Parallel Graph Connectivity

Parallel algorithms: lecture 3

15-853, Spring 2018

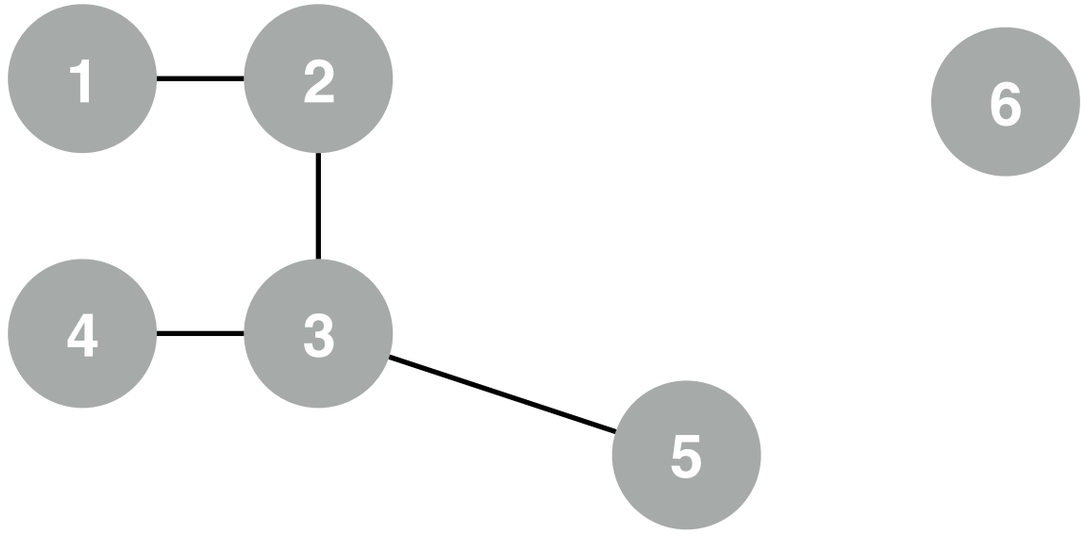
Outline

- Connectivity
- Parallel BFS
- Random-mate connectivity
- Low-diameter decomposition
- Work-efficient connectivity
- Are parallel graph algorithms practical?

Graph Connectivity

- G(V, E), n = #vertices, m = #edges
- Given an undirected graph G(V, E):

are $s, t, \in V$ connected?



- Sequential algorithm: run BFS or DFS. $O(n + m)$ time
- Nearly linear-work with union-find

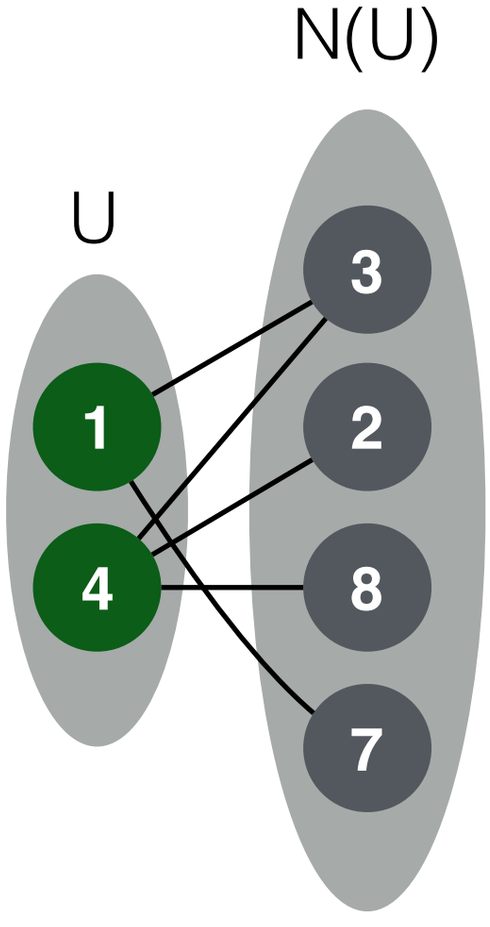
Parallel BFS

- BFS(G(V, E), v):
  - Compute a BFS tree rooted at v
  - I.e. compute a parent for all vertices reachable from v
- Idea: emulate sequential BFS. Run each step in parallel

- How do we compute the next frontier from current frontier?
- edge_map: primitive for traversal

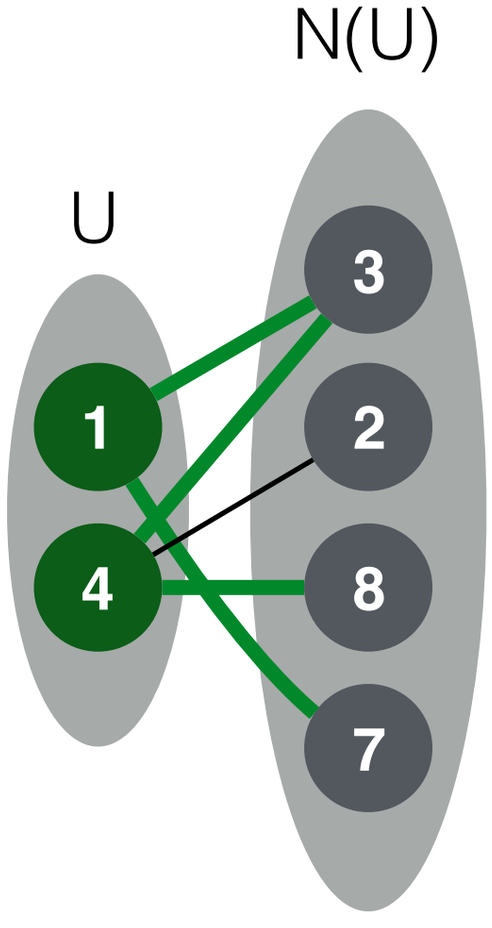
edge_map

- Input:
  - G(V, E)
  - U (subset of vertices)
  - update: vtx x vtx -> bool
- Output: $[v|(u,v) \in E, u \in U, \texttt{update}(u,v) = \texttt{true}]$



edge_map

- Input:
  - G(V, E)
  - U (subset of vertices)
  - update: vtx x vtx -> bool
- Output: $[v|(u,v) \in E, u \in U, \texttt{update}(u,v) = \texttt{true}]$

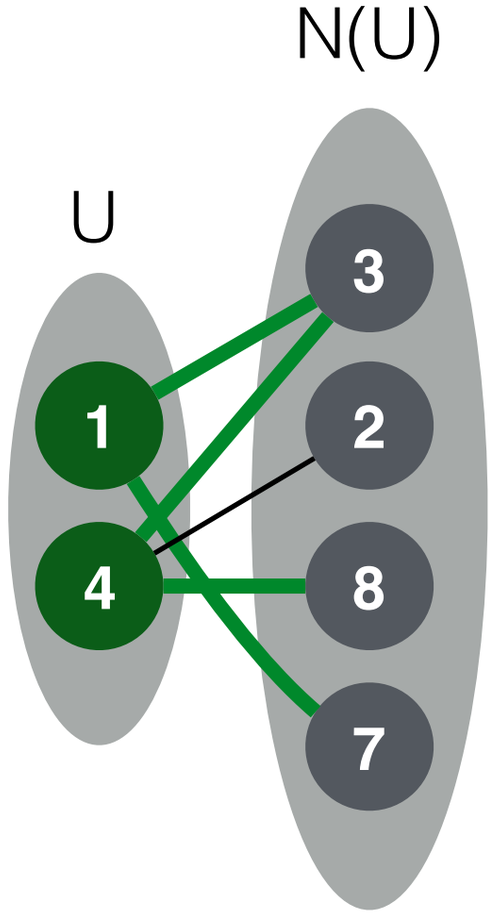


Output: [3, 7, 3, 8]

Usually implement update s.t. output is a set

edge_map

```
edge_map(G, U, update) =
  nghs = array(|U|, <>);
  parfor i in [0, |U|]
    v = U[i];
    out_nghs = G[v].out_nghs;
    update_vtx = lambda x.update(v, x);
    nghs[i] = filter(out_nghs, update_vtx);
return flatten(nghs);
```



- Runs in

$$O(|U| + \sum_{u \in U} deg_+(u)) \text{ work}$$

$$O(\log n) \text{ depth}$$

Parallel BFS

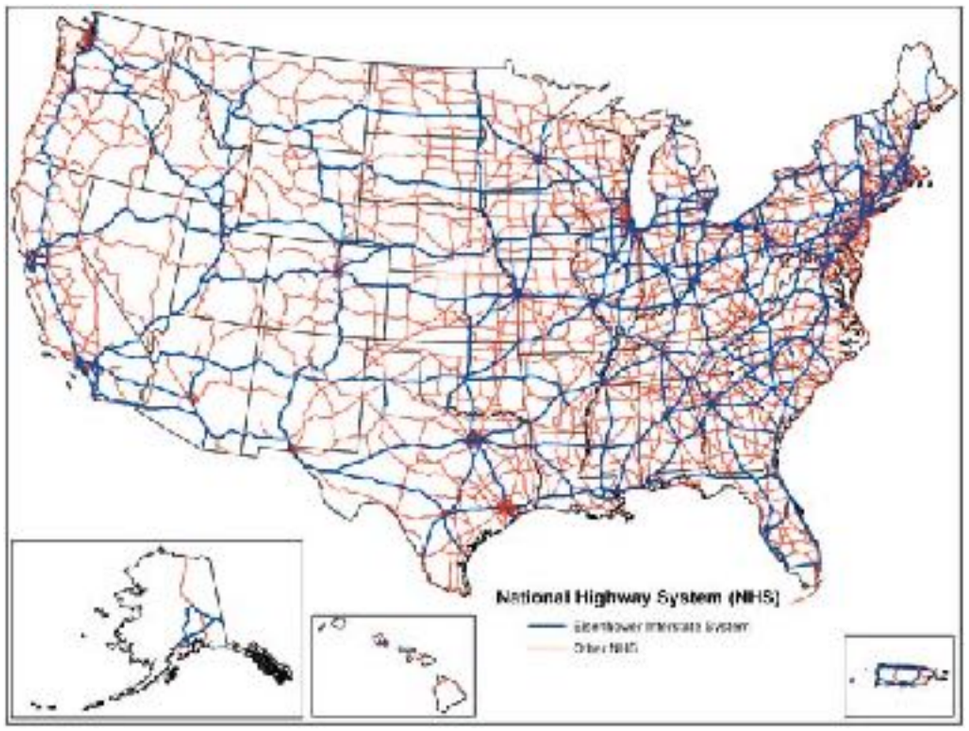
```
BFS(G(V, E), v) =
  n = |V|;
  frontier = array(v);
  visited = array(n, 0); visited[v] = 1;
  parents = array(n, -1);
  update = lambda (u, v).
    if (!visited[v] && test_and_set(&visited[v]))
      parents[v] = u;
      return true;
    return false;
while (|frontier| > 0):
  frontier = edge_map(G, frontier, update);
return parents;
```

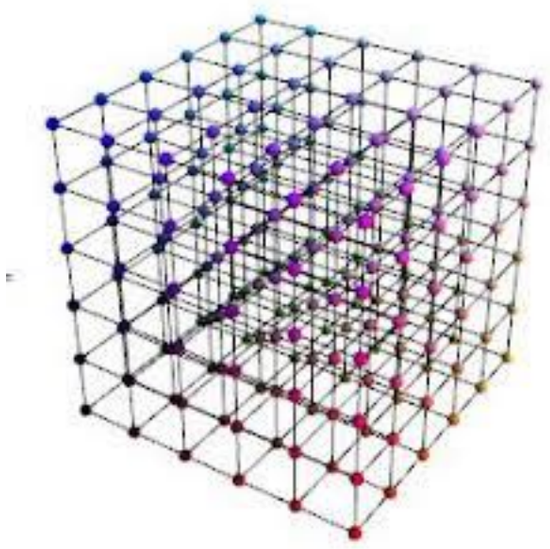
run at most diam(G) times

$O(m)$ work $\qquad$ $O(\mathsf{diam}(G) \log n)$ depth

Parallel BFS for connectivity

- Real world graphs can have high diameter
  - e.g. road networks and meshes



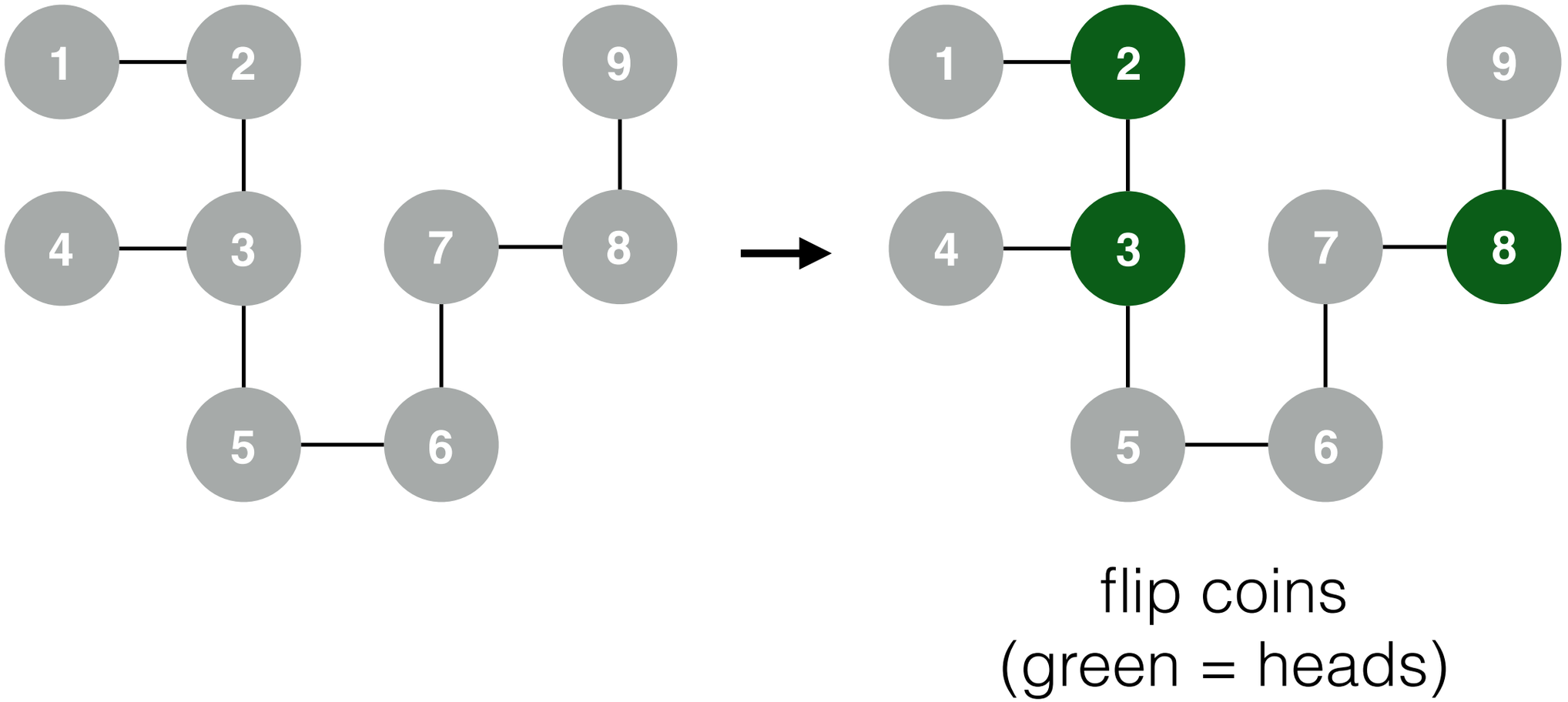


- Sequential dependencies between components
- BFS-based approach can be very fast on social/web graphs*
  - Process the massive component using BFS
  - Process remaining (small) components using LP

* Multistep connectivity, Slota et al. (2014)

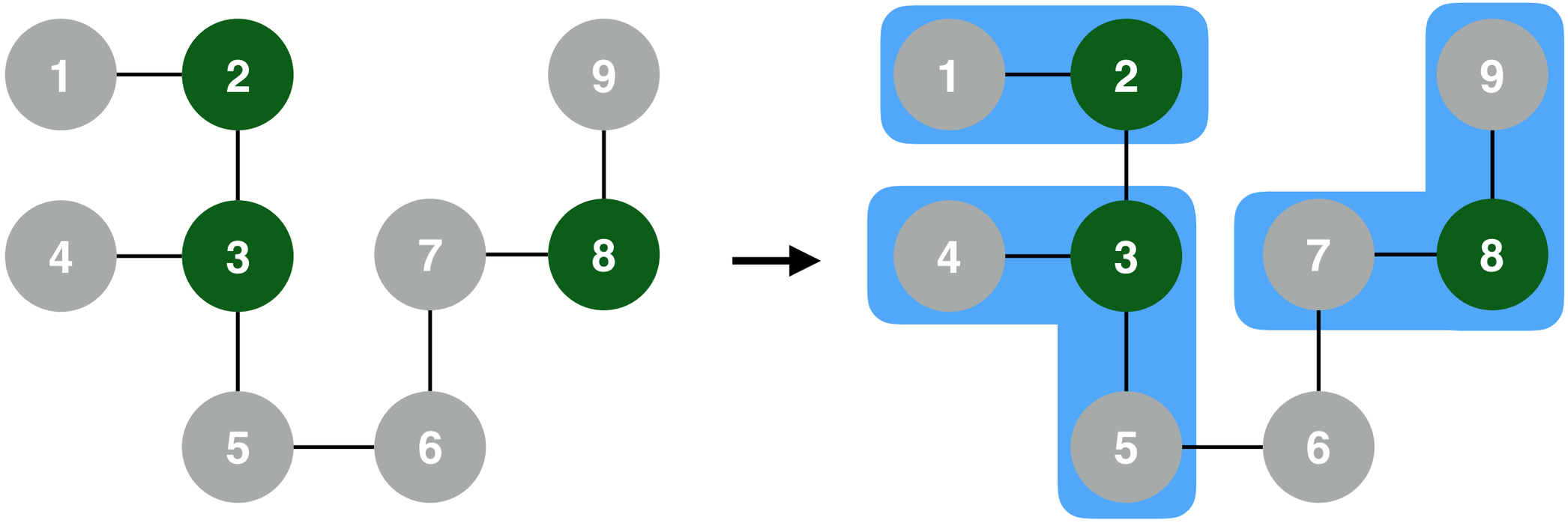
Random Mate

- Idea: form a set of disjoint stars and contract
- #vertices decrease by a constant fraction each round
  - Can show this implies O(log n) rounds w.h.p.



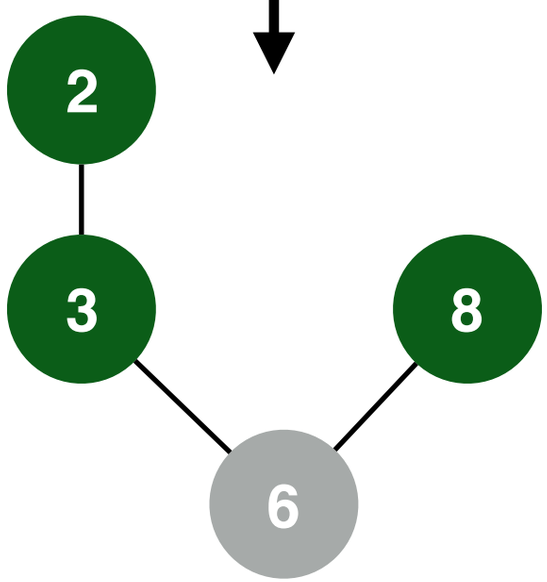
Source: Blelloch and Maggs, 6886-s18, lecture5.2

Random Mate



flip coins
(green = heads)

form stars



contract

Source: Blelloch and Maggs, 6886-s18, lecture5.2

Random Mate Implementation

- Use edgelist format E = [(1, 3), (1, 4), (3, 1), (4, 1), …]

2m edges

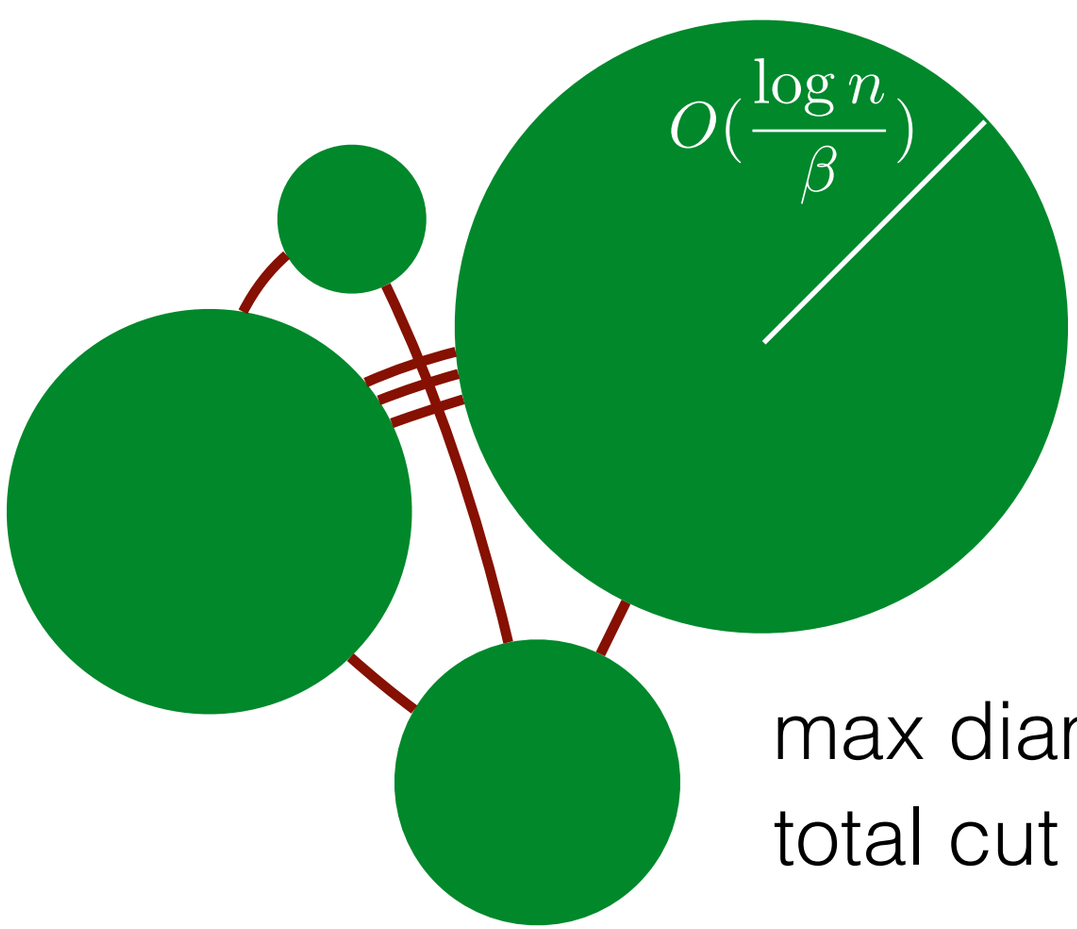
```
CC_Random_Mate(L, E) =
  if (|E| = 0) return L;
  1. flip coins for all n vertices
  2. For v where flip(v)=tails, hook to an
     arbitrary heads ngh w, set L(v) = w
  3. E' = filter(E, lambda (u,v).
              return L(u) != L(v);); // remove self edges
  4. L' = CC_Random_Mate(L, E');
  5. For v where flip(v)=tails, set L'(v)=L'(w)
     (v hooked to w in step 2)
  return L';
```

- Each iteration: O(m) work and O(log n) depth
- Each iteration reduces #active vertices by 1/4 in expectation
  - O(log n) rounds w.h.p.
  - O(m log n) work, O(log^2 n) depth in total (both w.h.p.)

Source: Blelloch and Maggs, 6886-s18, lecture5.2

Low-diameter decomposition

- Goal: decompose V into a set of clusters s.t.
  - the number of inter-cluster edges is “small”
  - diameter of each cluster is “small” (~log(n))
- More formally, given a parameter β, $0 < \beta < 1$



max diameter is $O(\log n/\beta)$
total cut edges $\leq \beta m$

Low-diameter decomposition

- Even more formally, a $(\beta, d)-$decomposition, $0 < \beta < 1$ is a partition of V into $V_1, \ldots, V_k$ s.t.
  - The shortest path between $u, v \in V_i$ using only vertices in V_i is at most d (strong diameter)
  - The number of edges $(u, v) \in E, u \in V_i, v \in V_j, i \neq j$ is at most βm (few inter-component edges)

Sequential low-diameter decomposition

An interesting sequential algorithm:*

- Pick an arbitrary vertex
- Grow a ball around it using BFS. Stop at the first radius r s.t. the #boundary edges is < than beta * #internal edges
- Can show that the radius is at most $O(\log n/\beta)$
- LDD = run substep until all vertices are covered

Can you prove this gives a $(\beta, O(\log n/\beta))$-decomposition?

Finding each ball is parallelizable, but there are sequential dependencies between different balls…

*see: 15-745 Lecture 12 Notes for details

Parallel low-diameter decomposition

- Miller, Peng and Xu give an algorithm that computes an $(\beta, O(\log n/\beta))$-decomposition in
  - $O(m+n)$ expected work
  - $O(\log^2 n)$ depth w.h.p.

Idea: grow balls in parallel from different vertices

Mimic sequential ball-growing process to ensure strong diameter.

Challenge: how to guarantee that not too many edges are cut and that the maximum radius is O(log n / beta)?

Use properties of the exponential distribution to ensure bounds on #cut edges and radius.

Miller, Peng, Xu (2013): https://arxiv.org/abs/1307.3692

Parallel low-diameter decomposition

```
LDD(G(V, E), beta) =
  n = |V|; num_finished = 0;
  E = array(n, lambda i.Exp(beta));
  C = array(n, -1);
parfor i in [0:n]
  C[i] = v in V minimizing (d(v, i) - E[v]);
return C;
```

shifted distance

Parallel low-diameter decomposition

Equivalently: compute start times based on E, run multi-BFS

```
LDD(G(V, E), beta) =
  n = |V|; num_finished = 0;
  E = array(n, lambda i.Exp(beta));
  S = array(n, lambda i.max(E) - E[i]);
  C = array(n, (infty, infty));
  num_processed = 0; round = 1;
  while (num_processed < n)
    F = F ∪ {v in V | S[v] < round, C[v] == infty};
    num_processed += |F|;
    update = lambda (u,v).
      if (C[v].snd == infty)
        writeMin(&C[v].fst, S[u]);
      return false;
    edge_map(G, F, update);
    check = lambda (u,v).
      if (C[v].fst == S[u])
        C[v].snd = u;
        return true;
      return false;
    F = edge_map(G, F, check);
    round++;
return C;
```

1. Compute start times

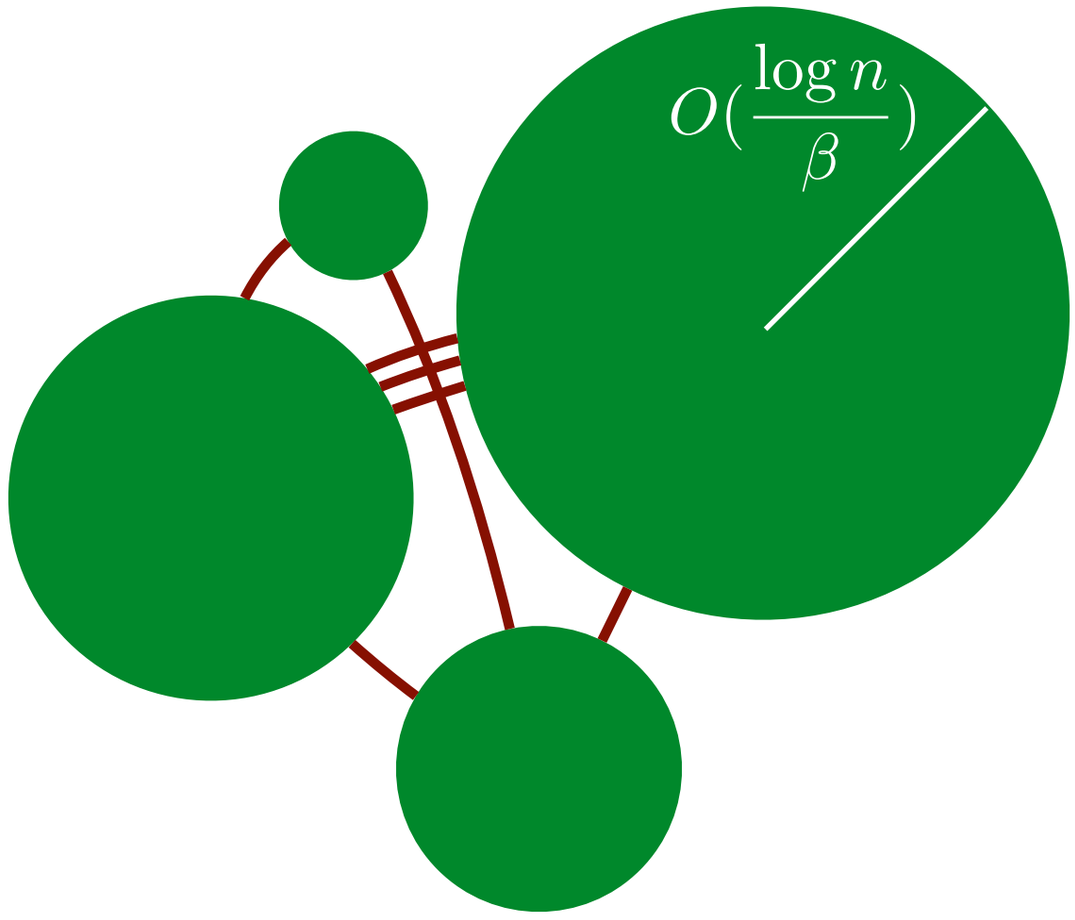
L1: Add ready centers

L2: Acquire unvisited nghs

L3: Set ngh's cluster id if we won

Parallel low-diameter decomposition

- Strong diameter is $O(\log n/\beta)$



Note: all vertices will start after max[E] rounds

- What is the maximum of n R.V.'s independently drawn from $Exp(\beta)$?

Parallel low-diameter decomposition

- What is the maximum of n R.V.'s independently drawn from $Exp(\beta)$?
- Let $\delta_v \sim Exp(\beta)$

$$\mathbf{Pr}\left[\delta_{\max} > \frac{k \log n}{\beta}\right] \leq \sum_{v \in V} \mathbf{Pr}\left[\delta_v > \frac{k \log n}{\beta}\right] \quad \text{(union bound)}$$

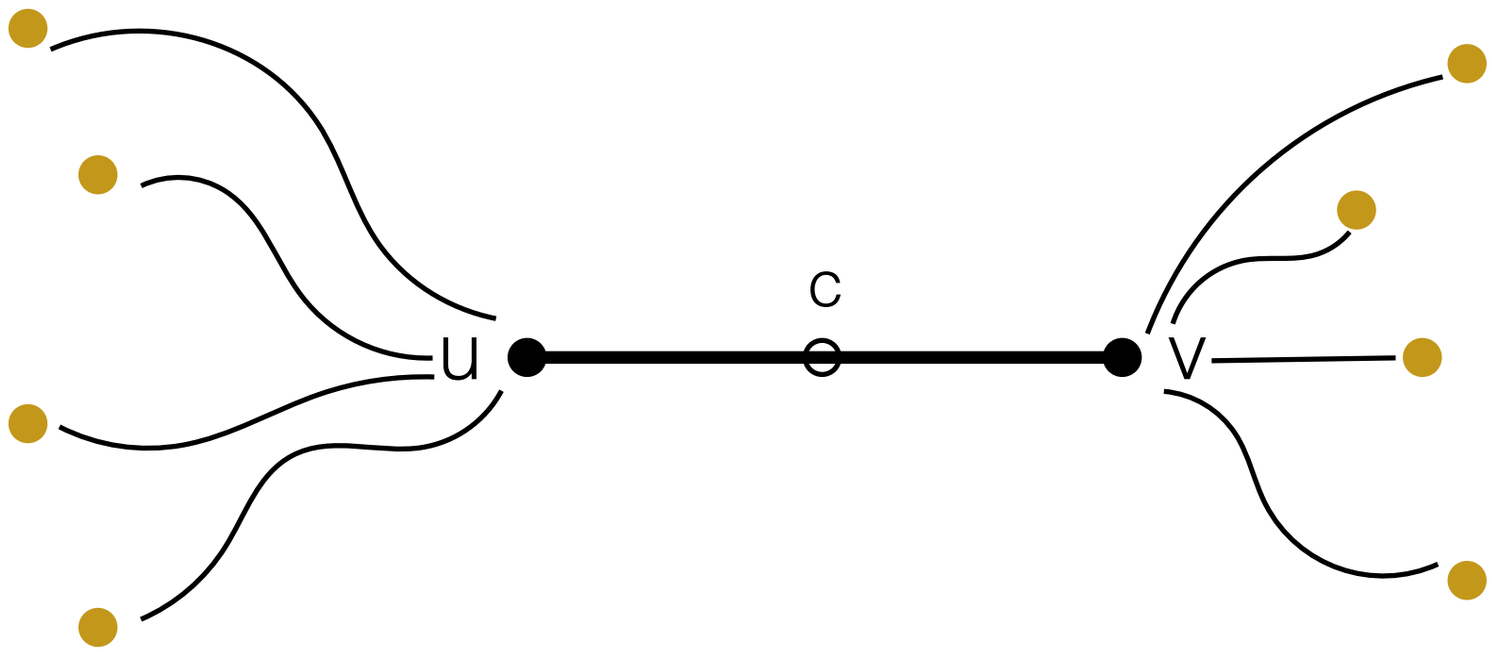
$$= n \cdot \exp\left(-\beta \cdot \frac{k \log n}{\beta}\right) \quad \text{(cdf of } Exp(\beta)\text{)}$$

$$= \frac{1}{n^{k-1}}$$

- Maximum diameter is $O(\log n/\beta)$ w.h.p.

Parallel low-diameter decomposition

- Claim: each edge is cut (intercluster) with probability $< \beta$



- Arrival times are random variables:

$$T_i = \delta_{\max} - \delta_i + d(i, c)$$

define $\hat{T}_i = \delta_{\max} - T_i = \delta_i - d(i, c)$

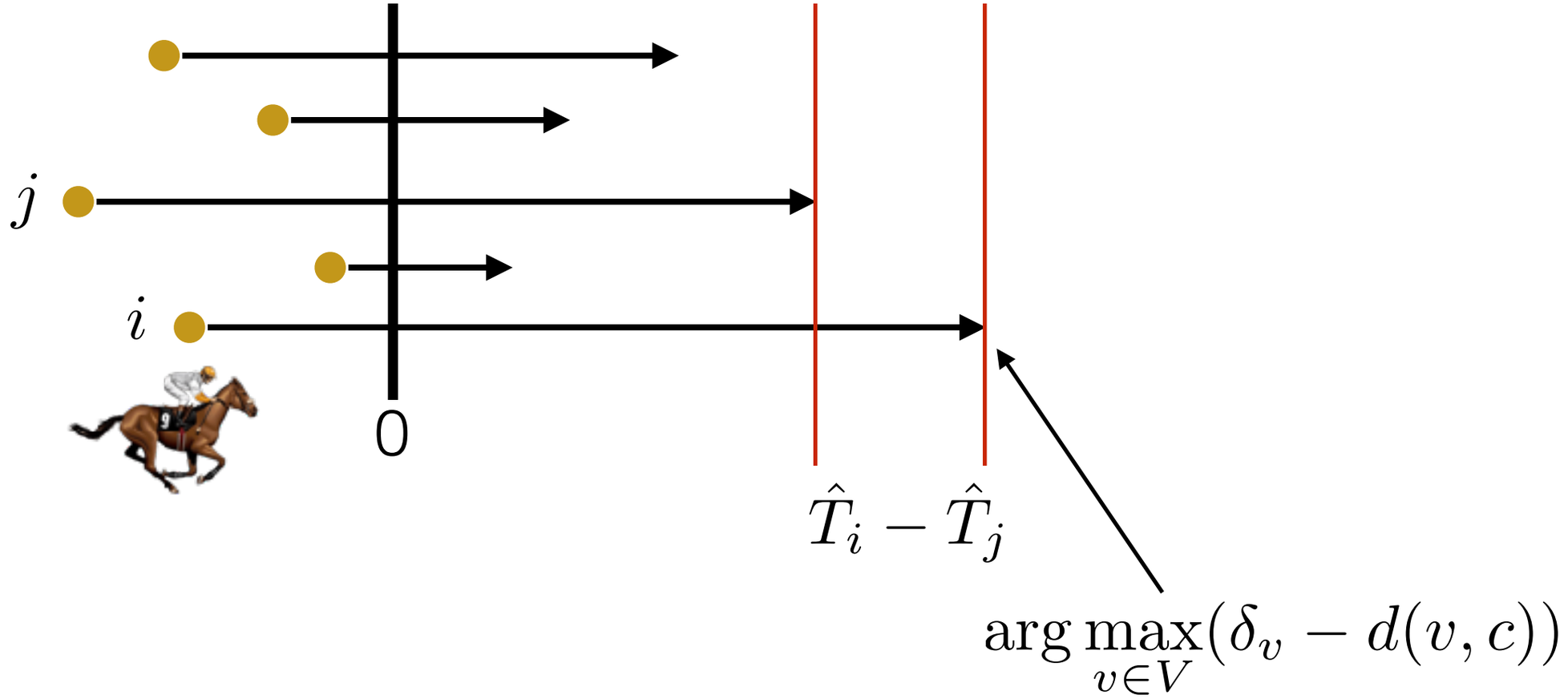
Source: 15-750 Spring 2017 notes

Parallel low-diameter decomposition

$$T_i = \delta_{\max} - \delta_i + d(i, c) \qquad \hat{T}_i = \delta_{\max} - T_i = \delta_i - d(i, c)$$

$$\arg\min_{v \in V}(\delta_{\max} - \delta_v + d(v, c)) = \arg\max_{v \in V}(\delta_v - d(v, c))$$

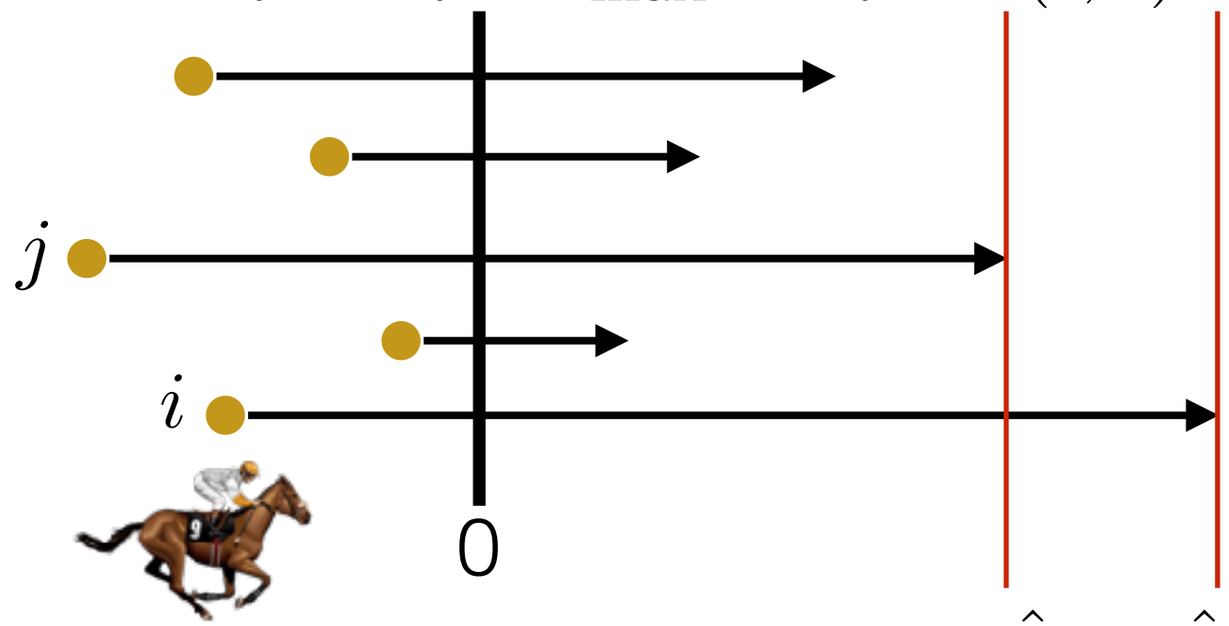
Note: Both expressions give the center that captures c



Source: 15-750 Spring 2017 notes

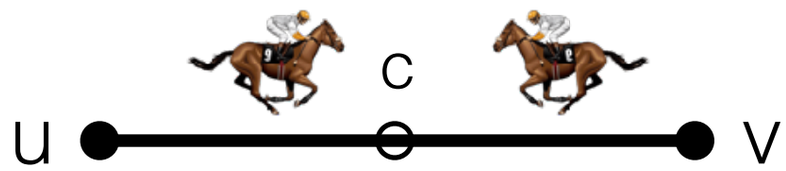
Parallel low-diameter decomposition

$$\hat{T}_i = T_i - \delta_{\max} = \delta_i - d(i,c)$$



$T_i - T_j$

- $\hat{T}_i - \hat{T}_j < 1$ is exactly the event that (u,v) is cut!



Note: $\hat{T}_i - \hat{T}_j \sim Exp(\beta)$ (memoryless property)

$$\mathbf{Pr}\left[\hat{T}_i - \hat{T}_j < 1\right] = 1 - e^{-\beta} < \beta$$

Source: 15-750 Spring 2017 notes

Parallel low-diameter decomposition

- Back to the algorithm:

```
LDD(G(V, E), beta) =
  n = |V|; num_finished = 0;
  E = array(n, lambda i.Exp(beta));
  S = array(n, lambda i.max(E) - E[i]);
  C = array(n, (infty, infty));
  num_processed = 0; round = 1;
  while (num_processed < n)
    F = F ∪ {v in V | S[v] < round, C[v] =
  num_processed += |F|;
  update = lambda (u,v).
    if (C[v].snd == infty)
      writeMin(&C[v].fst, S[u]);
    return false;
  edge_map(G, F, update);
  check = lambda (u,v).
    if (C[v].fst == S[u])
      C[v].snd = u;
      return true;
    return false;
  F = edge_map(G, F, check);
  round++;
return C;
```

$O(n)$ work, $O(\log n)$ depth

$O(\log n/\beta)$ rounds w.h.p.

Each round: $O(\log n)$ depth

edge_map: $O(m)$ work in total

L2: Acquire

L3: Set ngh's cluster id if we won

Parallel low-diameter decomposition

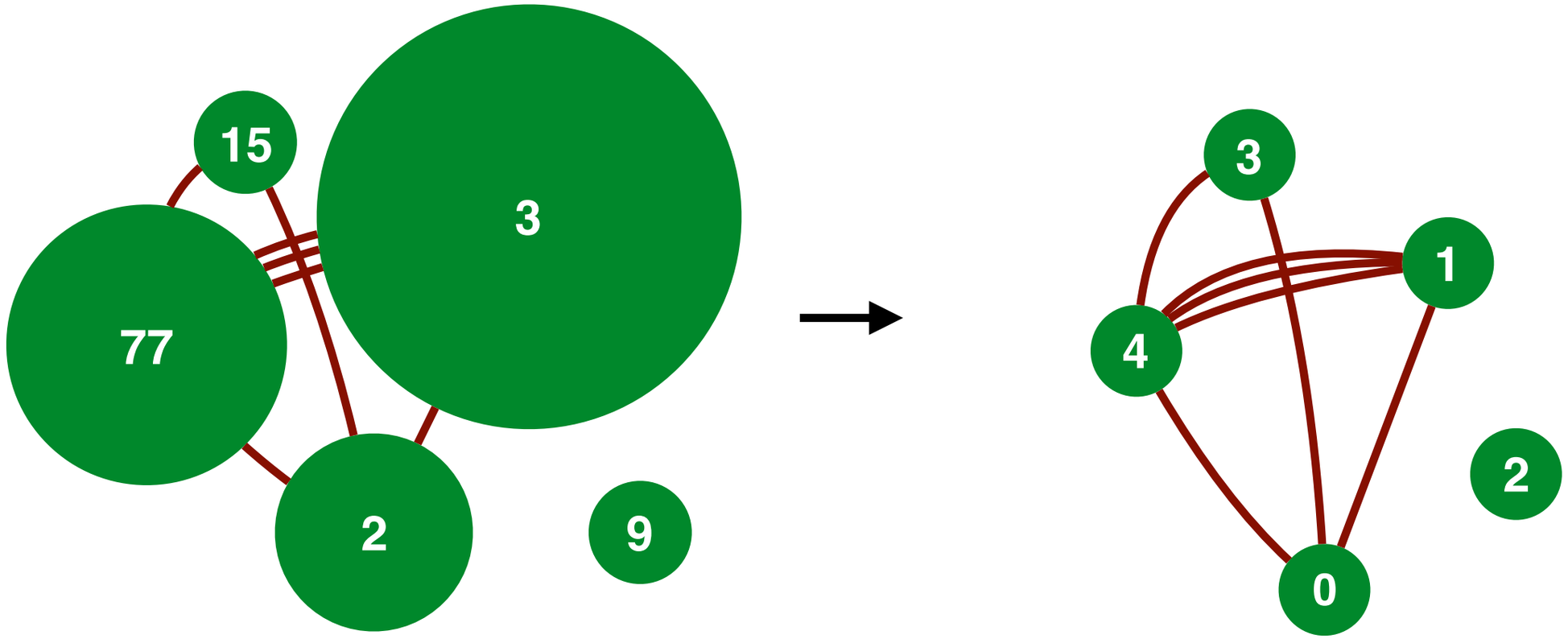
- MPX algorithm computes an $(\beta, O(\log n/\beta))$-decomp in
  - $O(m+n)$ expected work
  - $O(\log^2 n)$ depth w.h.p. (can be made $O(\log n \log^* n)$)

See for more details:

Parallel Graph Decompositions Using Random Shifts
Miller, Peng, Xu (SPAA 2013)

Improved Parallel Algorithms for Spanners and Hopsets
Miller, Peng, Vladu and Xu (SPAA 2015)

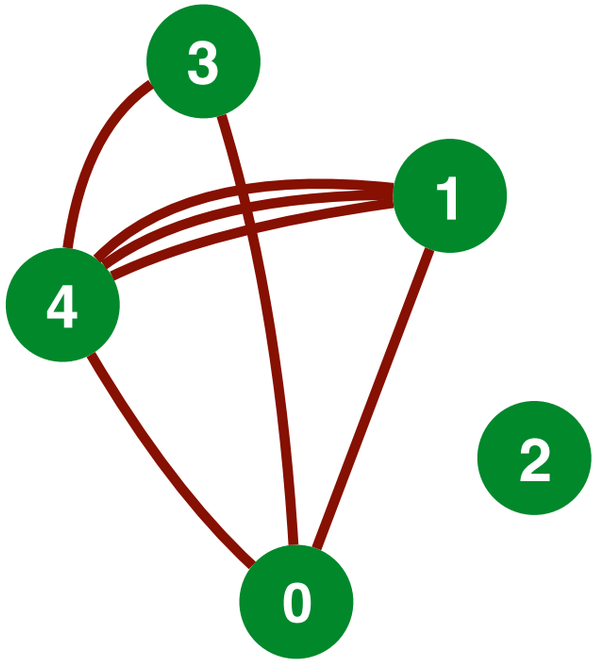
Work-efficient connectivity



Compute LDD

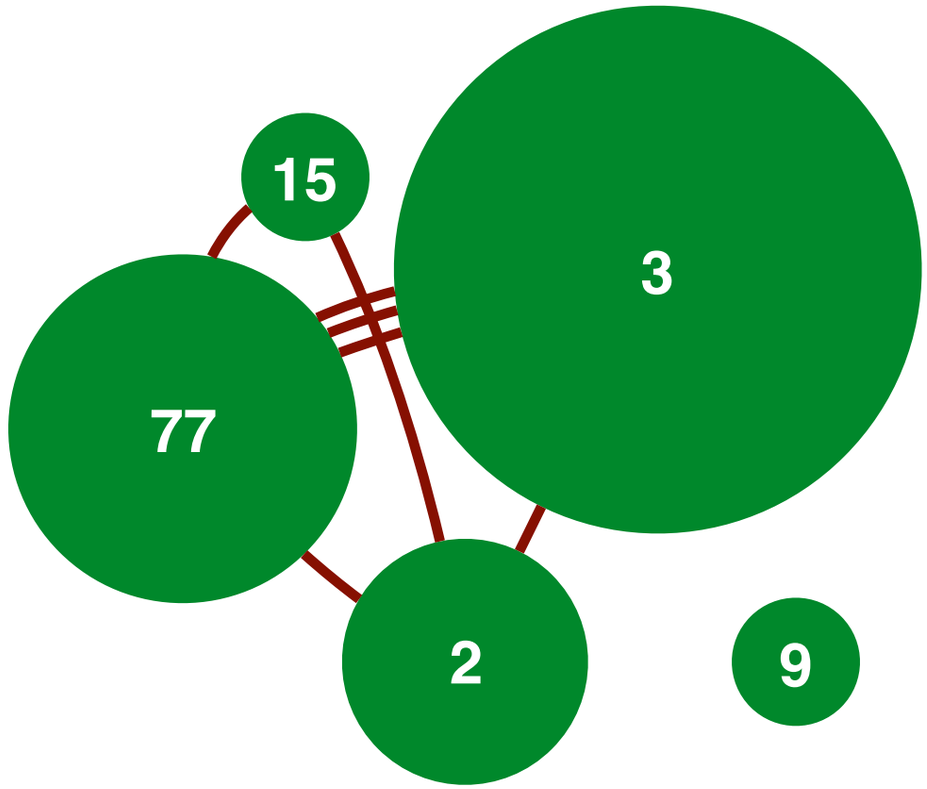
Contract and Recurse

Work-efficient connectivity



Contract and Recurse

L’ = [0, 0, 1, 0, 0]



Update labeling based on L’

L[i] = L’[cluster[i]]

Return L

Work-efficient connectivity

```
Connectivity(G(V, E), beta) =
  L = LDD(G, beta);
  G'(V',E') = Contract(G, L);
  if (|E'| == 0)
    return L
  L' = Connectivity(G', beta)
  L'' = array(n, lambda v.return L'[L[v]];);
  return L'';
```

- Assume contraction in O(m + n) work and O(log n) depth
- $\beta \cdot m$ edges after each round (expected)
  - $m + \beta \cdot m + \beta^2 \cdot m \ldots = O(m)$ work in expectation
- $O(\log n)$ levels w.h.p. => $O(\log^3 n)$ depth w.h.p.

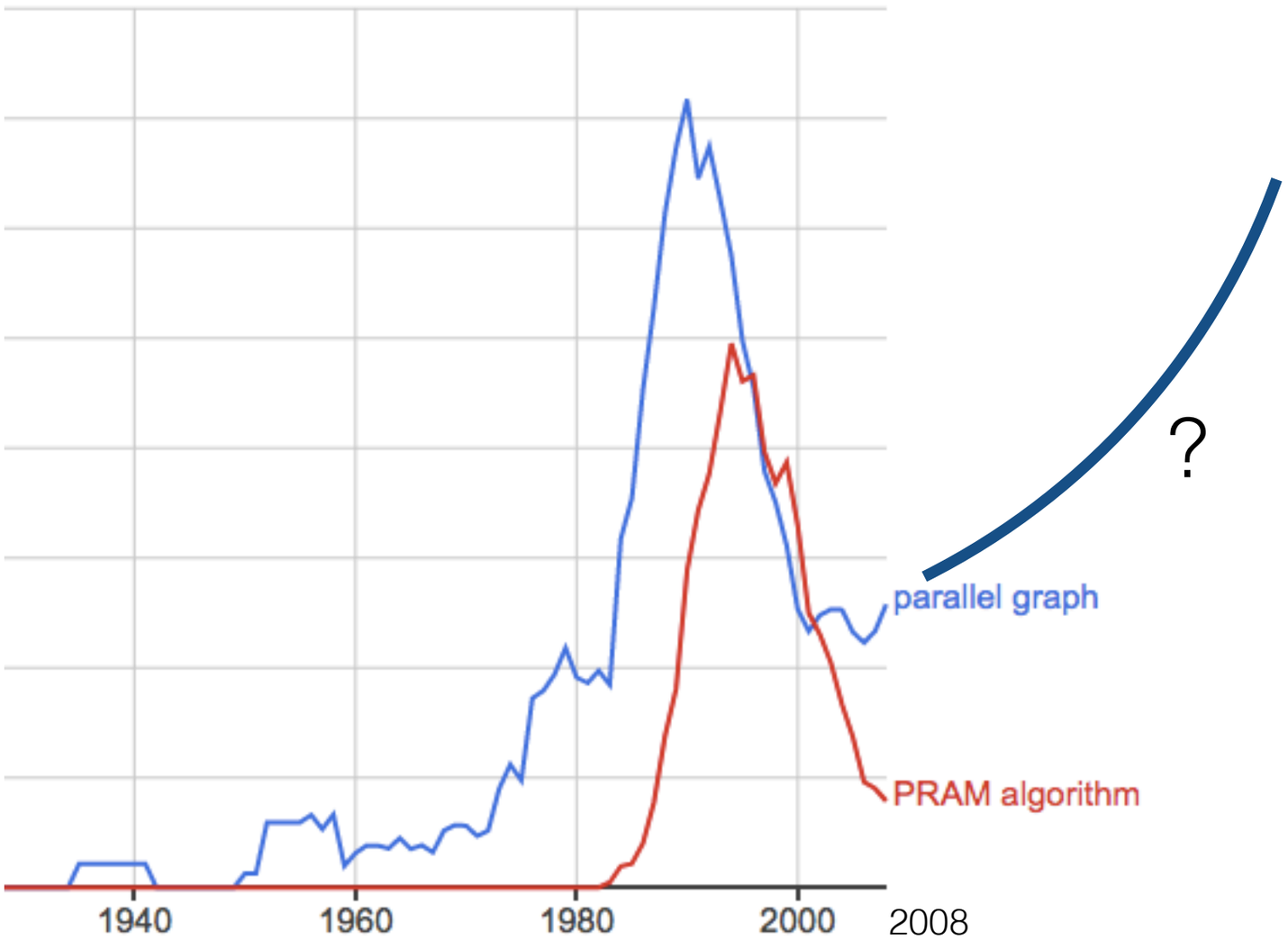
Work-efficient connectivity

- Connectivity can be solved in parallel in
  - $O(m+n)$ expected work
  - $O(\log^3 n)$ depth
- Depth can be improved to $O(\log n \log\log n \log^* n)$
- (Currently) only theoretically efficient connectivity algorithm that is also practical!

See for more details/experiments:
A Simple and Practical Linear-Work Parallel Algorithm for Connectivity
Shun, Dhulipala and Blelloch (SPAA 2014)

Are parallel graph algorithms practical?

- Huge amount of interest in 80s and 90s
- PRAM algorithms: lots of nice work, but hardware wasn't ready: never saw the gains promised by theory



Google n-gram viewer

Are parallel graph algorithms practical?

- Implement in cilk. Run on shared memory multicores, e.g.

Dell PowerEdge R930

- 72-cores (4 x 2.4GHz 18-core E7-8867 v4 Xeon processors)
- 1TB of main memory
- Costs less than a mid-range BMW

Example: k-core

- Ideas like work-efficiency matter!
- E.g. work-efficient vs work-inefficient k-core algorithms
- Run on the two largest publicly available graphs:

Graph	|V|	|E| (symmetrized)
Hyperlink2014	1.7B	124B
Hyperlink2012	3.5B	225B

$\rho =$ number of peeling steps done by the parallel algorithm

$W = O(|E| + \rho|V|)$

$D = O(\rho \log |V|)$

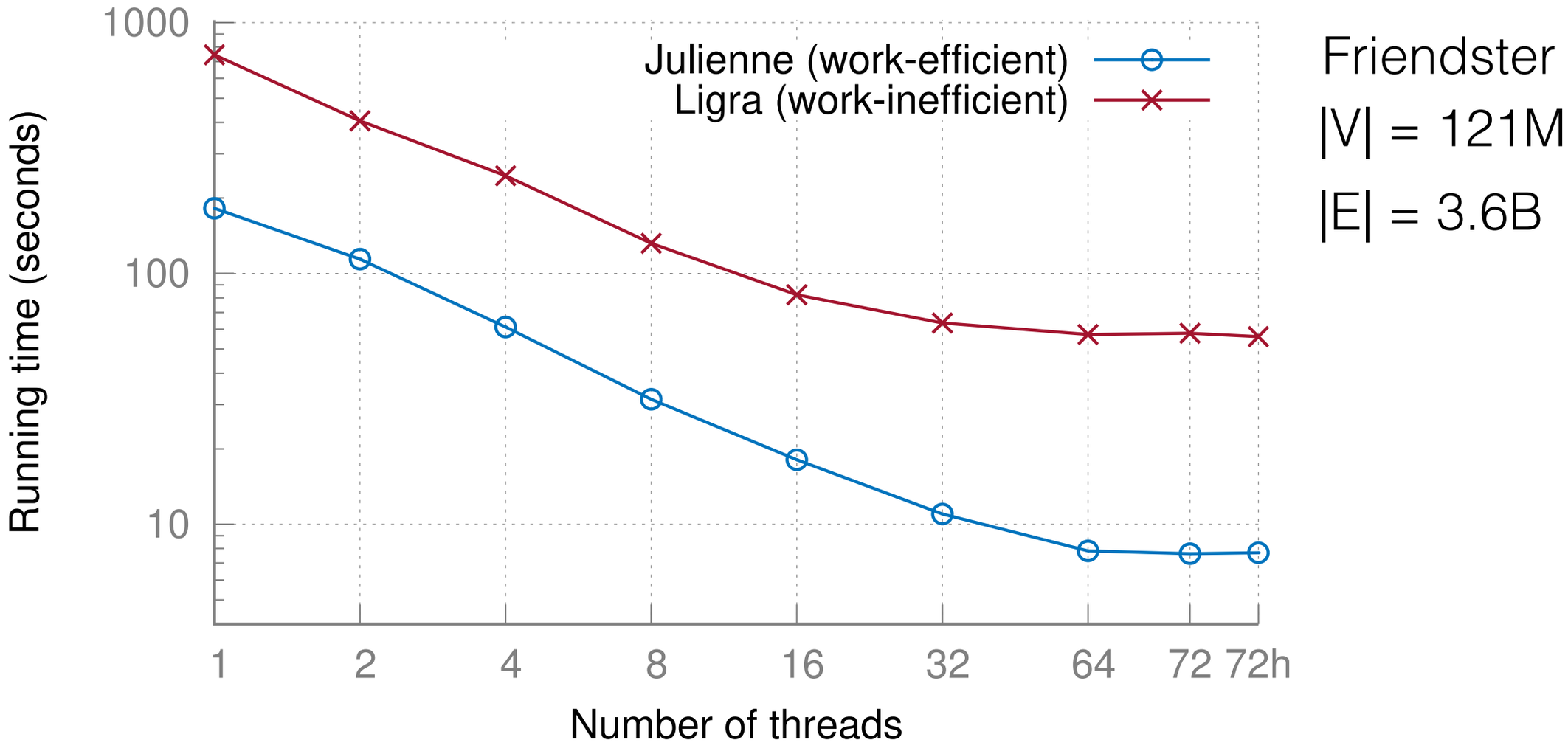
Work-inefficient

$O(|E| + |V|)$ expected work

$O(\rho \log |V|)$ depth w.h.p.

Work-efficient

Example: k-core



Friendster
|V| = 121M
|E| = 3.6B

Across all inputs:

- Between 4-41x speedup over sequential peeling
- Speedups are smaller on small graphs with large ρ
- 2-9x faster than work-inefficient implementation

Example: k-core

- Run on the two largest publicly available graphs:

Graph	|V|	|E| (symmetrized)
Hyperlink2014	1.7B	124B
Hyperlink2012	3.5B	225B

- On Hyperlink2012 graph our code takes 193s on 72h cores
- 8515s serially => 44x speedup
- Previous best time: 256-node cluster, each with 32 cores
  - 6 minutes to compute *approximate k-cores*
- We compute *exact k-cores*
  - 1.8x faster
  - using 113x fewer cores

Example: connectivity

Graph	|V|	|E|
Hyperlink2012	3.5B	225B

- Run connectivity on Hyperlink2012 graph
- 38.3s on 72h cores
- 2080s serially => 54x speedup
- Very recently, folks from Yahoo (Oath research) presented a new connectivity algorithm that runs in O(log n) rounds on BSP model
- Their algorithm runs in 341s using:
  - 1000 nodes, 24000 cores and 128Tb of memory

Our algorithm is 8x faster using 128x less memory and 333x fewer cores

- To be fair, their code runs on a graph with 272B vertices and 5.9T edges; out of reach of shared-memory for now…

Theoretically efficient parallel algorithms

Problem	Model	Work	Depth
Breadth-First Search	TS	$O(m)$	$O(\text{diam}(G)\log n)$
Integral-Weight SSSP (weighted BFS)	PW	$O(m)$ expected	$O(\text{diam}(G)\log n)$ w.h.p.*
General-Weight SSSP (Bellman-Ford)	PW	$O(\text{diam}(G)m)$	$O(\text{diam}(G)\log n)$
Single-Source Betweenness Centrality (BC)	FA	$O(m)$	$O(\text{diam}(G)\log n)$
Low-Diameter Decomposition	TS	$O(m)$ expected	$O(\log^2 n)$ w.h.p.
Connectivity	TS	$O(m)$ expected	$O(\log^3 n)$ w.h.p.
Biconnectivity	FA	$O(m)$ expected	$O(\max(\text{diam}(G)\log n, \log^3 n))$ w.h.p.
Strongly Connected Components	PW	$O(m \log n)$ expected	$O(\text{diam}(G)\log n)$ w.h.p.
Minimum Spanning Forest	PW	$O(m \log n)$	$O(\log^2 n)$
Maximal Independent Set	FA	$O(m)$ expected	$O(\log^2 n)$ w.h.p.
Maximal Matching	PW	$O(m)$ expected	$O(\log^3 m / \log\log m)$ w.h.p.
Graph Coloring	FA	$O(m+n)$	$O(\log n + L\log\Delta)$
k-core	FA	$O(m+n)$ expected	$O(\rho \log n)$ w.h.p.
Approximate Set Cover	PW	$O(m)$ expected	$O(\log^3 n)$ w.h.p.
Triangle Counting	-	$O(m^{3/2})$	$O(\log n)$

Based on 30 years of research on parallel algorithms

Theoretically efficient parallel algorithms

Problem	(1)	(72h)	(S)
Breadth-First Search	631	12	52
Integral-Weight SSSP (weighted BFS)	4700	58.1	80
General-Weight SSSP (Bellman-Ford)	3180	53	60
Single-Source Betweenness Centrality (BC)	3170	40	79
Low-Diameter Decomposition	464	12	38
Connectivity	2080	38.3	54
Biconnectivity	—	201	—
Strongly Connected Components	7720	182	42
Minimum Spanning Forest	—	228	—
Maximal Independent Set	2210	34	65
Maximal Matching	11000	140	78
Graph Coloring	12200	174	70
k-core	8515	193	44
Approximate Set Cover	3720	104	35
Triangle Counting	—	1470	—

Running times in seconds on Hyperlink2012

All implementations are theoretically efficient and scalable!