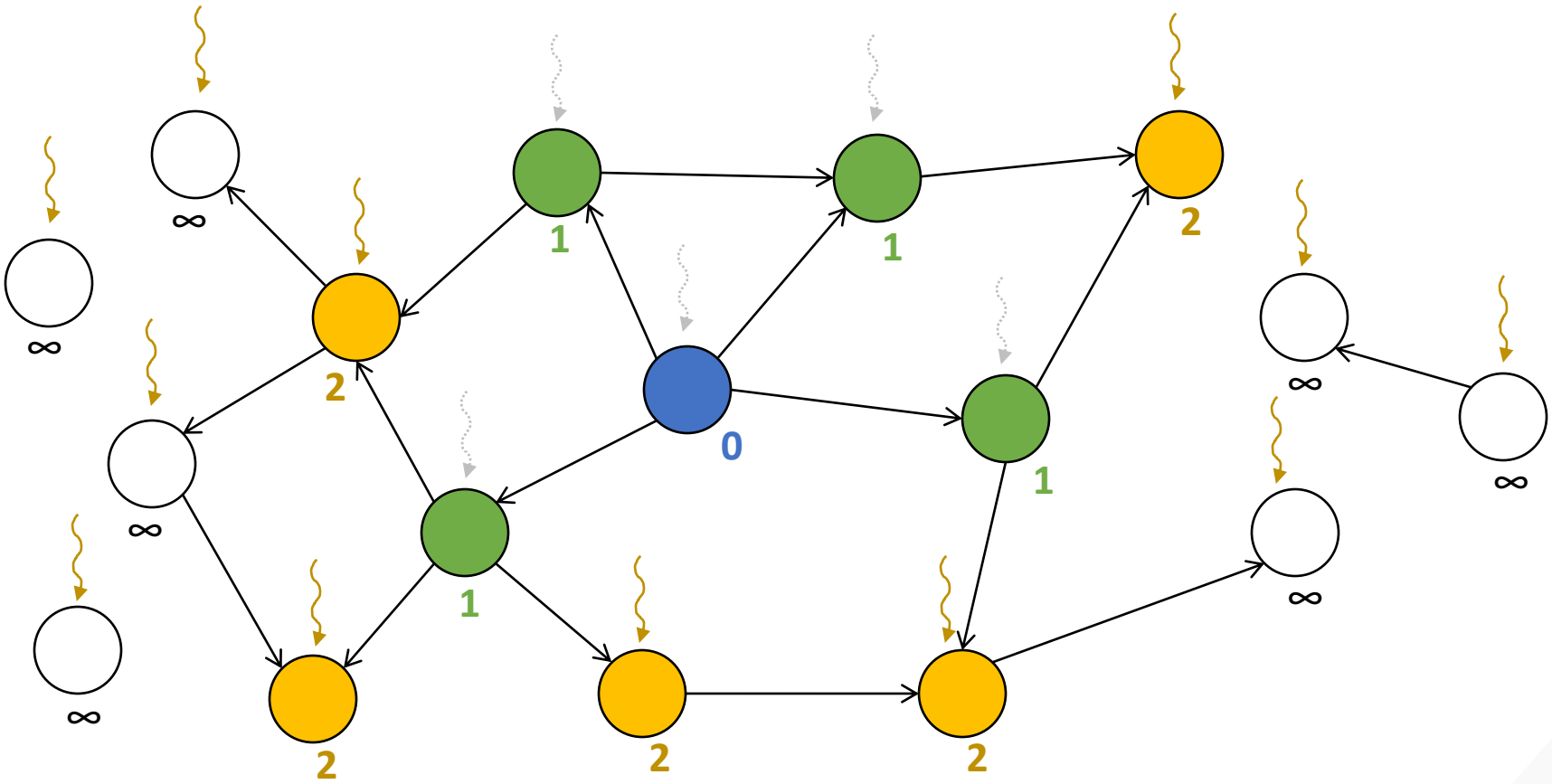
Vertex-Centric BFS Code (Push)

```
__global__ void bfs_kernel(CSRGraph csrGraph, unsigned int* level, unsigned int* newVertexVisited,
                                                                    unsigned int currLevel) {
    unsigned int vertex = blockIdx.x*blockDim.x + threadIdx.x;
    if(vertex < csrGraph.numVertices) {
        if(level[vertex] == currLevel - 1) {
            for(unsigned int edge = csrGraph.srcPtrs[vertex]; edge < csrGraph.srcPtrs[vertex + 1]; ++edge) {
                unsigned int neighbor = csrGraph.dst[edge];
                if(level[neighbor] == UINT_MAX) { // Neighbor not previously visited
                    level[neighbor] = currLevel;
                    *newVertexVisited = 1;
                }
            }
        }
    }
}
```

Approaches to Parallelizing BFS

- Vertex-centric (two versions)
  - **Push:** For every vertex, if it was in the previous level, add all its unvisited outgoing neighbors to the current level
    - i.e., a vertex pushes information to its outgoing neighbors
  - **Pull:** For every vertex, if it has not been visited, if any of its incoming neighbors are in the previous level, add it to the current level
    - i.e., a vertex pulls information from its incoming neighbors

Vertex-Centric BFS (Pull)



Level 1 => Level 2

Threads whose vertices are unvisited mark their vertices as being in **level 2** if any of their neighbors are in **level 1**

Vertex-Centric BFS Code (Pull)

```
__global__ void bfs_kernel(CSCGraph cscGraph, unsigned int* level, unsigned int* newVertexVisited,
                                                                    unsigned int currLevel) {
    unsigned int vertex = blockIdx.x*blockDim.x + threadIdx.x;
    if(vertex < cscGraph.numVertices) {
        if(level[vertex] == UINT_MAX) { // Vertex not previously visited
            for(unsigned int edge = cscGraph.dstPtrs[vertex]; edge < cscGraph.dstPtrs[vertex + 1]; ++edge) {
                unsigned int neighbor = cscGraph.src[edge];
                if(level[neighbor] == currLevel - 1) {
                    level[vertex] = currLevel;
                    *newVertexVisited = 1;
                    break;
                }
            }
        }
    }
}
```

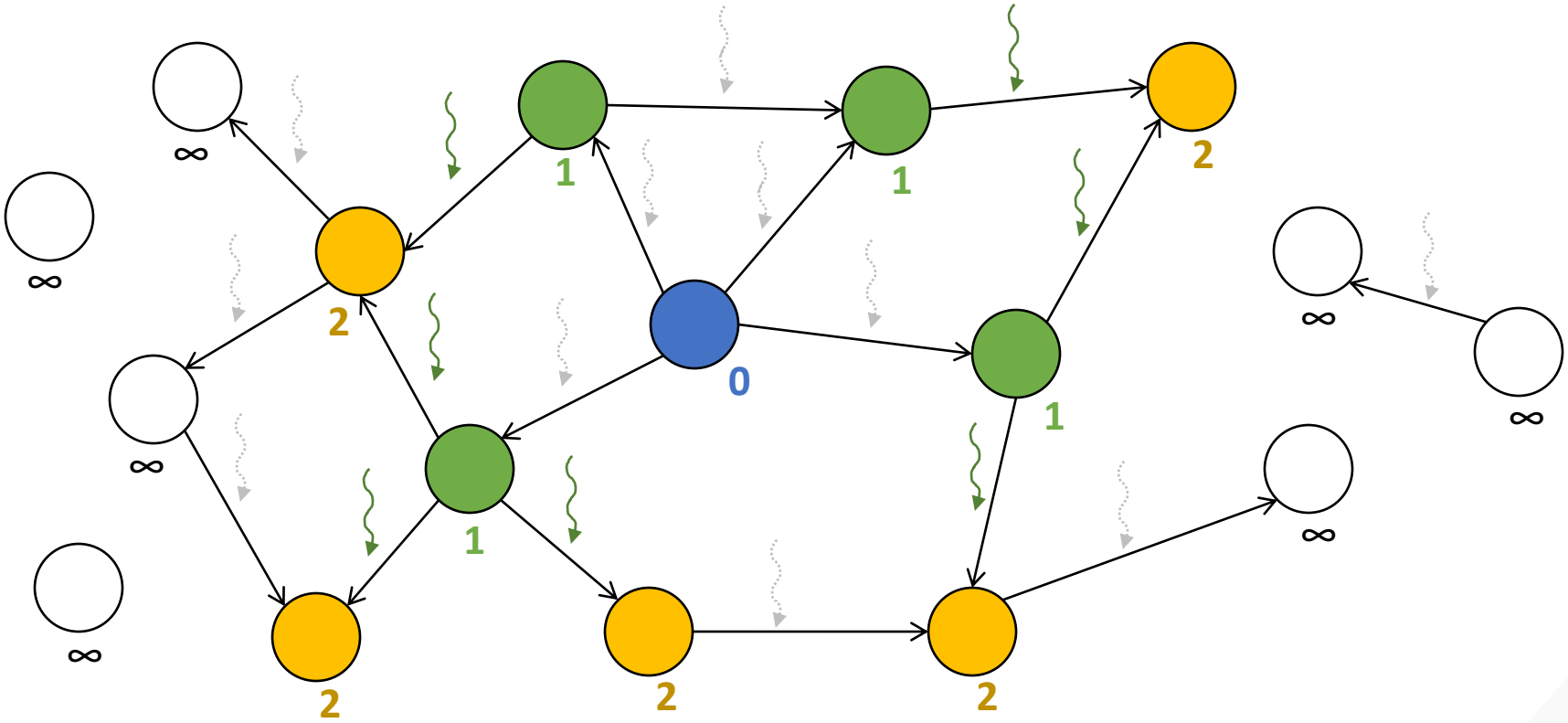
Approaches to Parallelizing BFS

- Vertex-centric (two versions)
  - **Push:** For every vertex, if it was in the previous level, add all its unvisited outgoing neighbors to the current level
    - i.e., a vertex pushes information to its outgoing neighbors
  - **Pull:** For every vertex, if it has not been visited, if any of its incoming neighbors are in the previous level, add it to the current level
    - i.e., a vertex pulls information from its incoming neighbors
  - **Direction-optimized:** Start with push then switch to pull
    - Pull is inefficient at the beginning because most vertices will search all neighbors and not find any that are in the previous level

Approaches to Parallelizing BFS

- Vertex-centric (two versions)
  - **Push:** For every vertex, if it was in the previous level, add all its unvisited outgoing neighbors to the current level
    - i.e., a vertex pushes information to its outgoing neighbors
  - **Pull:** For every vertex, if it has not been visited, if any of its incoming neighbors are in the previous level, add it to the current level
    - i.e., a vertex pulls information from its incoming neighbors
  - **Direction-optimized:** Start with push then switch to pull
    - Pull is inefficient at the beginning because most vertices will search all neighbors and not find any that are in the previous level
- Edge-centric
  - For every edge, if its source vertex was in the previous level, add its destination vertex to the current level

Edge-Centric BFS



Level 1 => Level 2

Threads whose edge's source vertex is in **level 1** and whose edge's destination vertex is unvisited mark their edge's destination vertex as being in **level 2**

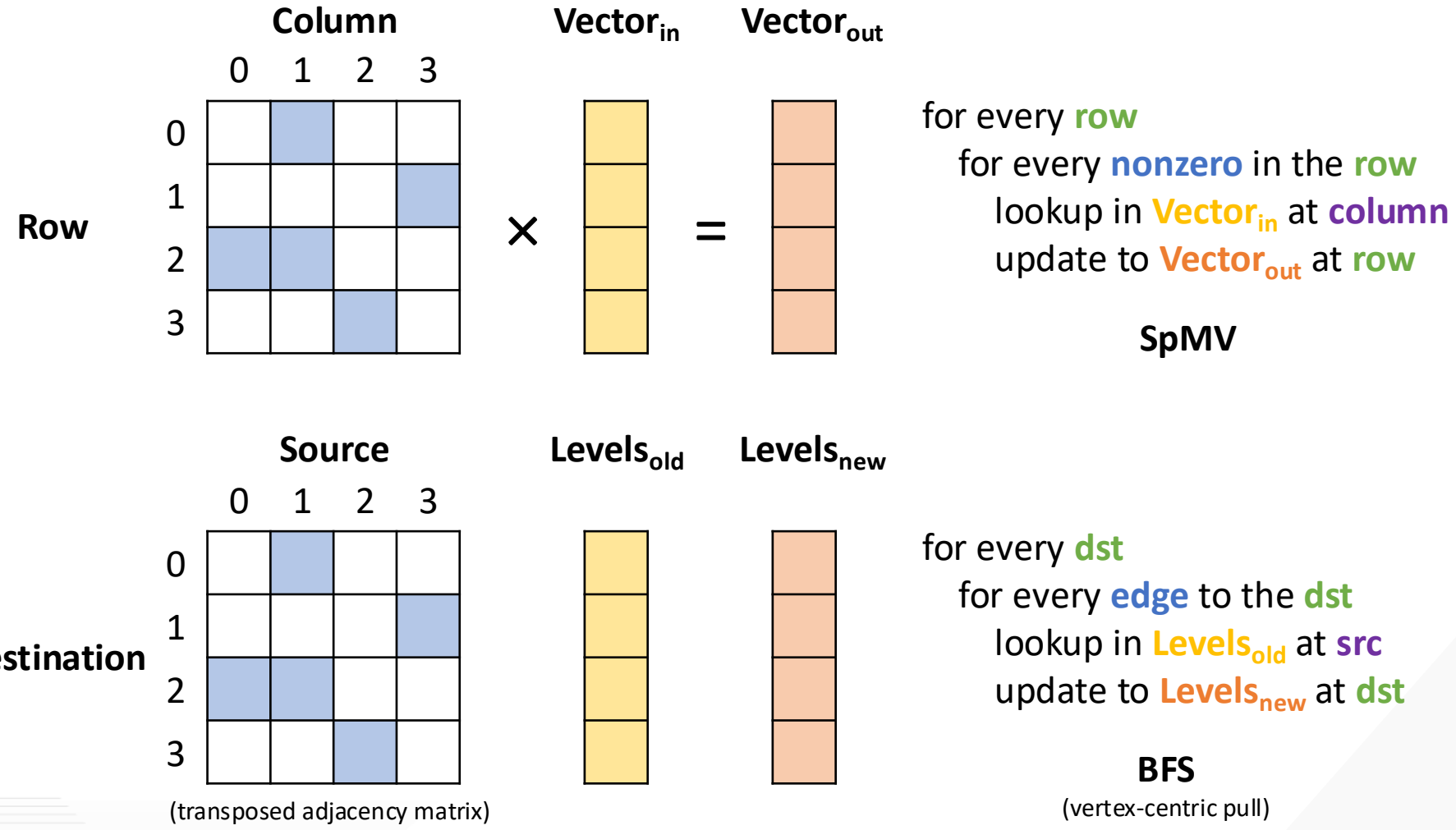
Edge-Centric BFS Code

```
__global__ void bfs_kernel(COOGraph cooGraph, unsigned int* level, unsigned int* newVertexVisited,
                                                                    unsigned int currLevel) {
    unsigned int edge = blockIdx.x*blockDim.x + threadIdx.x;
    if(edge < cooGraph.numEdges) {
        unsigned int vertex = cooGraph.src[edge];
        unsigned int neighbor = cooGraph.dst[edge];
        if(level[vertex] == currLevel - 1) {
            if(level[neighbor] == UINT_MAX) { // Destination vertex not previously visited
                level[neighbor] = currLevel;
                *newVertexVisited = 1;
            }
        }
    }
}
```

Dataset Implications

- The best parallelization approach depends on the structure of the graph
- The vertex-centric pull and the edge-centric approaches are better on high-degree graphs
  - e.g., social network graphs with celebrities
  - Better at dealing with load imbalance
- The vertex-centric push approach is better on low-degree graphs
  - e.g., map of roads in a geographical area
  - Only promising threads will iterate through neighbors

Similarity Between BFS and SpMV



Vertex-centric push and edge-centric also correspond to other ways of performing SpMV

Linear Algebraic Formulation of Graph Problems

- With a few tweaks, BFS can be formulated exactly as SpMV

- Many graph problems can be formulated in terms of sparse linear algebra computations
  - Advantage: leverage mature and well-optimized parallel libraries for high performance sparse linear algebra
  - Disadvantage: not always the most efficient way to solve the problem

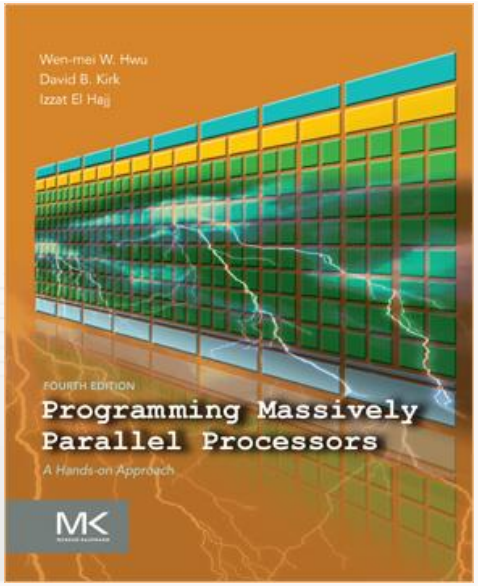
References

- Wen-mei W. Hwu, David B. Kirk, and Izzat El Hajj. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2022.

Programming Massively Parallel Processors

A Hands-on Approach

CHAPTER 15 (PART 2) › Graph Traversal

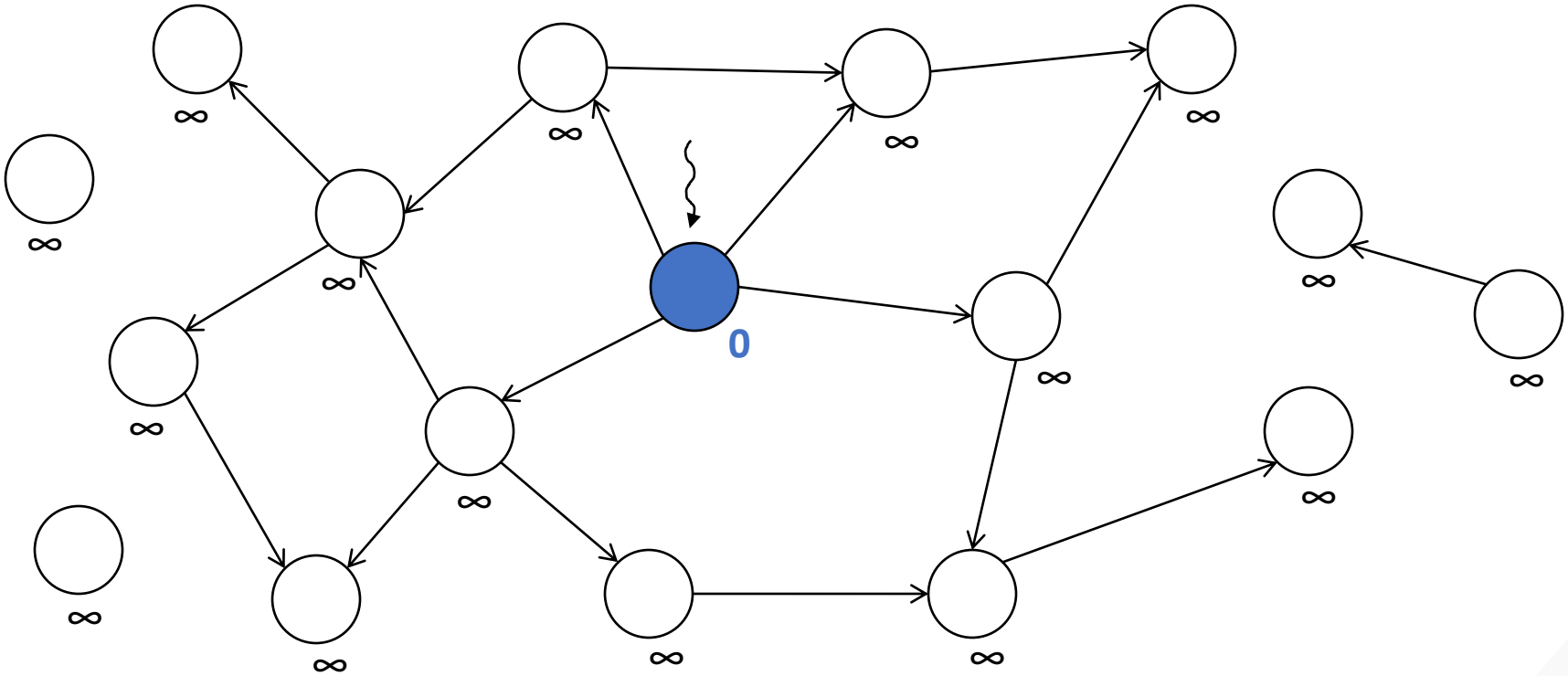


Redundant Work

- Approaches discussed so far check every vertex or edge on every iteration for relevance
  - Strengths: easy to implement, highly parallel, no synchronization across threads
  - Weaknesses: a lot of unnecessary threads/work
    - Many threads will find that their vertex or edge are not relevant for this iteration and just exit

- Alternative for reducing redundancy:
  - Objective: only check the vertices that are part of the previous level
  - Approach: each level adds the vertices it visits to a list for the next level to process
    - Vertices added at each level form that level's *frontier*
  - Overhead: synchronization across threads to add to a shared list

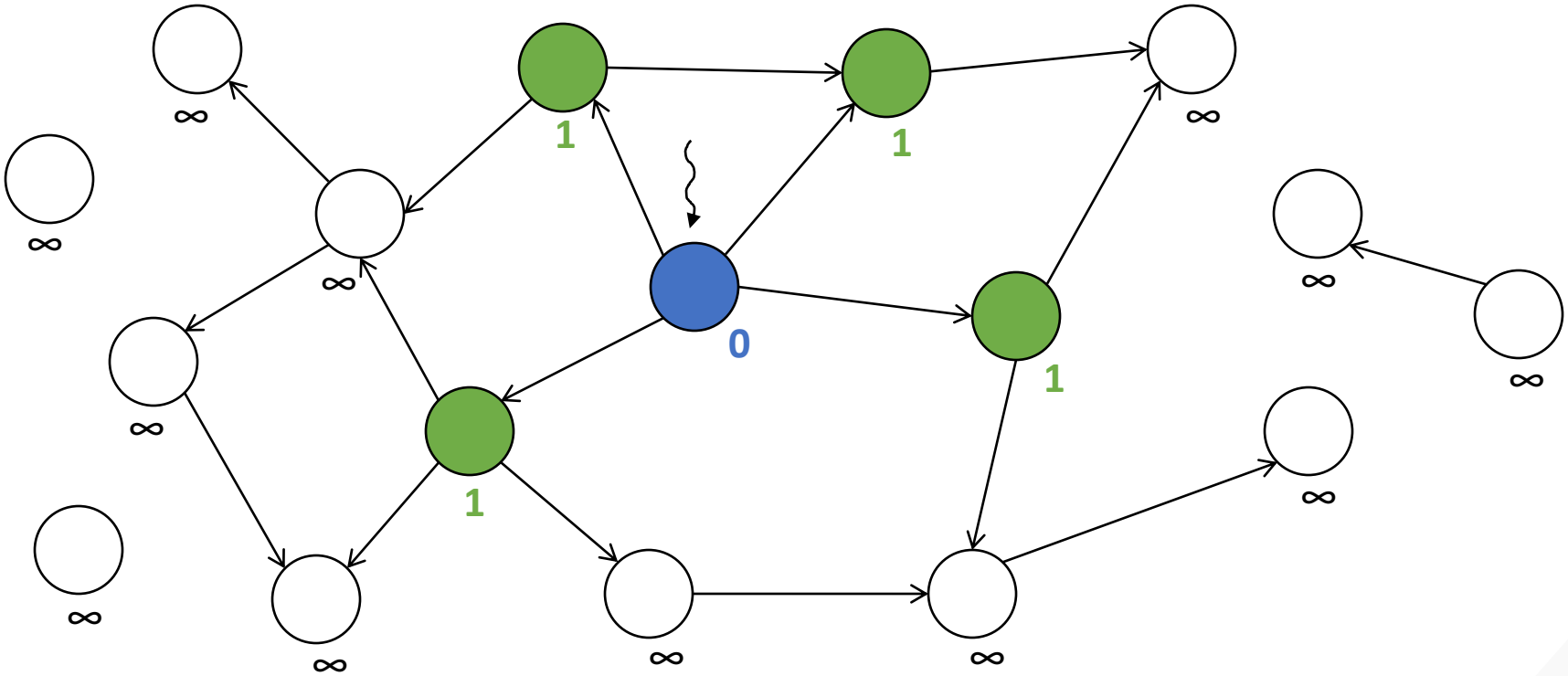
Vertex-Centric Frontier-Based BFS



Level 0 => Level 1

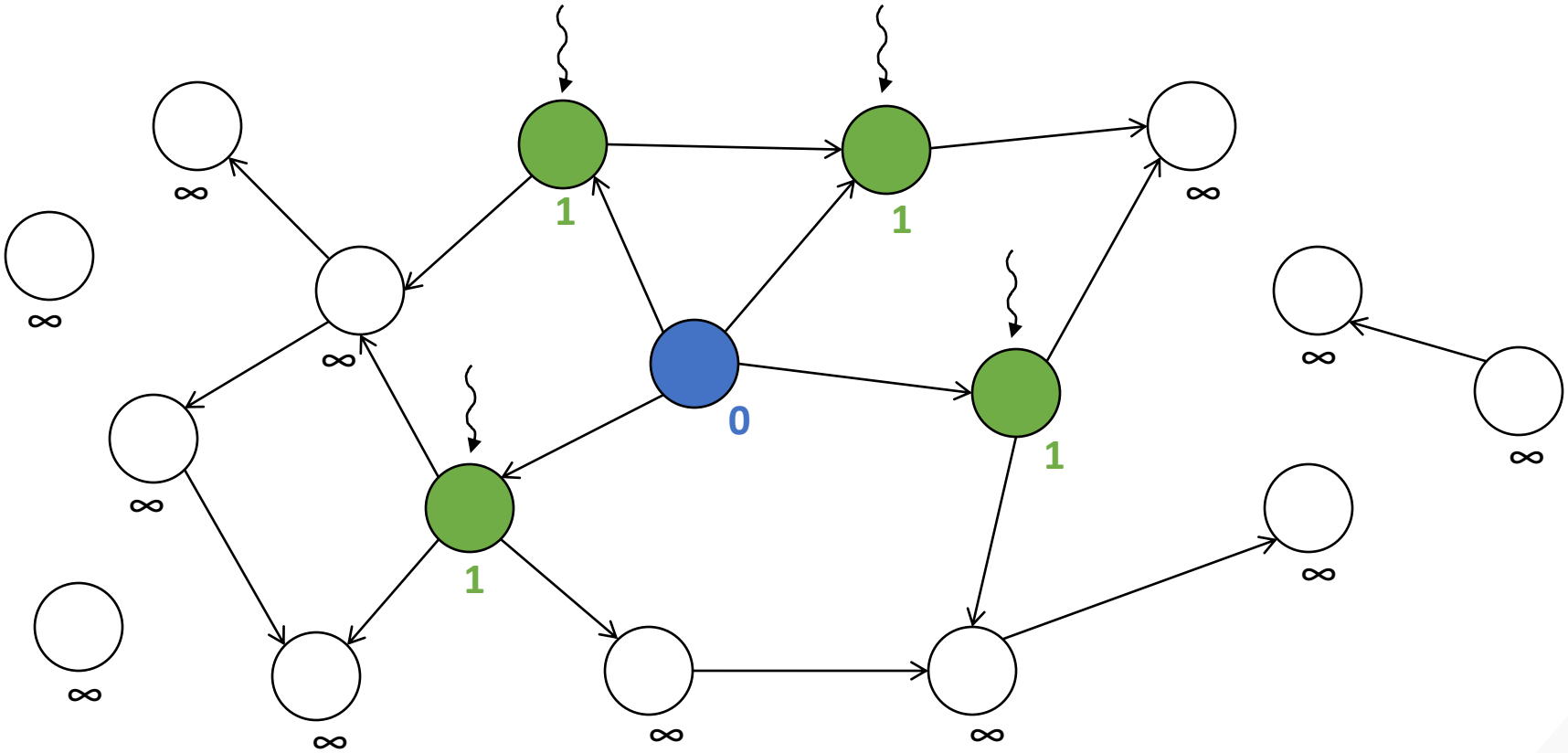
Previous Frontier:

Vertex-Centric Frontier-Based BFS

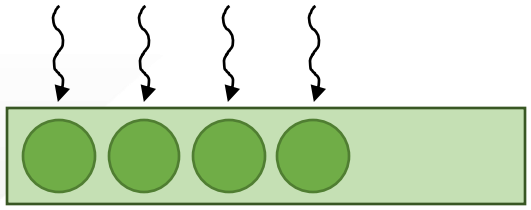


Level 0 => Level 1

Vertex-Centric Frontier-Based BFS

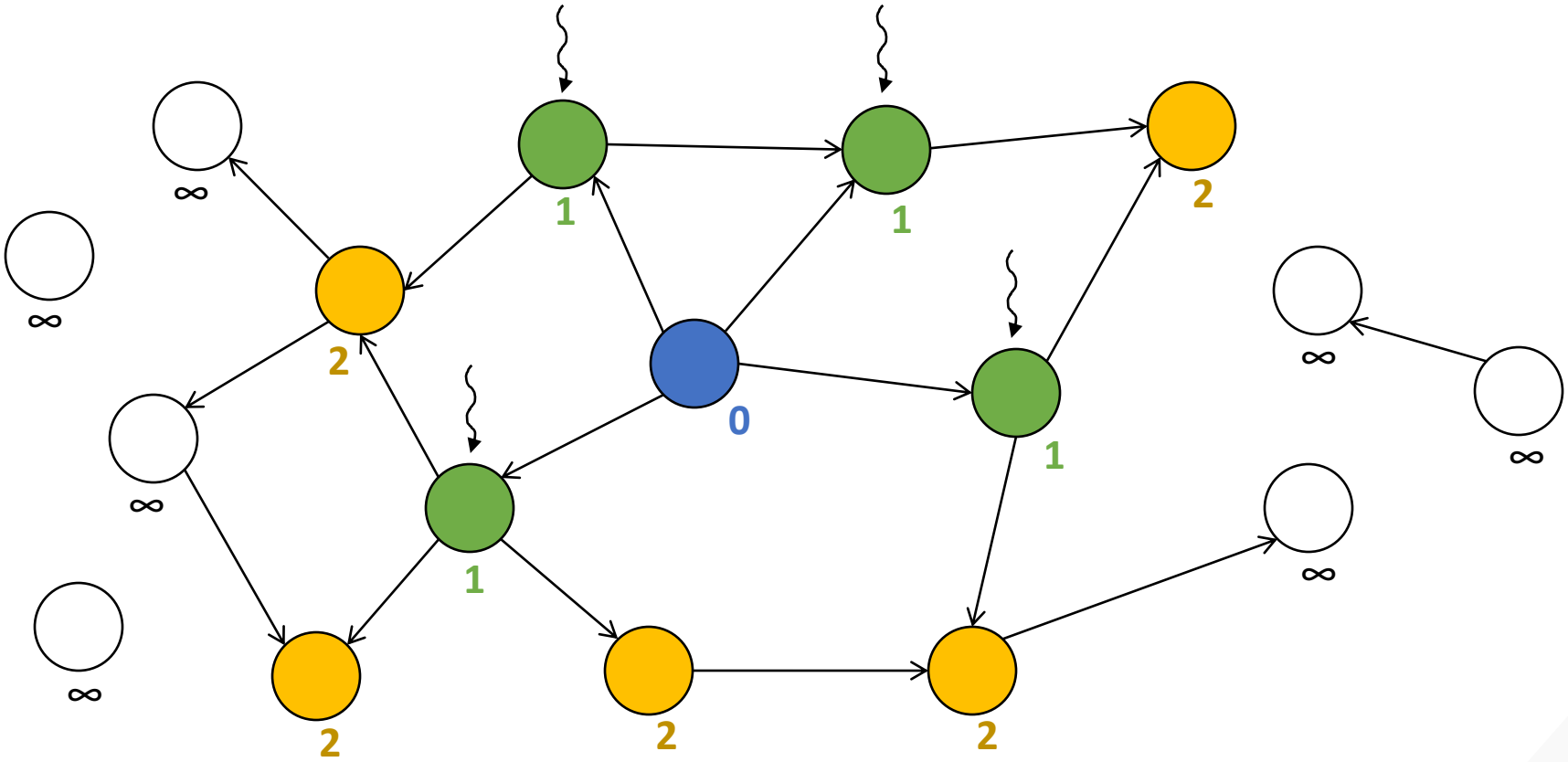


Level 1 => Level 2



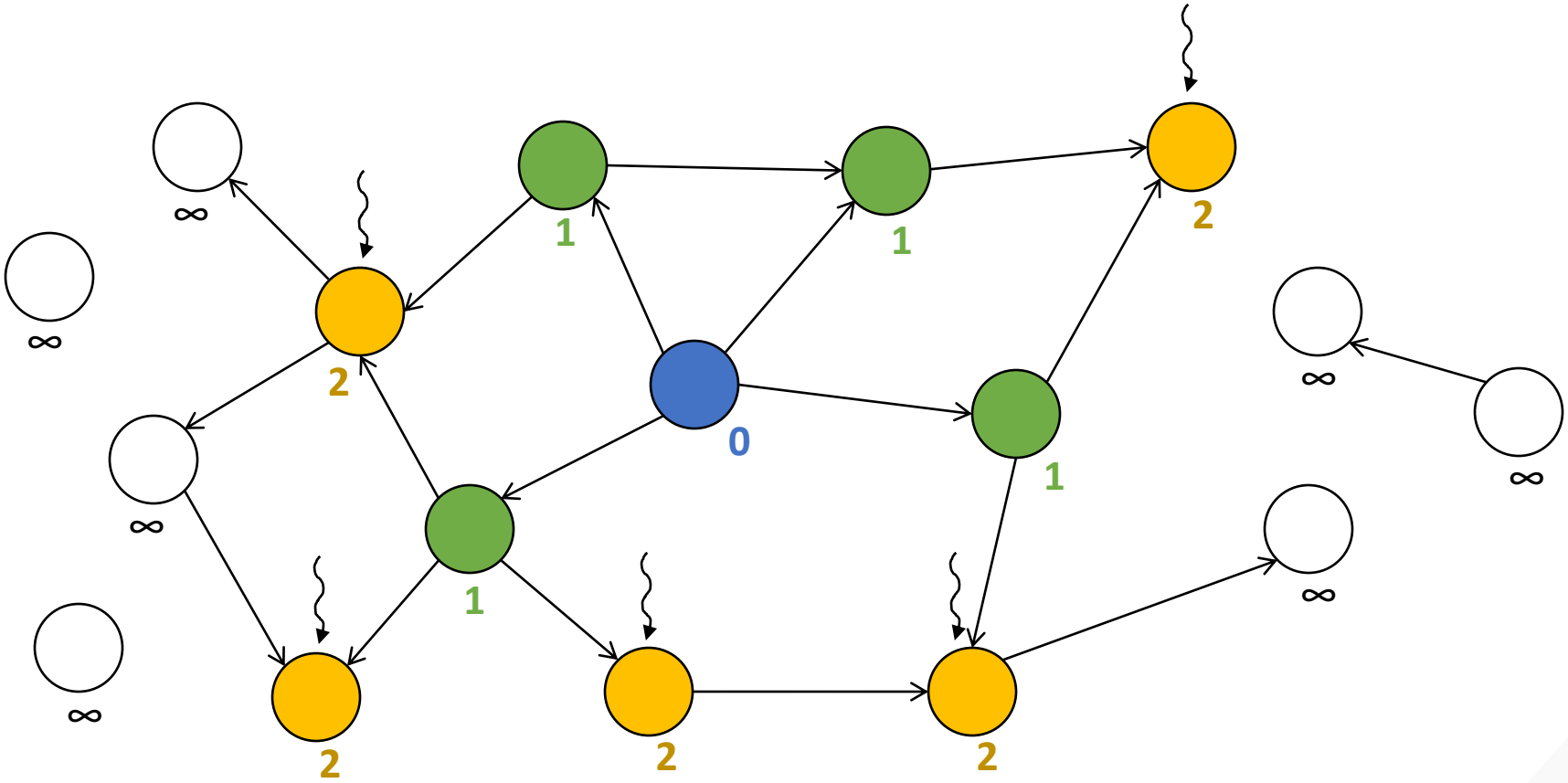
Previous Frontier:

Vertex-Centric Frontier-Based BFS

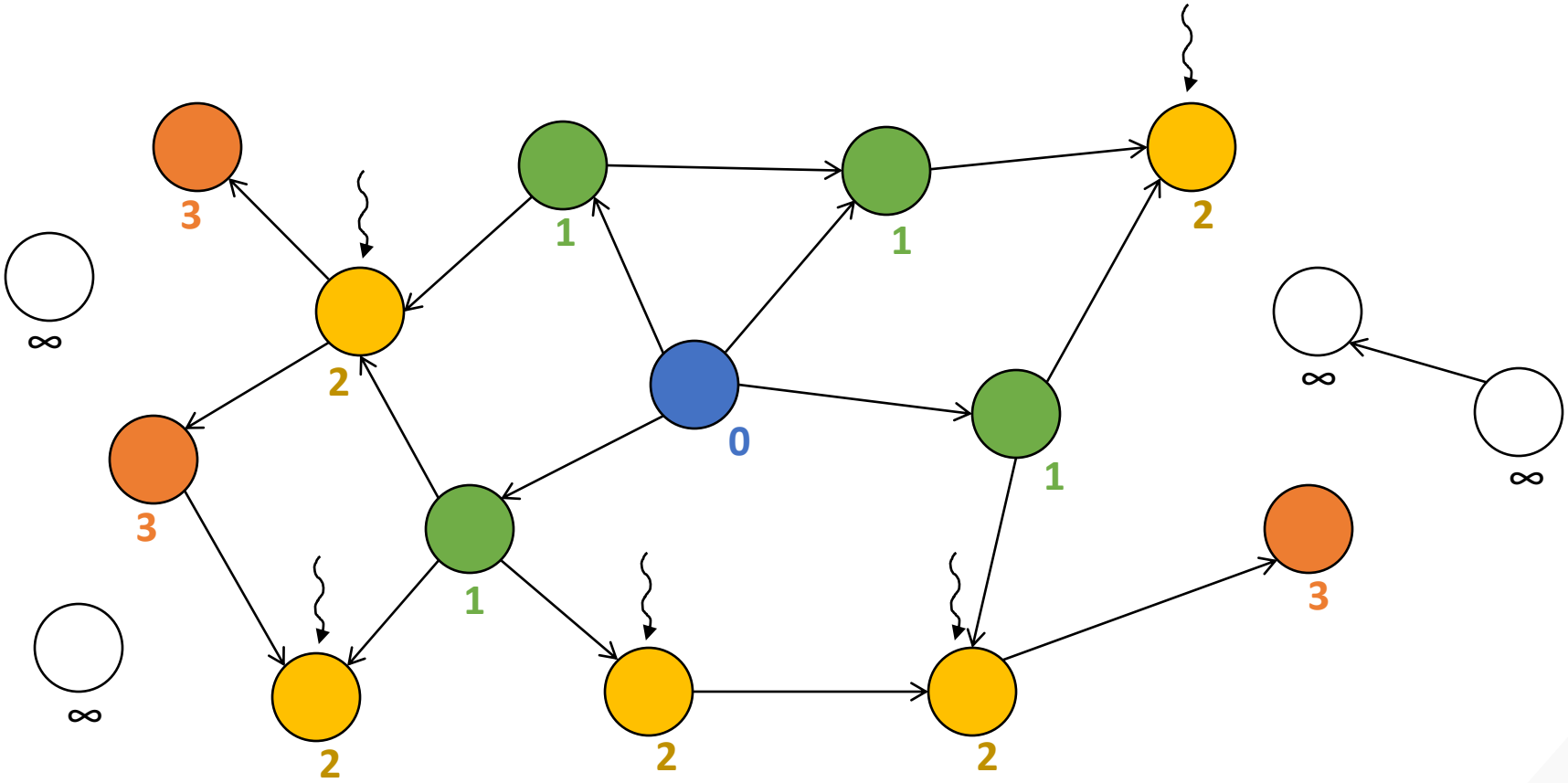


Level 1 => Level 2

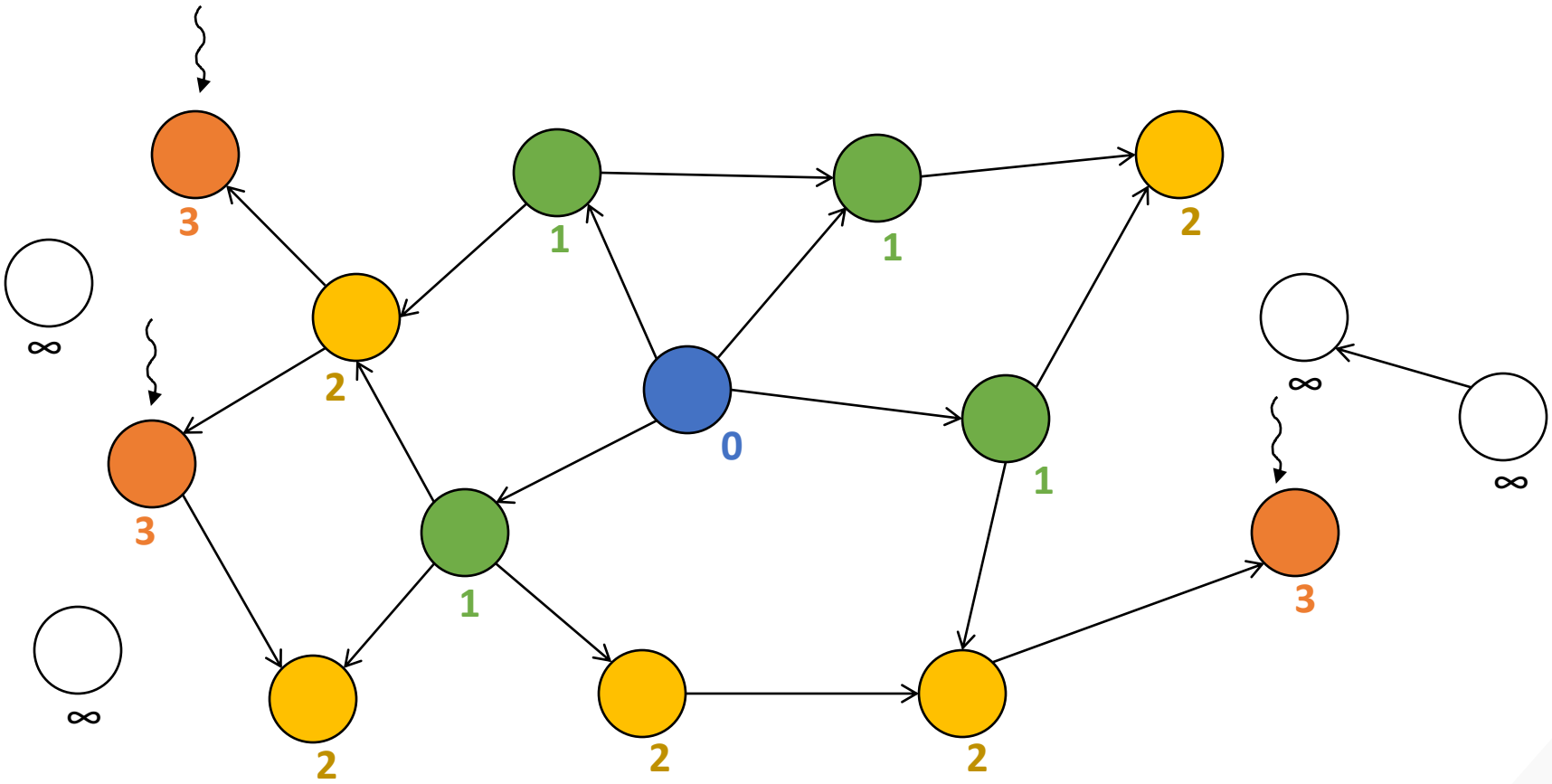
Vertex-Centric Frontier-Based BFS



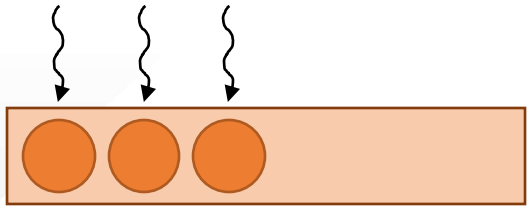
Vertex-Centric Frontier-Based BFS



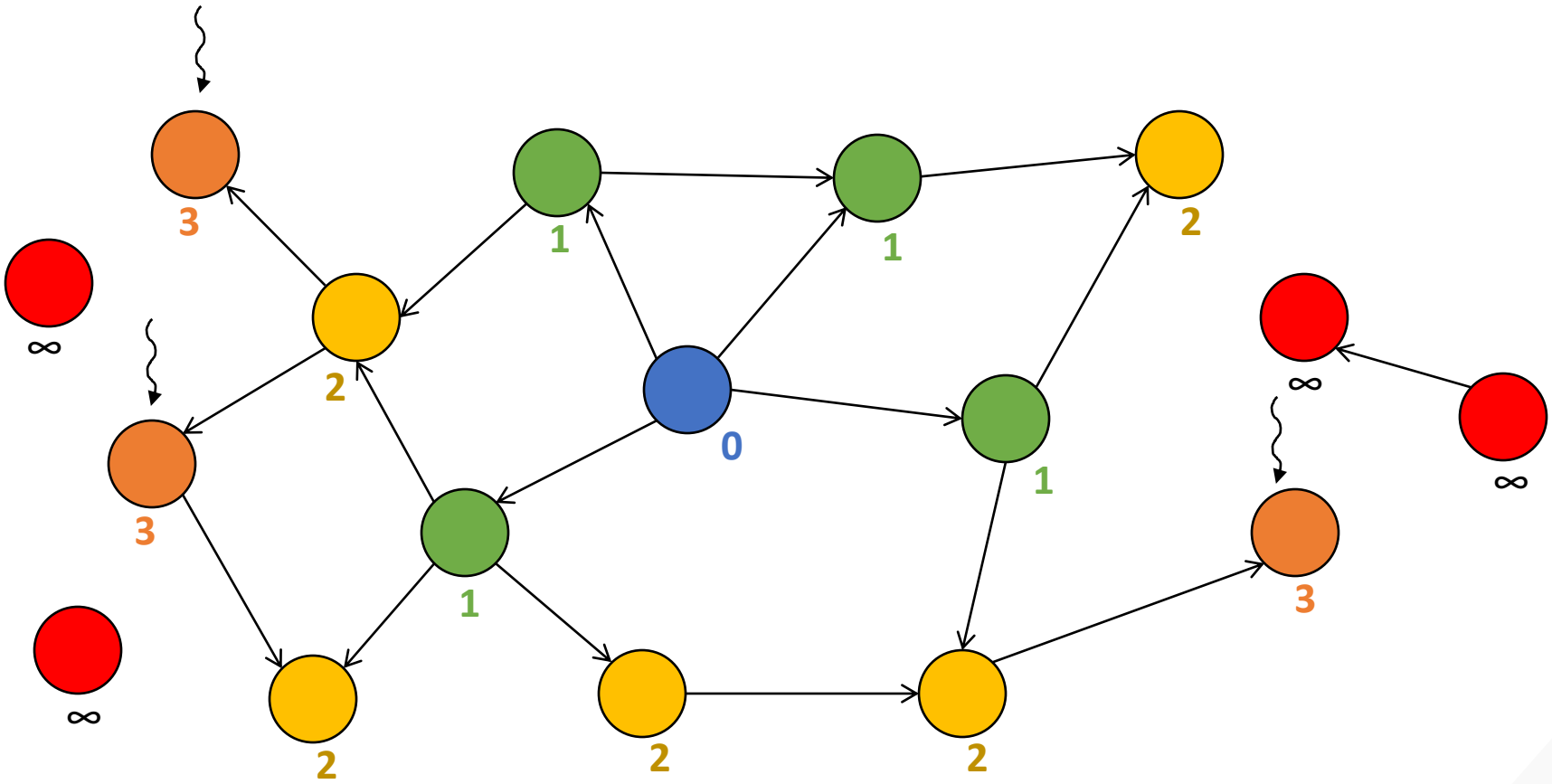
Vertex-Centric Frontier-Based BFS



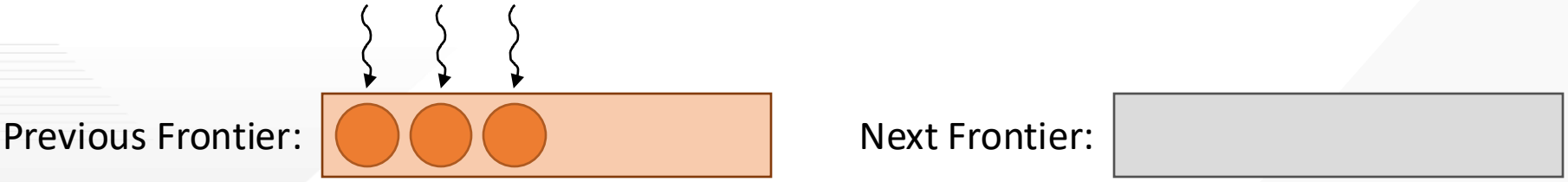
Level 3 => Level 4



Vertex-Centric Frontier-Based BFS



Level 3 => Level 4



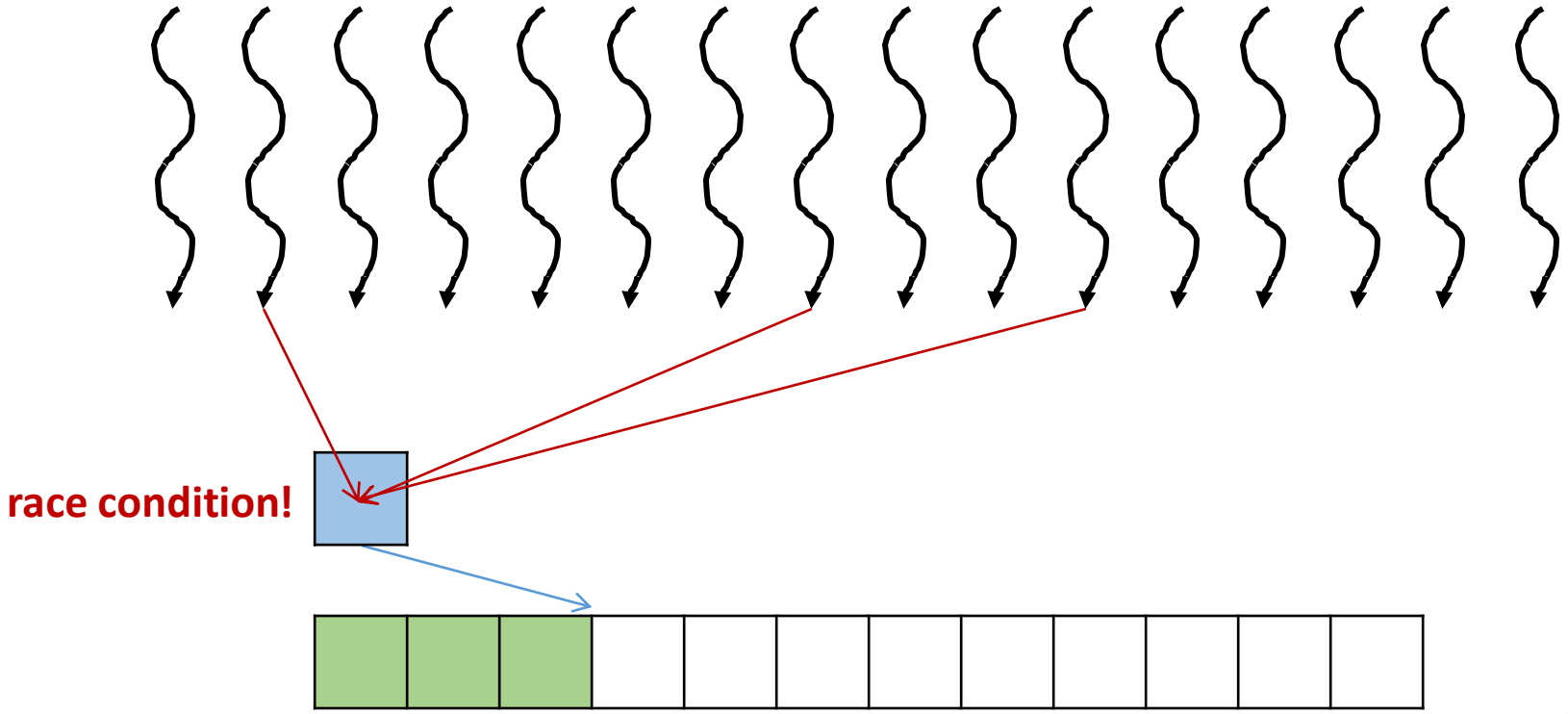
Vertex-Centric Frontier-Based BFS Code

```
__global__ void bfs_kernel(CSRGraph csrGraph, unsigned int* level, unsigned int* prevFrontier,
                           unsigned int* currFrontier, unsigned int numPrevFrontier,
                           unsigned int* numCurrFrontier, unsigned int currLevel) {
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    if(i < numPrevFrontier) {
        unsigned int vertex = prevFrontier[i];
        for(unsigned int edge = csrGraph.srcPtrs[vertex]; edge < csrGraph.srcPtrs[vertex + 1];
                                                                          ++edge) {
            unsigned int neighbor = csrGraph.dst[edge];
            if(atomicCAS(&level[neighbor], UINT_MAX, currLevel) == UINT_MAX) {
                unsigned int currFrontierIdx = atomicAdd(numCurrFrontier, 1);
                currFrontier[currFrontierIdx] = neighbor;
            }
        }
    }
}
```

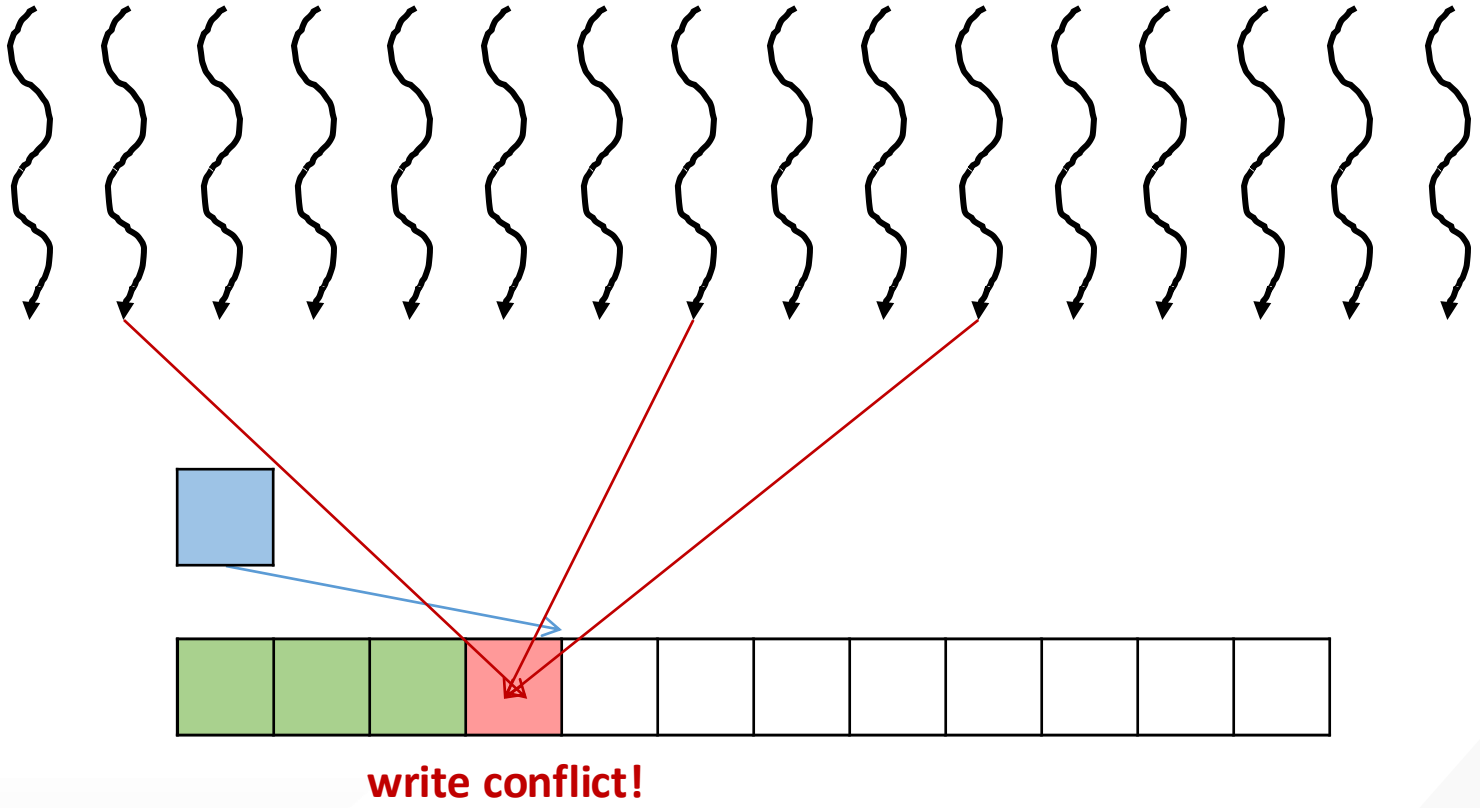
Ensure only one thread visits the neighbor to avoid adding it to the frontier multiple times

Avoid race conditions when adding to the frontier

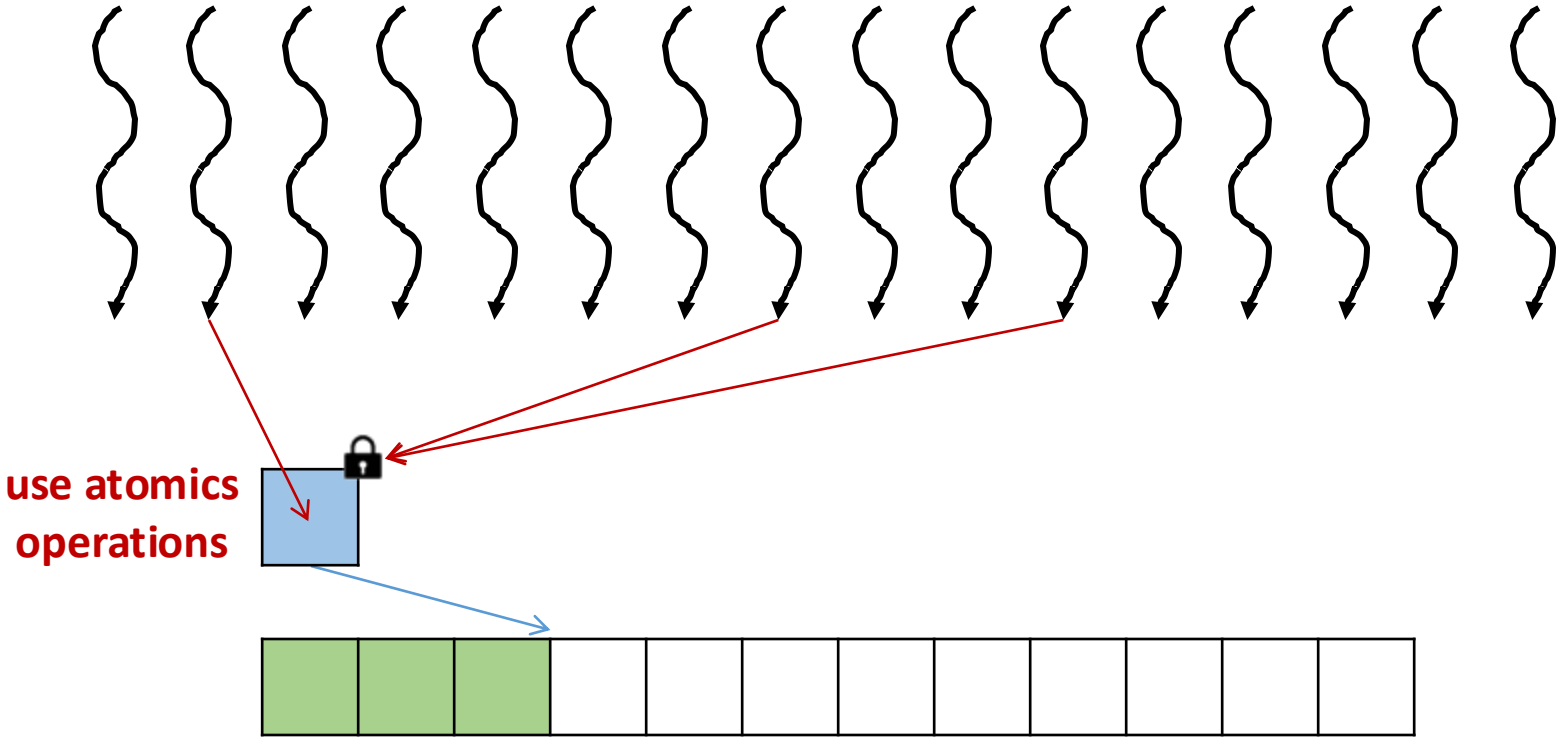
Adding Frontier Vertices



Adding Frontier Vertices



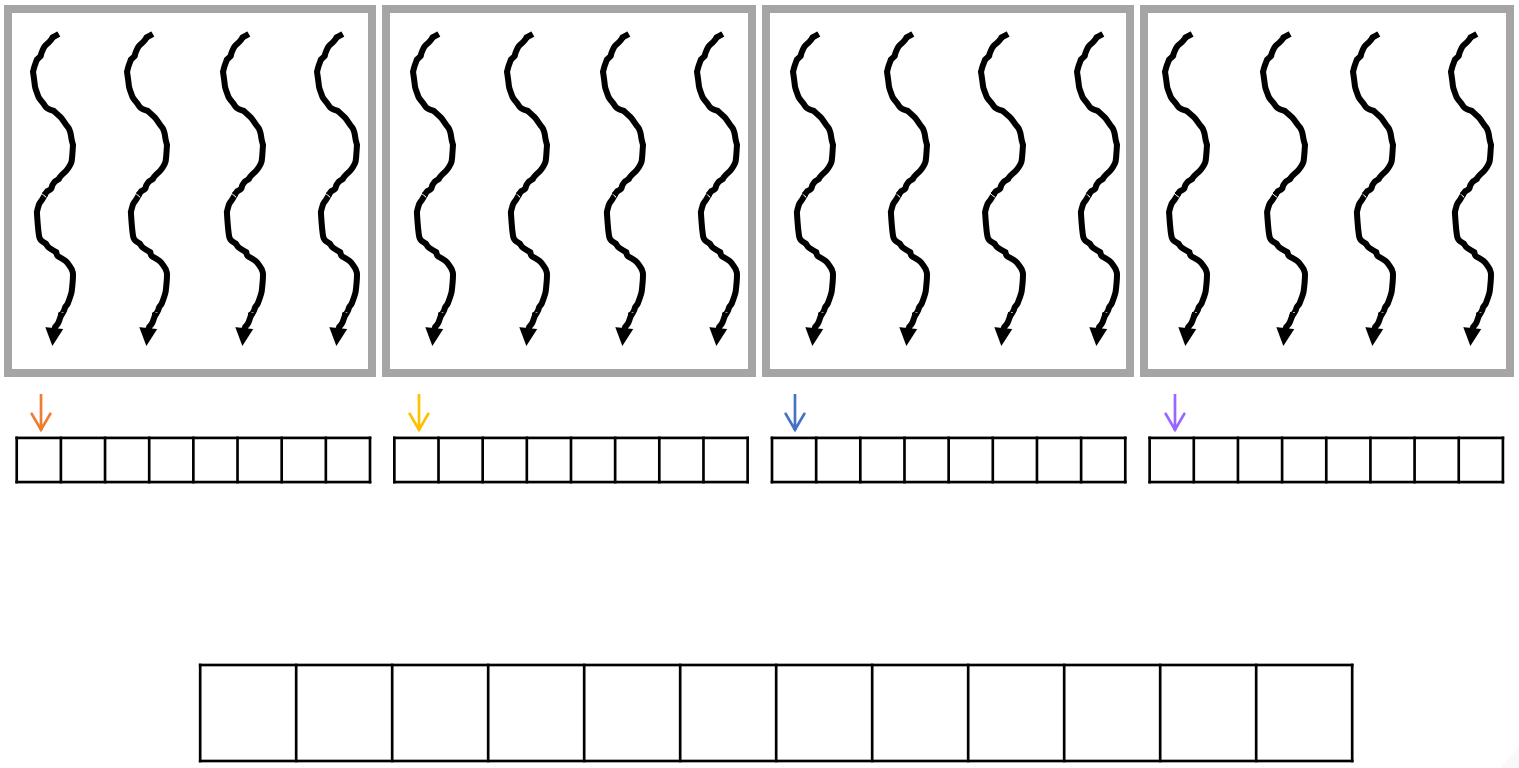
Adding Frontier Vertices



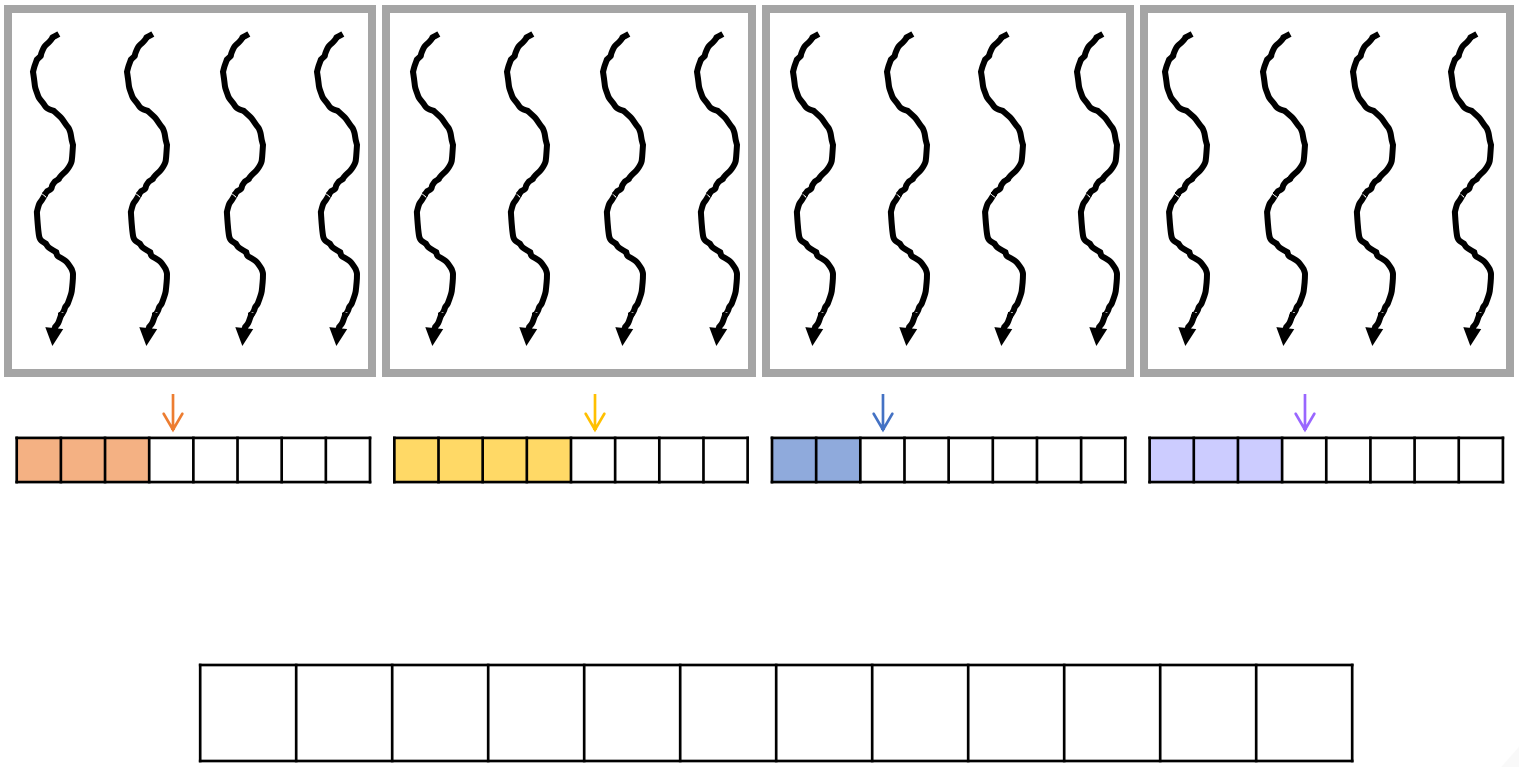
Frontier Privatization

- All threads atomically increment the same global counter to add elements to the frontier
  - High latency due to global memory access and serialization due to high contention
- Optimization: privatization and shared memory
  - Each thread block maintains a private frontier in shared memory and commits entries to the global frontier upon completion

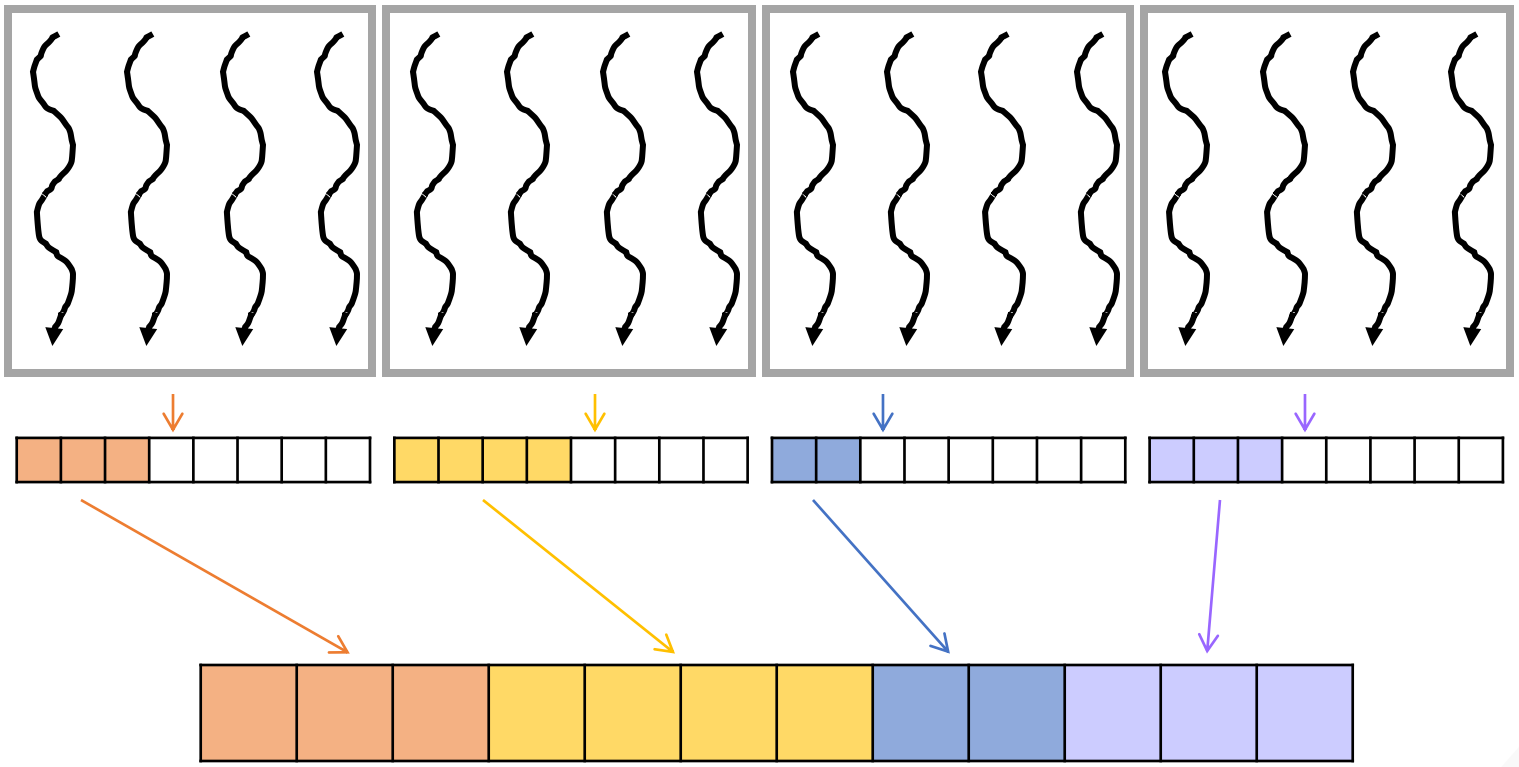
Frontier Privatization



Frontier Privatization



Frontier Privatization



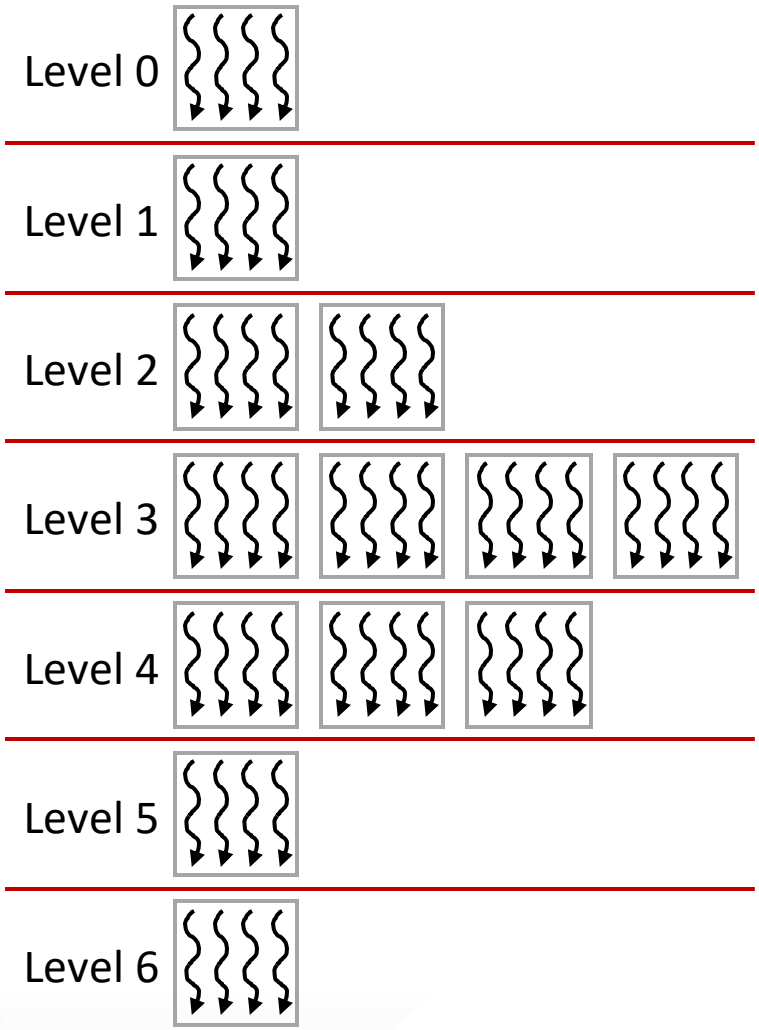
Vertex-Centric Frontier-Based BFS Code with Privatization

```
__global__ void bfs_kernel(CSRGraph csrGraph, unsigned int* level, unsigned int* prevFrontier,
      unsigned int* currFrontier, unsigned int numPrevFrontier, unsigned int* numCurrFrontier,  unsigned int currLevel) {

    // Initialize privatized frontier
    __shared__ unsigned int currFrontier_s[LOCAL_FRONTIER_CAPACITY];
    __shared__ unsigned int numCurrFrontier_s;
    if(threadIdx.x == 0) {
        numCurrFrontier_s = 0;
    }
    __syncthreads();

    // Perform BFS
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    if(i < numPrevFrontier) {
        unsigned int vertex = prevFrontier[i];
        for(unsigned int edge = csrGraph.srcPtrs[vertex]; edge < csrGraph.srcPtrs[vertex + 1]; ++edge) {
            unsigned int neighbor = csrGraph.dst[edge];
            if(atomicCAS(&level[neighbor], UINT_MAX, currLevel) == UINT_MAX) { // Vertex not previously visited
                unsigned int currFrontierIdx_s = atomicAdd(&numCurrFrontier_s, 1);
                if(currFrontierIdx_s < LOCAL_FRONTIER_CAPACITY) {
                    currFrontier_s[currFrontierIdx_s] = neighbor;
                } else {
                    numCurrFrontier_s = LOCAL_FRONTIER_CAPACITY;
                    unsigned int currFrontierIdx = atomicAdd(numCurrFrontier, 1);
                    currFrontier[currFrontierIdx] = neighbor;
                }
            }
        }
    }
    __syncthreads();


    // Allocate in global frontier
    __shared__ unsigned int currFrontierStartIdx;
    if(threadIdx.x == 0) {
        currFrontierStartIdx = atomicAdd(numCurrFrontier, numCurrFrontier_s);
    }
    __syncthreads();

    // Commit to global frontier
    for(unsigned int currFrontierIdx_s = threadIdx.x; currFrontierIdx_s < numCurrFrontier_s;
                                                                    currFrontierIdx_s += blockDim.x) {
        unsigned int currFrontierIdx = currFrontierStartIdx + currFrontierIdx_s;
        currFrontier[currFrontierIdx] = currFrontier_s[currFrontierIdx_s];
    }

}
```

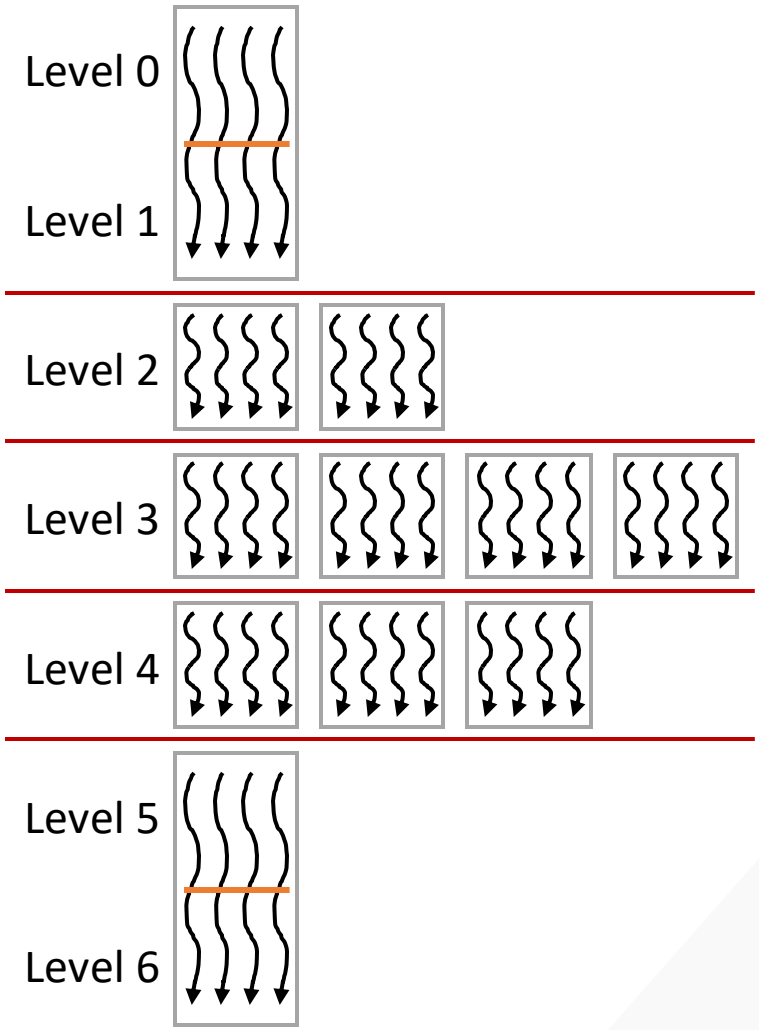
Minimizing Launch Overhead

- Need to synchronize between levels
  - Wait for all threads to add all vertices in the current level to the frontier before proceeding to the next level
- So far, we have launched a new grid for each level
  - Overhead of launching new grid and copying counter
- Optimization: If consecutive levels have few enough vertices that can be executed by one thread block:
  - Execute multiple levels in one single-block grid and synchronize between levels using `__syncthreads()`
  - Reduces total number of number of grid launches

Minimizing Launch Overhead



Launching a new grid for each level



Consecutive small levels in one grid

References

- Wen-mei W. Hwu, David B. Kirk, and Izzat El Hajj. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2022.