CSE 6220/CX 4220
Introduction to HPC

Lecture 22: Parallel Graph Optimization

Helen Xu
hxu615@gatech.edu

Slides from Prof. Julian Shun

Properties of Real-World Graphs

They can be big (but not too big)

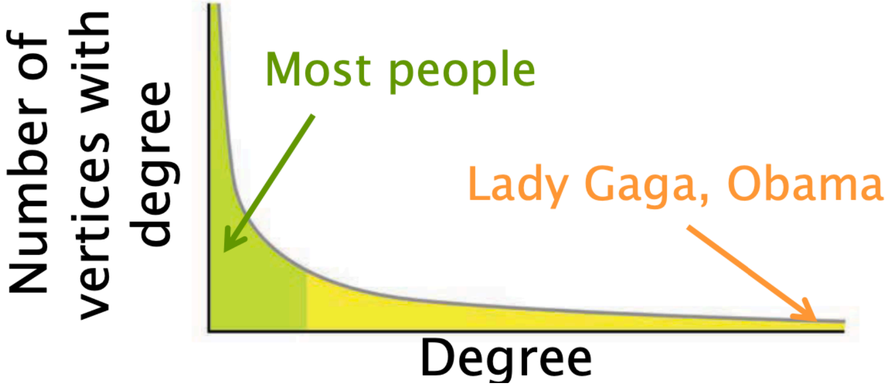
Social network
41 million vertices
1.5 billion edges
(6.3 GB)

Web graph
1.4 billion vertices
6.6 billion edges
(38 GB)

Web graph
3.5 billion vertices
128 billion edges
(540 GB)

Sparse (number of edges is much less than n^2)

Degrees can be highly **skewed**



Studies have shown that many real-world graphs have a power law degree distribution

#vertices with deg. $d \approx a \times d^{-p}$ $(2 < p < 3)$

From MIT 6.172

Graph Applications

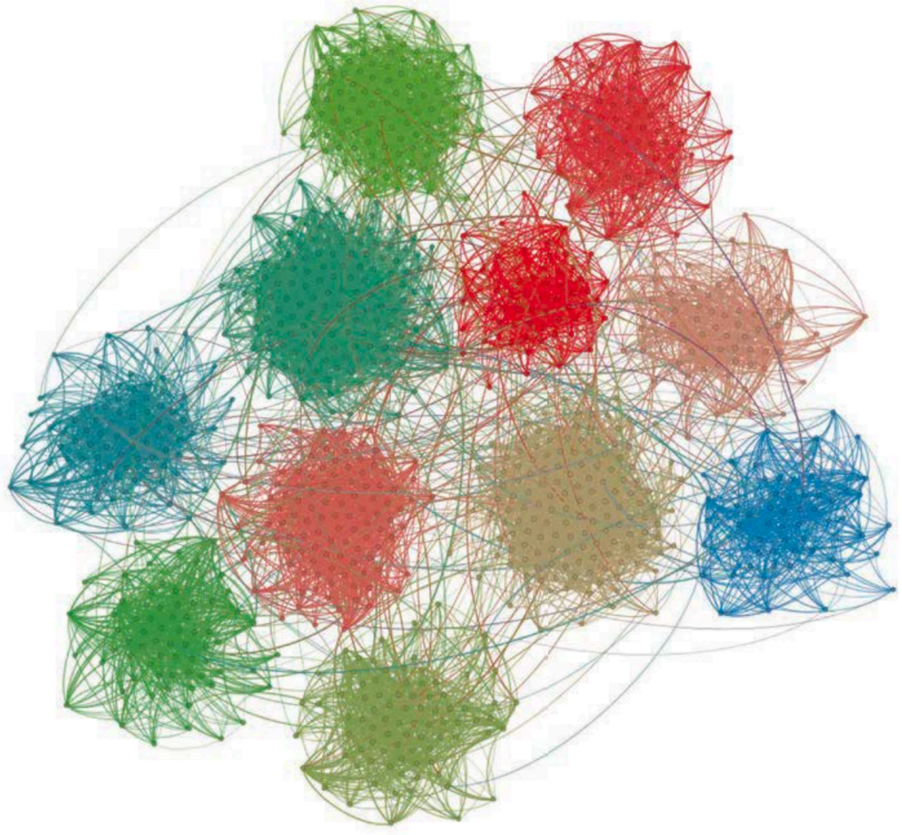
Social network queries

Examples:

- Finding all your friends who went to the same high school as you
- Finding common friends with someone
- Social networks recommending people whom you might know
- Product recommendation

Finding good clusters

Finding **groups of vertices** that are “well-connected” internally and “poorly-connected” externally



Some applications

- Finding people with similar interests
- Detecting fraudulent websites
- Document clustering
- Unsupervised learning

From MIT 6.172

Implementing a Graph Algorithm: Breadth-First Search

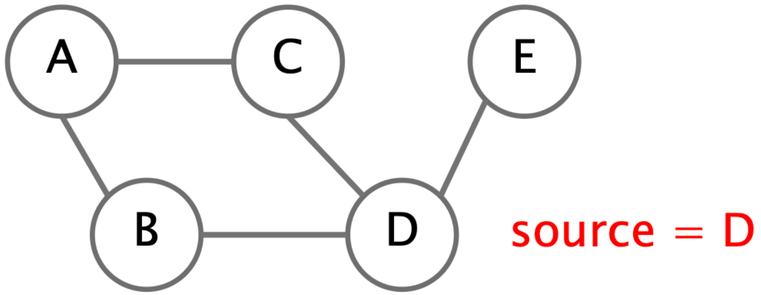
Breadth-First Search (BFS)

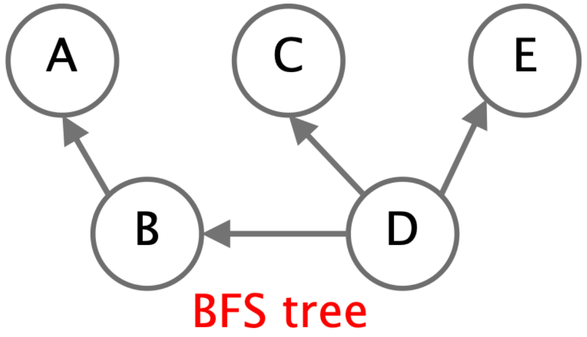
- Given a source vertex *s*, visit the vertices in order of distance from *s*
- Possible outputs:
  - Vertices in the order they were visited
    - D, B, C, E, A
  - The distance from each vertex to *s*

A	B	C	D	E
2	1	1	0	1

  - A BFS tree, where each vertex has a parent to a neighbor in the previous level

Applications
Betweenness centrality
Eccentricity estimation
Maximum flow
Web crawlers
Network broadcasting
Cycle detection
…





Serial BFS Algorithm Initialization

Suppose that we will compute the **parents array** (BFS tree)

Output

Nodes to visit next

Init queue with source

```
int* parent =
 (int*) malloc(sizeof(int)*n);
int* queue =
 (int*) malloc(sizeof(int)*n);

for(int i=0; i<n; i++) {
   parent[i] = -1;
}

queue[0] = source;
parent[source] = source;

int q_front = 0, q_back = 1;
```

https://en.wikipedia.org/wiki/Breadth-first_search

Serial BFS Algorithm

Assume the graph is in CSR: offsets and edges array
We have n vertices and m edges

```
//while queue not empty
while(q_front != q_back) {
    int current = queue[q_front++]; //dequeue
    int degree =
          Offsets[current+1]-Offsets[current];
    for(int i=0;i<degree; i++) {
          int ngh = Edges[Offsets[current]+i];
          //check if neighbor has been visited
          if(parent[ngh] == -1) {
              parent[ngh] = current;
              //enqueue neighbor
              queue[q_back++] = ngh;
          }
    }
}
```

Total of m random accesses

Remember: random access costs more than sequential access

What is the most expensive part of the code?

Analyzing the program

```
int* parent =
 (int*) malloc(sizeof(int)*n);
int* queue =
 (int*) malloc(sizeof(int)*n);

for(int i=0; i<n; i++) {
   parent[i] = -1;
}

queue[0] = source;
parent[source] = source;

int q_front = 0; q_back = 1;
```

```
//while queue not empty
while(q_front != q_back) {
   int current = queue[q_front++]; //dequeue
   int degree =
        Offsets[current+1]-Offsets[current];
   for(int i=0;i<degree; i++) {
        int ngh = Edges[Offsets[current]+i];
        //check if neighbor has been visited
        if(parent[ngh] == -1) {
            parent[ngh] = current;
            //enqueue neighbor
            queue[q_back++] = ngh;
        }
   }
}
```

How can we reduce cache misses?

(Approx.) analyze number of cache misses (cold cache; cache size << n; 64 byte cache line size; 4 byte int)

- n/16 for initialization
- n/16 for dequeueing
- n for accessing Offsets array
- ≤ 2n + m/16 for accessing Edges array
- m for accessing parent array
- n/16 for enqueueing

Total ≤ (51/16)n + (17/16)m

From MIT 6.172

Analyzing the program

```
int* parent =
 (int*) malloc(sizeof(int)*n);
int* queue =
 (int*) malloc(sizeof(int)*n);

for(int i=0; i<n; i++) {
   parent[i] = -1;
}

queue[0] = source;
parent[source] = source;

int q_front = 0; q_back = 1;
```

```
//while queue not empty
while(q_front != q_back) {
   int current = queue[q_front++]; //dequeue
   int degree =
        Offsets[current+1]-Offsets[current];
   for(int i=0;i<degree; i++) {
        int ngh = Edges[Offsets[current]+i];
        //check if neighbor has been visited
        if(parent[ngh] == -1) {
            parent[ngh] = current;
            //enqueue neighbor
            queue[q_back++] = ngh;
        }
   }
}
```

Check bitvector first before accessing parent array

n cache misses instead of m

- What if we can fit a bitvector of size n in cache?
  - Might reduce the number of cache misses
  - More computation to do bit manipulation

BFS with bitvector

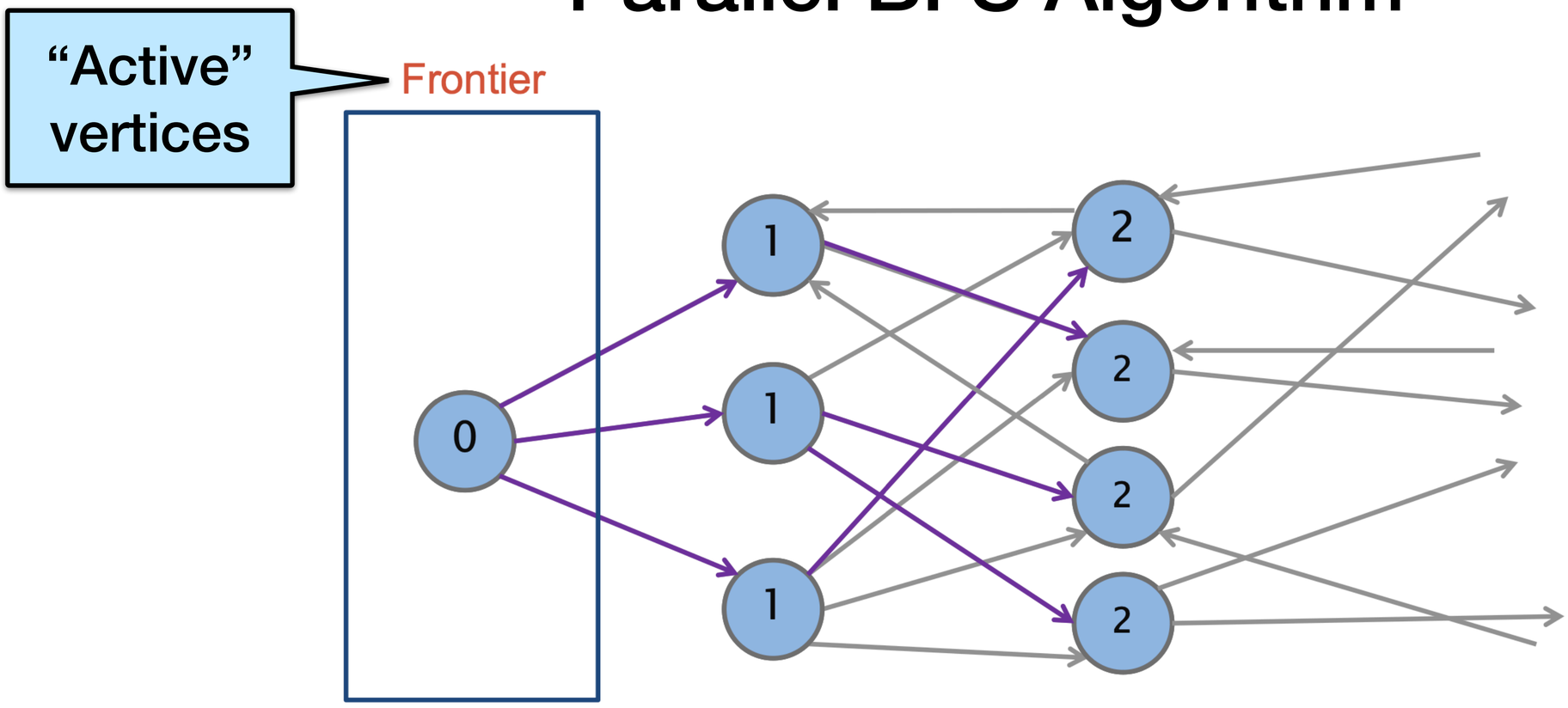
```
int* parent =
 (int*) malloc(sizeof(int)*n);
int* queue =
 (int*) malloc(sizeof(int)*n);
int nv = 1+n/32;
int* visited =
 (int*) malloc(sizeof(int)*nv);

for(int i=0; i<n; i++) {
   parent[i] = -1;
}

for(int i=0; i<nv; i++) {
   visited[i] = 0;
}

queue[0] = source;
parent[source] = source;
visited[source/32]
   = (1 << (source % 32));

int q_front = 0; q_back = 1;
```

```
//while queue not empty
while(q_front != q_back) {
   int current = queue[q_front++]; //dequeue
   int degree =
        Offsets[current+1]-Offsets[current];
   for(int i=0;i<degree; i++) {
      int ngh = Edges[Offsets[current]+i];
      //check if neighbor has been visited
      if(!((1 << ngh%32) & visited[ngh/32])){
         visited[ngh/32] |= (1 << (ngh%32));
         parent[ngh] = current;
         //enqueue neighbor
         queue[q_back++] = ngh;
      }
   }
}
```

- Bitvector version is faster for large enough values of m

Parallelizing Breadth-First Search

Parallel BFS Algorithm



- Can process each frontier in parallel
  - Parallelize over both the vertices and their outgoing edges
- Races, load balancing

Parallel BFS Code - Initialization

Instead of a queue, we have arrays for frontier, frontierNext, degrees

```
BFS(Offsets, Edges, source) {
  parent, frontier, frontierNext, and degrees are arrays
  parallel_for(int i=0; i<n; i++) parent[i] = -1;
  frontier[0] = source, frontierSize = 1, parent[source] = source;

  …
```

Parallel BFS: Overview

While the **frontier** is not empty:

In parallel, for all vertices v in the frontier:

Copy all neighbors of v into frontierNext (for the next iteration) - only if they have not yet been visited

Set v as the parent of all ngh(v) in the parents array - if ngh(v) does not yet have a parent in the parents array

Set frontierNext to frontier

Parallel BFS: Overview

While the frontier is not empty:

Problem: How do we know where to copy into?

In parallel, for all vertices v in the frontier:

Copy all neighbors of v into frontierNext (for the next iteration) - only if they have not yet been visited

Set v as the parent of all ngh(v) in the parents array - if ngh(v) does not yet have a parent in the parents array

Set frontierNext to frontier

Parallel BFS: Overview

While the frontier is not empty:

Problem: How do we know where to copy into?

In parallel, for all vertices v in the frontier:

Copy all neighbors of v into frontierNext (for the next iteration) - only if they have not yet been visited

Set v as the parent of all ngh(v) in the parents array - if ngh(v) does not yet have a parent in the parents array

Set frontierNext to frontier

Problem: What if multiple vertices in the frontier have the same neighbor?

Parallel BFS Code - Degree Setup

Problem: How do we know **where to copy the neighbors** for each vertex in the frontier to?

Answer: **Prefix sum** on the degrees

```
…

while(frontierSize > 0) {
  parallel_for(int i=0; i<frontierSize; i++)
    degrees[i] = Offsets[frontier[i]+1] – Offsets[frontier[i]];

  perform prefix sum on degrees array

  …
}
```

For all vertices in frontier, get their degrees

Exclusive scan to get starting point for each vertex

Example: Degrees:

Parallel BFS Code

```
…

while(frontierSize > 0) {
  // SETUP DEGREES AS ON PREVIOUS SLIDE

  parallel_for(int i=0; i<frontierSize; i++) {
    v = frontier[i], index = degrees[i], d = Offsets[v+1]-Offsets[v];
    for(int j=0; j<d; j++) { //can be parallel
      ngh = Edges[Offsets[v]+j];
      if(parent[ngh] == -1 && compare-and-swap(&parent[ngh], -1, v)) {
        frontierNext[index+j] = ngh;
      } else { frontierNext[index+j] = -1; }
    }
  }
  filter out “-1” from frontierNext, store in frontier, and update
  frontierSize to be the size of frontier (all done using prefix sum)
}
```

Iterate over vertices in frontier

Parallel BFS Code

```
…

while(frontierSize > 0) {
  // SETUP DEGREES AS ON PREVIOUS SLIDE

  parallel_for(int i=0; i<frontierSize; i++) {
    v = frontier[i], index = degrees[i], d = Offsets[v+1]-Offsets[v];
    for(int j=0; j<d; j++) { //can be parallel
      ngh = Edges[Offsets[v]+j];
      if(parent[ngh] == -1 && compare-and-swap(&parent[ngh], -1, v)) {
        frontierNext[index+j] = ngh;
      } else { frontierNext[index+j] = -1; }
    }
  }
  filter out “-1” from frontierNext, store in frontier, and update
  frontierSize to be the size of frontier (all done using prefix sum)
}
```

Iterate over vertices in frontier

Copy in using starting points computed previously

Parallel BFS Code

```
…

while(frontierSize > 0) {
  // SETUP DEGREES AS ON PREVIOUS SLIDE

  parallel_for(int i=0; i<frontierSize; i++) {
    v = frontier[i], index = degrees[i], d = Offsets[v+1]-Offsets[v];
    for(int j=0; j<d; j++) { //can be parallel
      ngh = Edges[Offsets[v]+j];
      if(parent[ngh] == -1 && compare-and-swap(&parent[ngh], -1, v)) {
        frontierNext[index+j] = ngh;
      } else { frontierNext[index+j] = -1; }
    }
  }
  filter out “-1” from frontierNext, store in frontier, and update
  frontierSize to be the size of frontier (all done using prefix sum)
}
```

Iterate over vertices in frontier

Copy in using starting points computed previously

If this neighbor hasn’t been explored yet

Parallel BFS Code

```
…

while(frontierSize > 0) {
  // SETUP DEGREES AS ON PREVIOUS SLIDE

  parallel_for(int i=0; i<frontierSize; i++) {
    v = frontier[i], index = degrees[i], d = Offsets[v+1]-Offsets[v];
    for(int j=0; j<d; j++) { //can be parallel
      ngh = Edges[Offsets[v]+j];
      if(parent[ngh] == -1 && compare-and-swap(&parent[ngh], -1, v)) {
        frontierNext[index+j] = ngh;
      } else { frontierNext[index+j] = -1; }
    }
  }
  filter out “-1” from frontierNext, store in frontier, and update
  frontierSize to be the size of frontier (all done using prefix sum)
}
```

Iterate over vertices in frontier

Copy in using starting points computed previously

If this neighbor hasn’t been explored yet

Other vertices in the frontier may also have ngh as their neighbor. Only one should add it.

Parallel BFS Code

```
…

while(frontierSize > 0) {
  // SETUP DEGREES AS ON PREVIOUS SLIDE

  parallel_for(int i=0; i<frontierSize; i++) {
    v = frontier[i], index = degrees[i], d = Offsets[v+1]-Offsets[v];
    for(int j=0; j<d; j++) { //can be parallel
      ngh = Edges[Offsets[v]+j];
      if(parent[ngh] == -1 && compare-and-swap(&parent[ngh], -1, v)) {
        frontierNext[index+j] = ngh;
      } else { frontierNext[index+j] = -1;
    }
  }
  filter out “-1” from frontierNext, store in frontier, and update
  frontierSize to be the size of frontier (all done using prefix sum)
}
```

Iterate over vertices in frontier

Copy in using starting points computed previously

If this neighbor hasn’t been explored yet

Otherwise, do not add to frontierNext

Other vertices in the frontier may also have ngh as their neighbor. Only one should add it.

Parallel BFS Code

```
…

while(frontierSize > 0) {
  // SETUP DEGREES AS ON PREVIOUS SLIDE

  parallel_for(int i=0; i<frontierSize; i++) {
    v = frontier[i], index = degrees[i], d = Offsets[v+1]-Offsets[v];
    for(int j=0; j<d; j++) { //can be parallel
      ngh = Edges[Offsets[v]+j];
      if(parent[ngh] == -1 && compare-and-swap(&parent[ngh], -1, v)) {
        frontierNext[index+j] = ngh;
      } else { frontierNext[index+j] = -1;
    }
  }
  filter out "-1" from frontierNext, store in frontier, and update
  frontierSize to be the size of frontier (all done using prefix sum)
}
```

Iterate over vertices in frontier

Copy in using starting points computed previously

If this neighbor hasn't been explored yet

Otherwise, do not add to frontierNext

Other vertices in the frontier may also have ngh as their neighbor. Only one should add it.

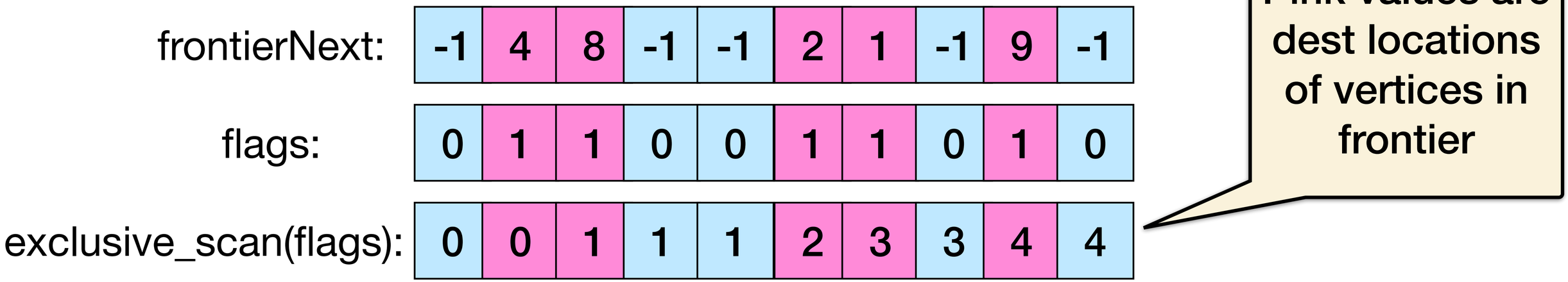
Question: How would you do this?

Filter: Filling in next frontier with prefix sum

Problem: We have frontierNext, which has some -1 (empty) and some valid vertices (>=0). How do we pack them to the front of frontierNext?

Answer: Parallel filter with prefix sum

Example:




```
parallel_for i from 0 to len(frontierNext):
  if flags[i] == 1:
    frontier[result_of_flag_scan[i]] = frontierNext[i]
```

Compare and swap

Compare-and-swap (CAS) is an **atomic** instruction that compares the contents of a memory location with a given (old) value and, only if they are the same, modifies the contents of the location to a new given value.

CAS is used to implemented mutexes, as well as lock-free and wait-free algorithms.

```
function cas(p: pointer to int, old: int, new: int)
    if *p ≠ old
        return false

    *p ← new

    return true
```

BFS Span Analysis

Longest path in graph

Number of iterations <= **diameter** D of graph

Each iteration takes $\Theta(\log(m))$ span for parallel for loops, prefix sum, and filter (assuming inner loop is parallelized)

Span = $\Theta(D \log(m))$

BFS Work Analysis

Sum of frontier sizes = n

Each edge traversed once -> m total visits

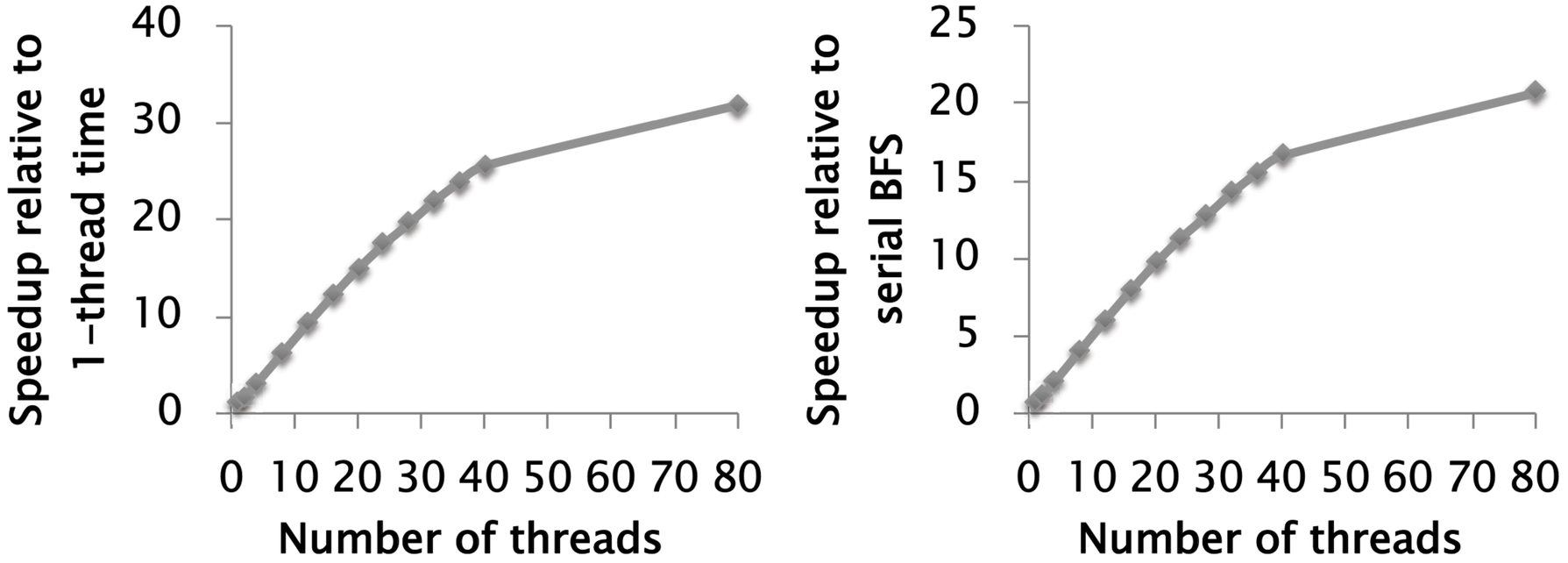
Work of prefix sum on each iteration is proportional to frontier size -> $\Theta(n)$ total

Work of filter on each iteration is proportional to number of edges traversed -> $\Theta(m)$ total

Work = $\Theta(m + n)$

Performance of Parallel BFS

- Random graph with $n=10^7$ and $m=10^8$
  - 10 edges per vertex
- 40-core machine with 2-way hyperthreading



- 31.8x speedup on 40 cores with hyperthreading
- Serial BFS is 54% faster than parallel BFS on 1 thread

From MIT 6.172

Dealing with nondeterminism

```
…

while(frontierSize > 0) {
  // SETUP DEGREES AS ON PREVIOUS SLIDE

  parallel_for(int i=0; i<frontierSize; i++) {
    v = frontier[i], index = degrees[i], d = Offsets[v+1]-Offsets[v];
    for(int j=0; j<d; j++) { //can be parallel
      ngh = Edges[Offsets[v]+j];
      if(parent[ngh] == -1 && compare-and-swap(&parent[ngh], -1, v)) {
        frontierNext[index+j] = ngh;
      } else { frontierNext[index+j] = -1; }
    }
  }
  …
}
```

Nondeterministic

Nondeterministic parallel programs are hard to debug. Can we substitute a **deterministic alternative**?

```
writeMin(addr, newval):
  oldval = *addr
  while(newval < oldval):
    if(CAS(addr, oldval, newval)) return
    else: oldval = addr*
```

Deterministic Parallel BFS

```
parallel(int i=0; i<frontierSize; i++) { //phase 1
  v = frontier[i], index = degrees[i], d = Offsets[v+1]-Offsets[v];
  for(int j=0; j<d; j++) { //can be parallel
    ngh = Edges[Offsets[v]+j];
    writeMin(&parent[ngh], v); }
}
parallel_for(int i=0; i<frontierSize; i++) { //phase 2
  v = frontier[i], index = degrees[i], d = Offsets[v+1]-Offsets[v];
  for(int j=0; j<d; j++) { //can be parallel
    ngh = Edges[Offsets[v]+j];
    if(parent[ngh] == v) {
      parent[ngh] = -v; //to avoid revisiting
      frontierNext[index+j] = ngh; }
    else { frontierNext[index+j] = -1; }}
  }
  filter out "-1" from frontierNext, store in frontier, and update frontierSize
}}
```

Smallest value gets written

Check if v "won"

Deterministic Parallel BFS

```
writeMin(addr, newval):
  oldval = *addr
  while(newval < oldval):
    if(CAS(addr, oldval, newval)) return
    else: oldval = addr*
```

```
parallel(int i=0; i<frontierSize; i++) { //phase 1
  v = frontier[i], index = degrees[i], d = Offsets[v+1]-Offsets[v];
  for(int j=0; j<d; j++) { //can be parallel
    ngh = Edges[Offsets[v]+j];
    writeMin(&parent[ngh], v); }
}
parallel_for(int i=0; i<frontierSize; i++) { //phase 2
  v = frontier[i], index = degrees[i], d = Offsets[v+1]-Offsets[v];
  for(int j=0; j<d; j++) { //can be parallel
    ngh = Edges[Offsets[v]+j];
    if(parent[ngh] == v) {
      parent[ngh] = -v; //to avoid revisiting
      frontierNext[index+j] = ngh; }
    else { frontierNext[index+j] = -1; }}
  }
  filter out "-1" from frontierNext, store in
}}
```

Smallest value gets written

Check if v "won"

On 32 cores, (an optimized version of) deterministic BFS is 5-20% slower than nondeterministic BFS