Introduction to Parallel and Distributed Computing

Selected Parallel Algorithms
Graph Algorithms

Lecture 14, Spring 2024

Instructor: 罗国杰

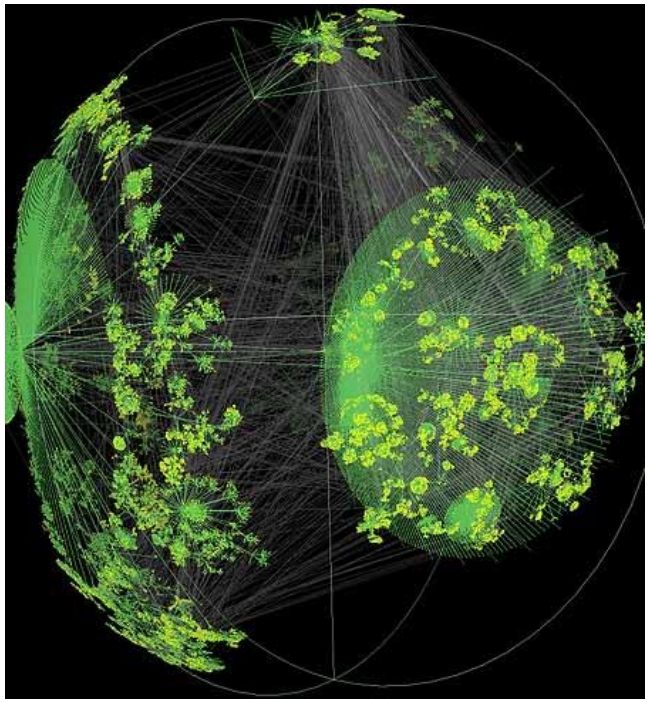
gluo@pku.edu.cn

Outline

- **Parallel graph algorithms**
  - Graph and sparse matrix
  - Independent set
  - Strongly-connected components
  - Shortest path and Δ-stepping
- **And we will introduce parallel & distributed graph processing frameworks in later lectures.**

Large Graphs are Everywhere…

Internet structure
Social interactions



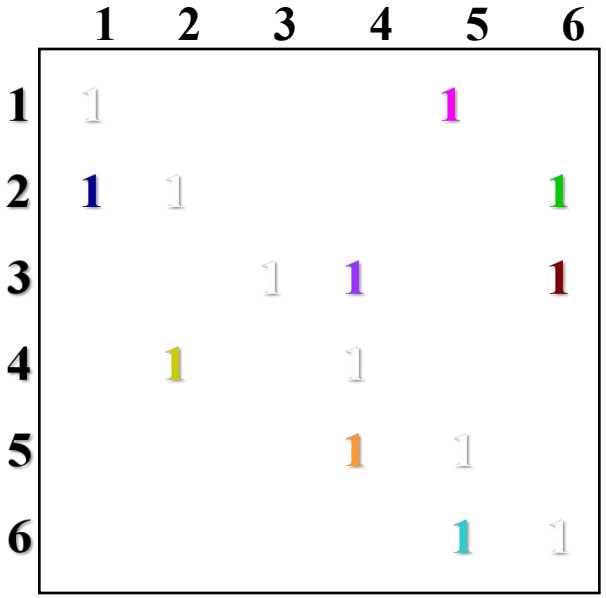
WWW snapshot, courtesy Y. Hyun

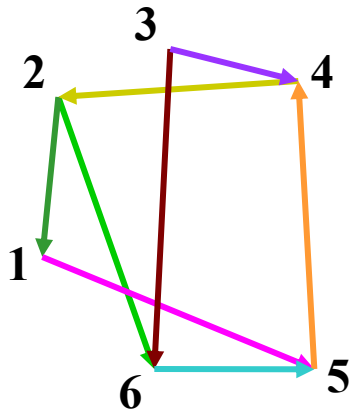
Scientific datasets: biological, chemical, cosmological, ecological, …

Yeast protein interaction network, courtesy H. Jeong

Graphs and Sparse Matrices

- **Sparse matrix is a representation of a (sparse) graph**

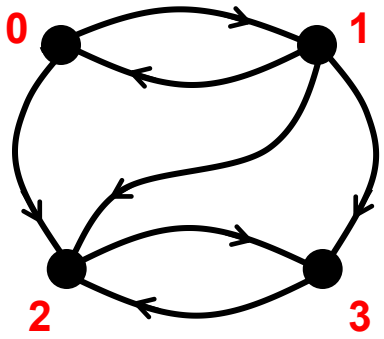


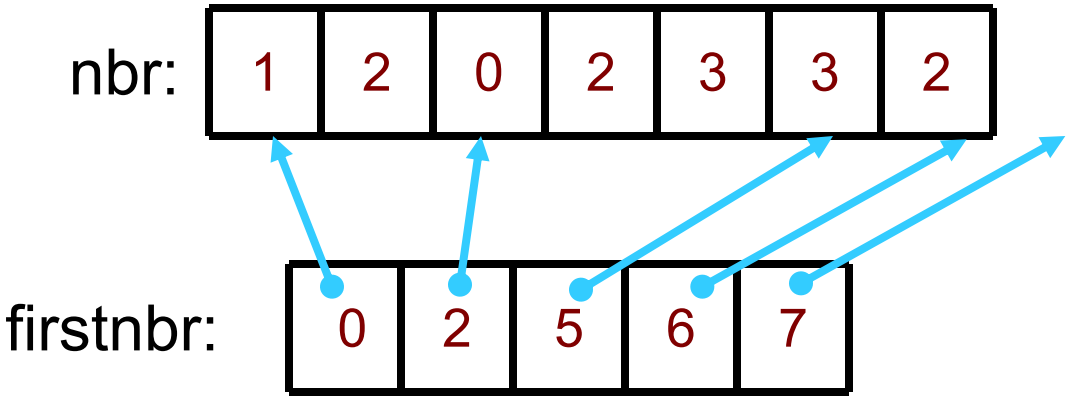


  - Matrix entries can be just 1's, or edge weights
  - Diagonal can represent self-loops or vertex weights
  - Nnz per row (off diagonal) is vertex out-degree

Compressed Graph Data Structure (CSR)

Like matrix CSR



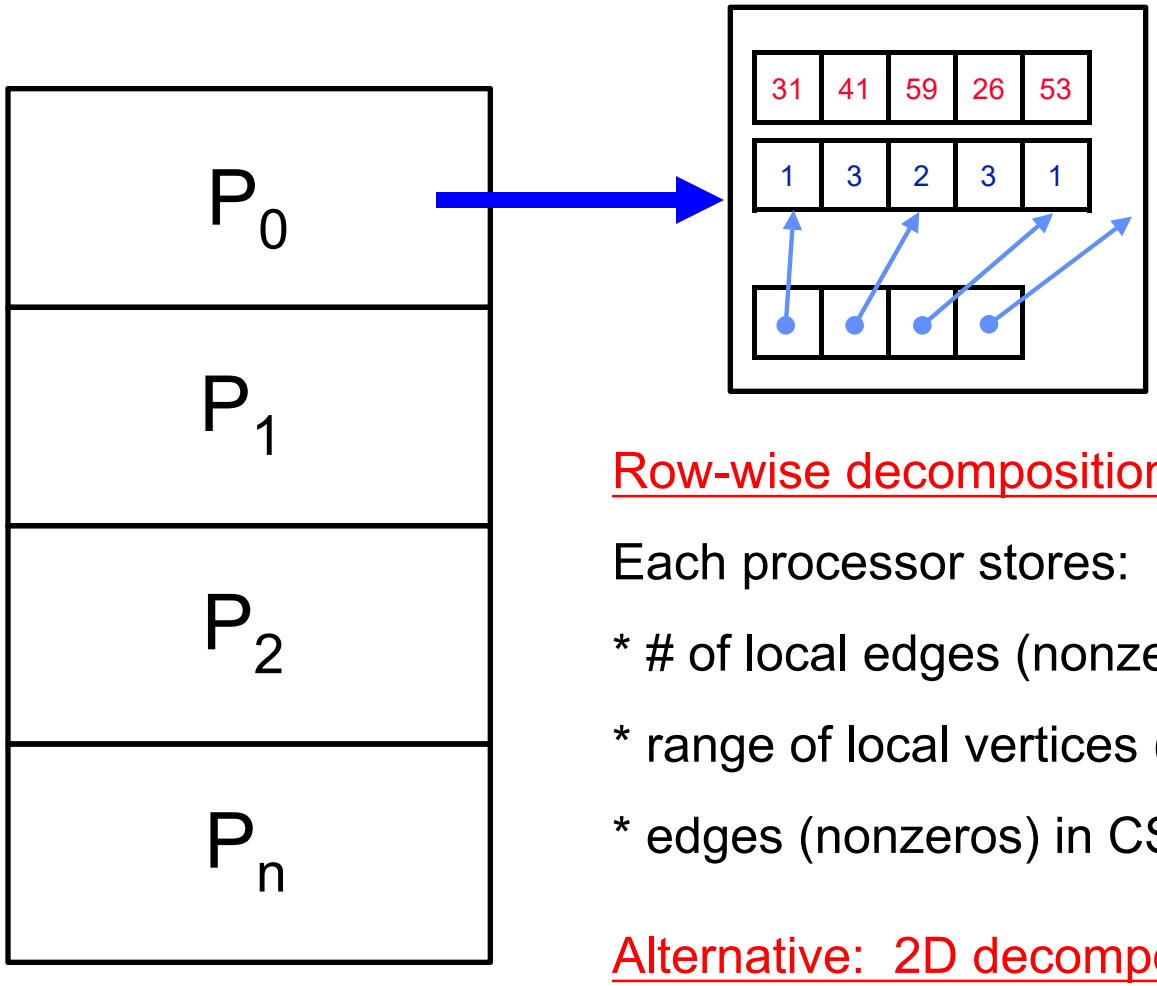


CSR graph storage:

- three 1-dimensional arrays
- digraph: $n_E + n_V + 1$ memory
- undirected graph: $2*n_E + n_V + 1$ memory; edge {v,w} appears once for v, once for w
- firstnbr[0] = 0; for a digraph, firstnbr[n_V] = n_E

J. Gilbert, "Applied Parallel Computing", UCSB CS240A, Spring 2011.

Graph (CSR) in Distributed Memory



Row-wise decomposition

Each processor stores:

* # of local edges (nonzeros)

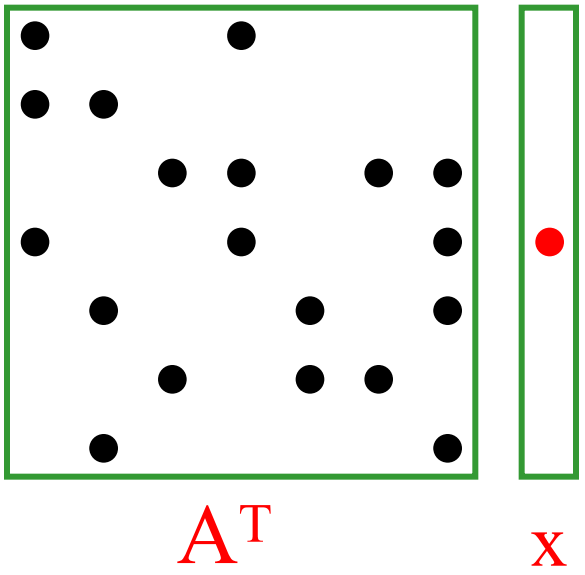
* range of local vertices (rows)

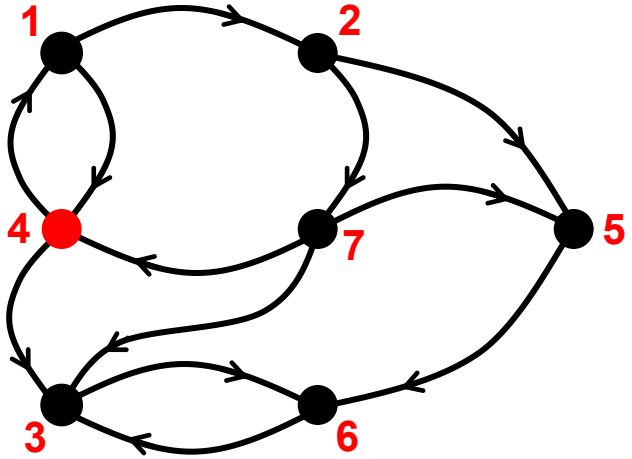
* edges (nonzeros) in CSR form

Alternative: 2D decomposition

J. Gilbert, "Applied Parallel Computing", UCSB CS240A, Spring 2011.

Breadth-First Search as SpMV

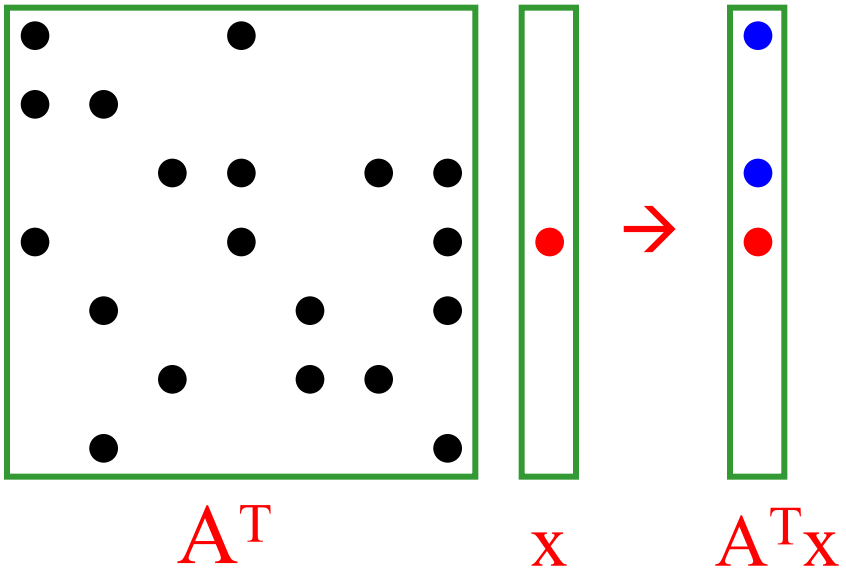


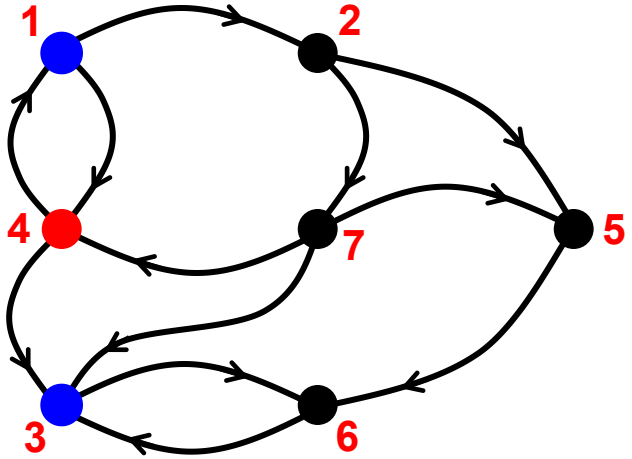


- Multiply by adjacency matrix → step to neighbor vertices
- Work-efficient implementation from sparse data structures

J. Gilbert, "Applied Parallel Computing", UCSB CS240A, Spring 2011.

Breadth-First Search as SpMV

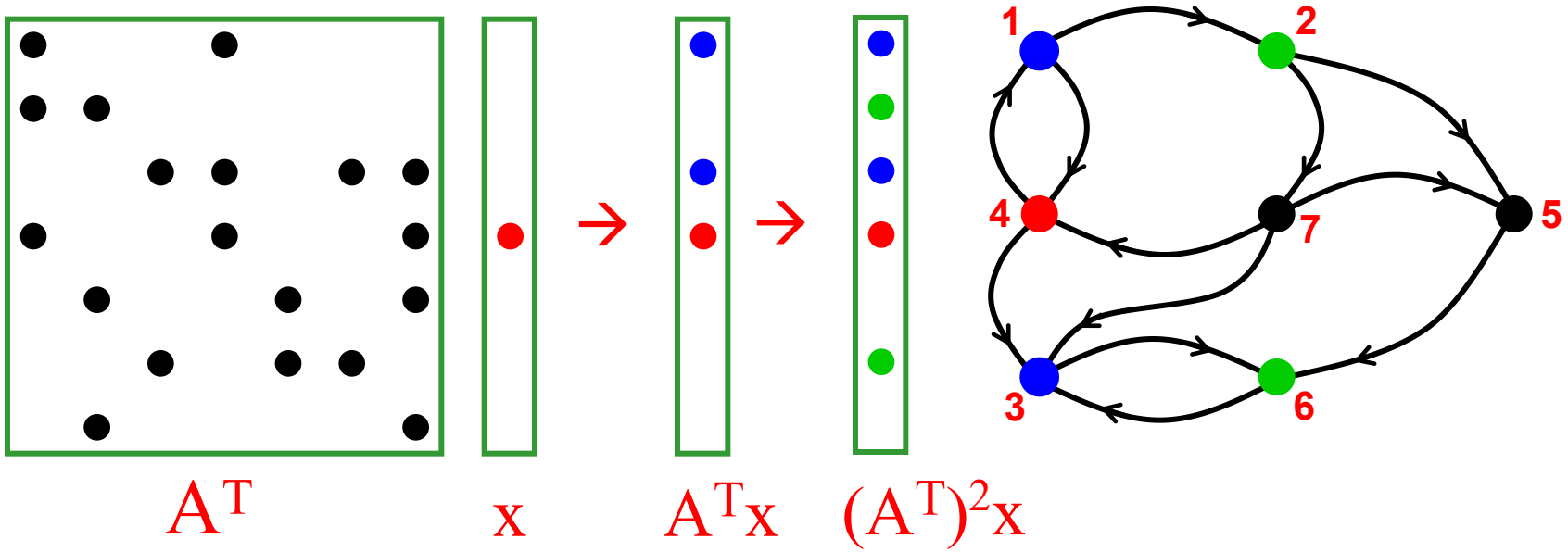




- Multiply by adjacency matrix → step to neighbor vertices
- Work-efficient implementation from sparse data structures

J. Gilbert, "Applied Parallel Computing", UCSB CS240A, Spring 2011.

Breadth-First Search as SpMV



- Multiply by adjacency matrix → step to neighbor vertices
- Work-efficient implementation from sparse data structures

J. Gilbert, "Applied Parallel Computing", UCSB CS240A, Spring 2011.

More Graph Algorithms in Linear Algebra

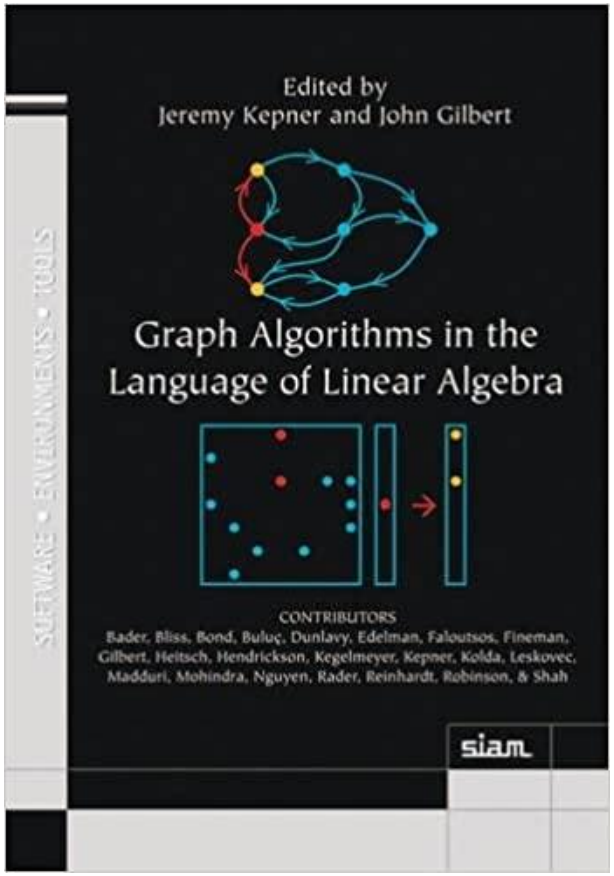
semiring	domain	$\oplus$	$\otimes$	0	graph semantics
any-pair	$\{T, F\}$	any	pair	F	traversal step
integer arithmetic	$\mathbb{N}$	$+$	$\times$	0	number of paths
min-plus	$\mathbb{R} \cup \{+\infty\}$	min	$+$	$+\infty$	shortest path

problem	algorithm	canonical complexity Θ	LA-based complexity Θ
breadth-first search		m	m
single-source shortest paths	Dijkstra	$m + n \log n$	n^2
	Bellman-Ford	mn	mn
all-pairs shortest paths	Floyd-Warshall	n^3	n^3
minimum spanning tree	Prim	$m + n \log n$	n^2
	Borůvka	$m \log n$	$m \log n$
maximum flow	Edmonds-Karp	$m^2 n$	$m^2 n$
maximal independent set	greedy	$m + n \log n$	$mn + n^2$
	Luby	$m + n \log n$	$m \log n$

Table from: Gábor Szárnyas's tutorial "Introduction to GraphBLAS"

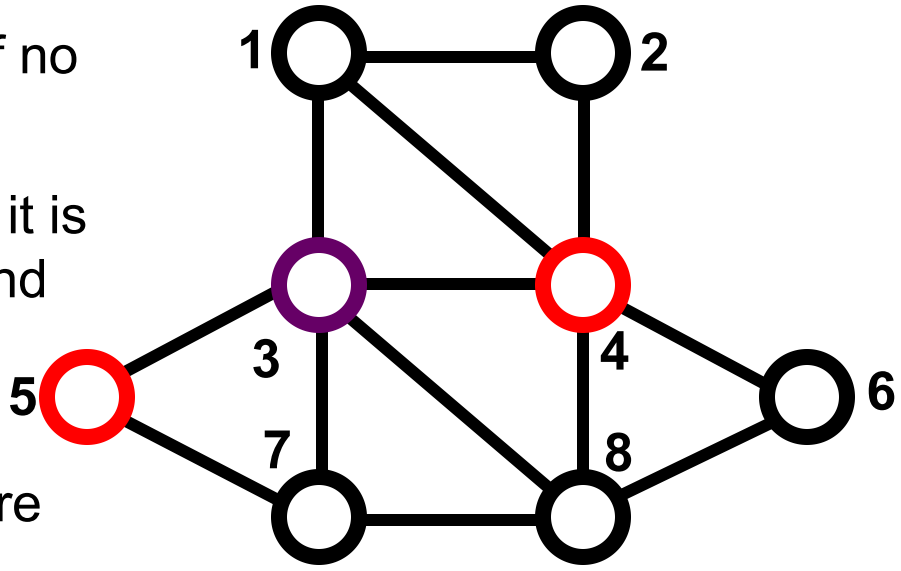
Further Readings on Graph & SpMV

- **GraphBLAS forum**
  - **https://graphblas.org/**
- **Jeremy Kepner and John Gilbert, "Graph Algorithms in the Language of Linear Algebra," Society for Industrial and Applied Mathematics, 2011.**



Graph Problem: Maximal Independent Set

- Graph with vertices V = {1,2,…,n}
- A set S of vertices is independent if no two vertices in S are neighbors.
- An independent set S is maximal if it is impossible to add another vertex and stay independent
- An independent set S is maximum if no other independent set has more vertices
- Finding a *maximum* independent set is intractably difficult (NP-hard)
- Finding a *maximal* independent set is easy, at least on one processor.



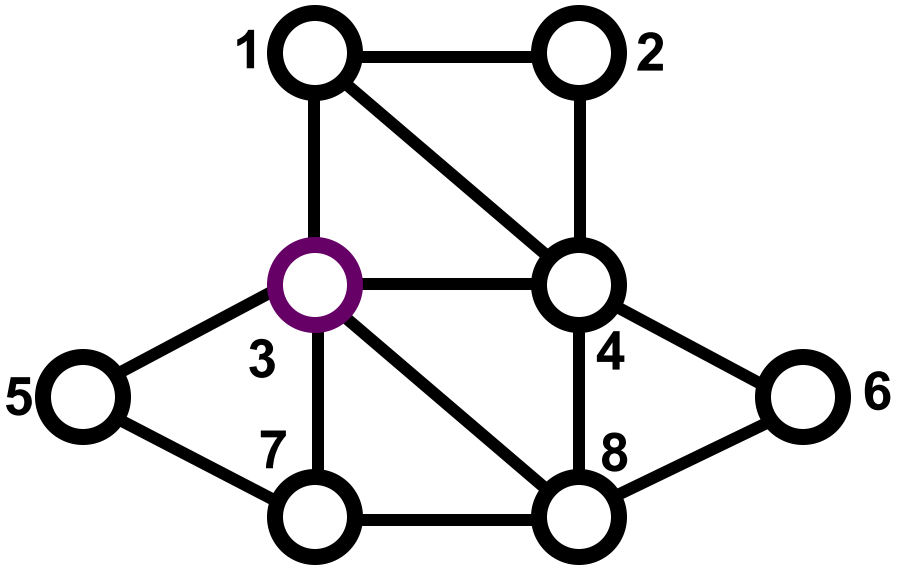
The set of red vertices
S = {4, 5} is *independent*
and is *maximal*
but not *maximum*

MIS: Applications in Distributed Systems

- **In a network graph consisting of nodes representing processors, a MIS defines a set of processors which can operate in parallel without interference**
  - For instance, in wireless ad hoc networks, to avoid interference, a conflict graph is built, and a MIS on that defines a clustering of the nodes enabling efficient routing

Sequential MIS Algorithm

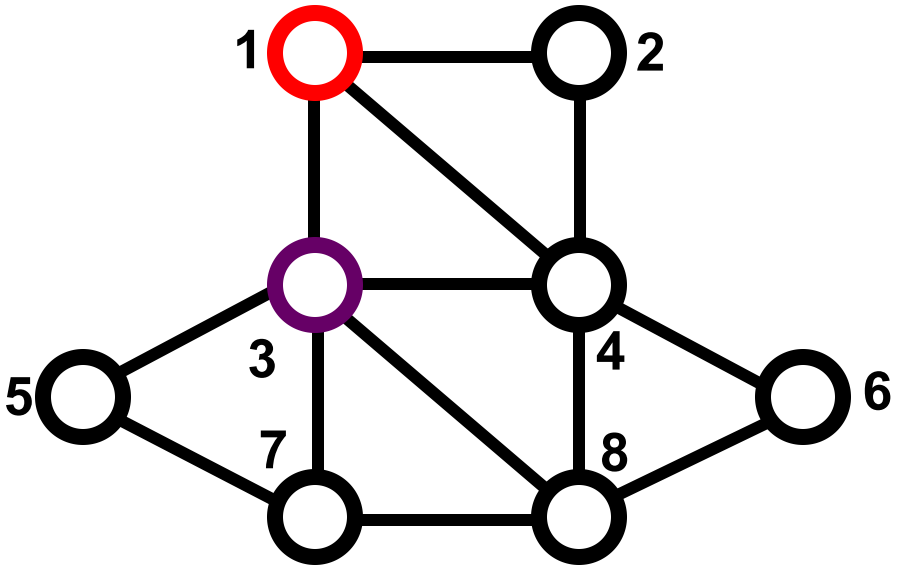
1. S = empty set;
2. for vertex v = 1 to n {
3. if (v has no neighbor in S) {
4. add v to S
5. }
6. }



S = { }

Sequential MIS Algorithm

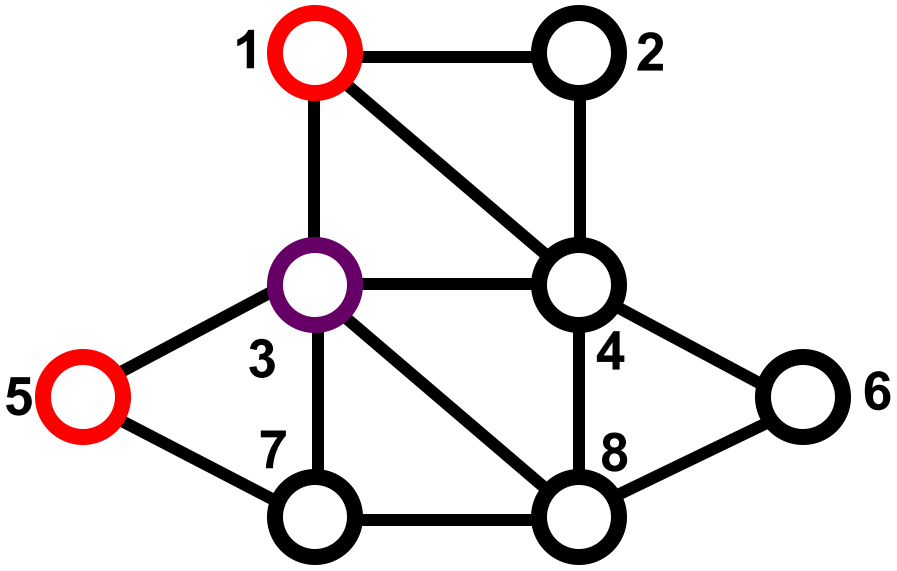
```
1.  S = empty set;
2.  for  vertex v = 1 to n {
3.      if (v has no neighbor in S) {
4.          add v to S
5.      }
6.  }
```



S = { 1 }

Sequential MIS Algorithm

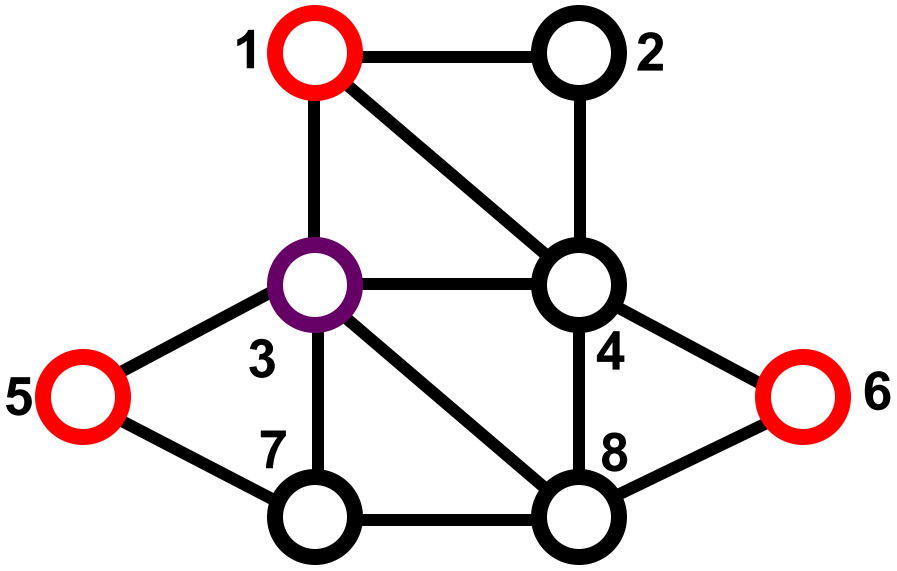
```
1.  S = empty set;
2.  for  vertex v = 1 to n {
3.      if (v has no neighbor in S) {
4.          add v to S
5.      }
6.  }
```



S = { 1, 5 }

Sequential MIS Algorithm

```
1.  S = empty set;
2.  for  vertex v = 1 to n {
3.      if (v has no neighbor in S) {
4.          add v to S
5.      }
6.  }
```

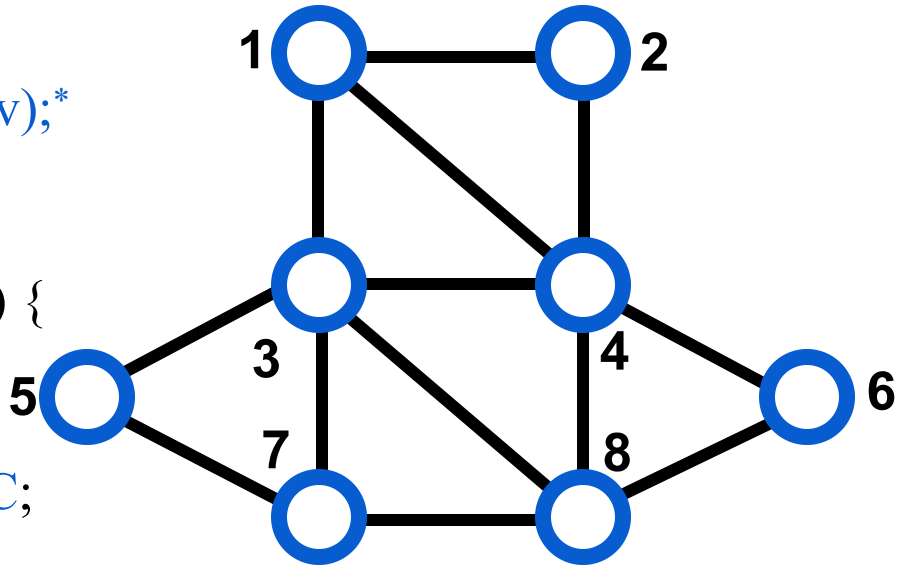


S = { 1, 5, 6 }

work **~ O(n), but** ***span*** **~O(n) and maximum speedup ~O(1)**

Common Implementation of the Parallel, Randomized MIS Algorithm

```
1.  S = empty set;  C = V;
2.  while  C  is not empty {
3.      label each v in C with a random r(v);*
4.      for all v in C in parallel {
5.          if r(v) < min( r(neighbors of v) ) {
6.              move v from C to S;
7.              remove neighbors of v from C;
8.          }
9.      }
10. }
```



S = { }

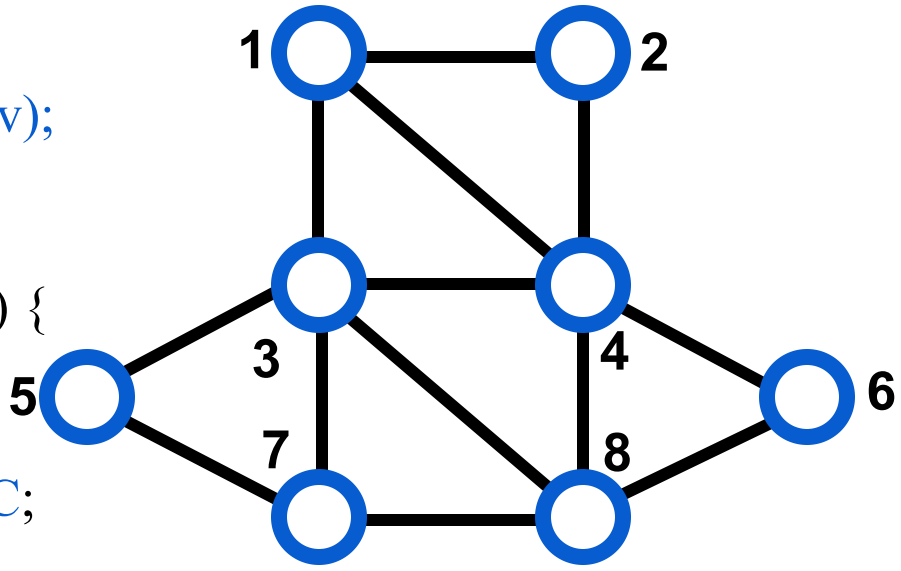
C = { 1, 2, 3, 4, 5, 6, 7, 8 }

* (simplified version with some details omitted)

M. Luby. "A Simple Parallel Algorithm for the Maximal Independent Set Problem". *SIAM Journal on Computing* **15** (4): 1036–1053, 1986

Common Implementation of the Parallel, Randomized MIS Algorithm

```
1.  S = empty set;  C = V;
2.  while  C  is not empty {
3.      label each v in C with a random r(v);
4.      for all v in C in parallel {
5.          if r(v) < min( r(neighbors of v) ) {
6.              move v from C to S;
7.              remove neighbors of v from C;
8.          }
9.      }
10. }
```

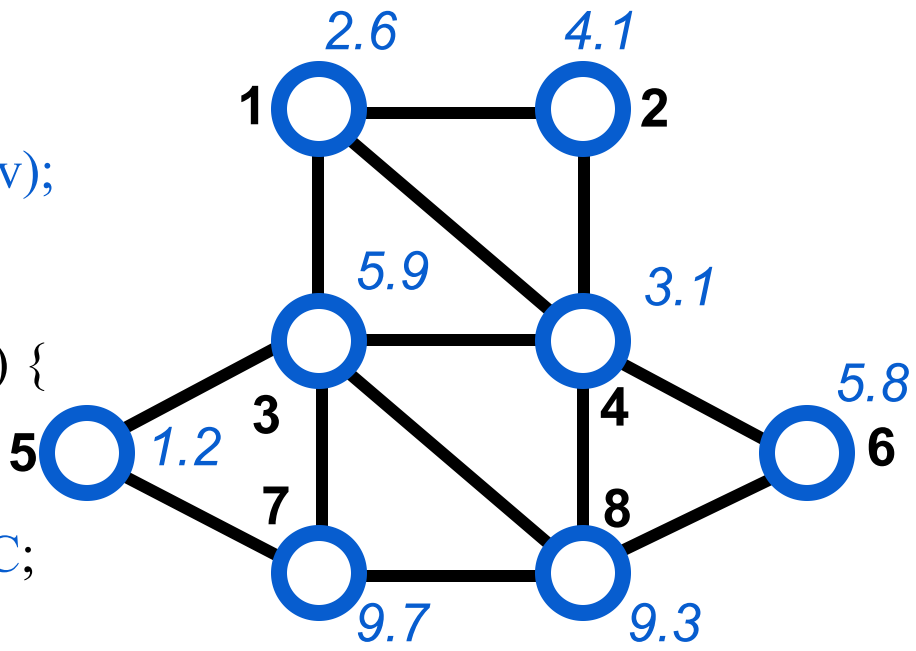


S = { }

C = { 1, 2, 3, 4, 5, 6, 7, 8 }

Common Implementation of the Parallel, Randomized MIS Algorithm

```
1.  S = empty set;  C = V;
2.  while  C  is not empty {
3.      label each v in C with a random r(v);
4.      for all v in C in parallel {
5.          if r(v) < min( r(neighbors of v) ) {
6.              move v from C to S;
7.              remove neighbors of v from C;
8.          }
9.      }
10. }
```

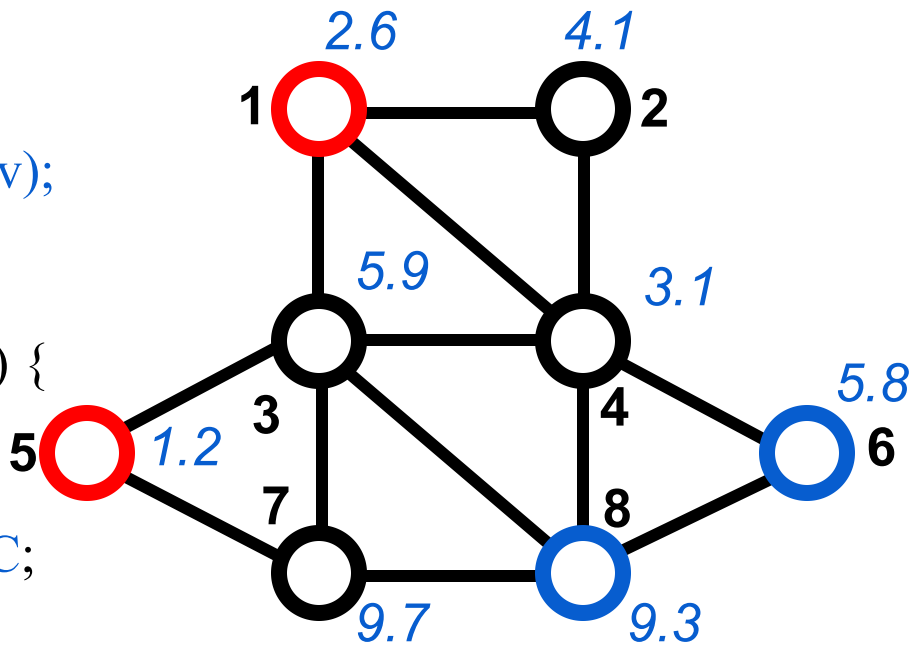


S = { }

C = { 1, 2, 3, 4, 5, 6, 7, 8 }

Common Implementation of the Parallel, Randomized MIS Algorithm

```
1.  S = empty set;  C = V;
2.  while  C  is not empty {
3.      label each v in C with a random r(v);
4.      for all v in C in parallel {
5.          if r(v) < min( r(neighbors of v) ) {
6.              move v from C to S;
7.              remove neighbors of v from C;
8.          }
9.      }
10. }
```

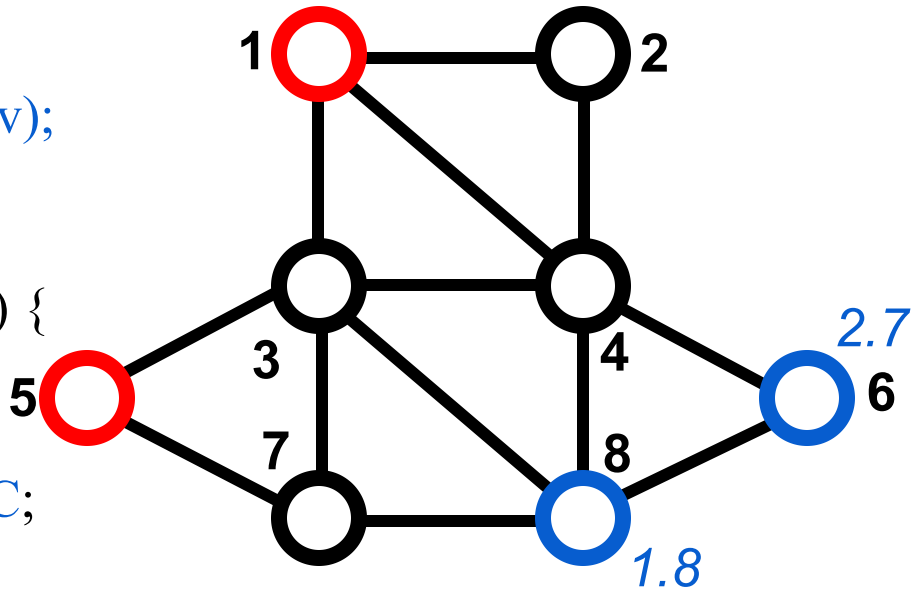


S = { 1, 5 }

C = { 6, 8 }

Common Implementation of the Parallel, Randomized MIS Algorithm

```
1.  S = empty set;  C = V;
2.  while  C  is not empty {
3.      label each v in C with a random r(v);
4.      for all v in C in parallel {
5.          if r(v) < min( r(neighbors of v) ) {
6.              move v from C to S;
7.              remove neighbors of v from C;
8.          }
9.      }
10. }
```

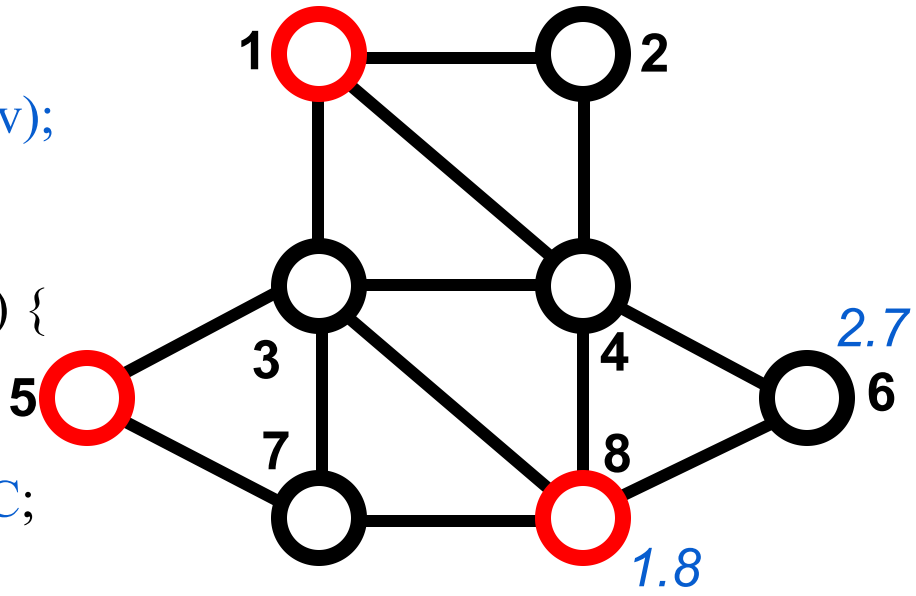


S = { 1, 5 }

C = { 6, 8 }

Common Implementation of the Parallel, Randomized MIS Algorithm

```
1.  S = empty set;  C = V;
2.  while  C  is not empty {
3.      label each v in C with a random r(v);
4.      for all v in C in parallel {
5.          if r(v) < min( r(neighbors of v) ) {
6.              move v from C to S;
7.              remove neighbors of v from C;
8.          }
9.      }
10. }
```

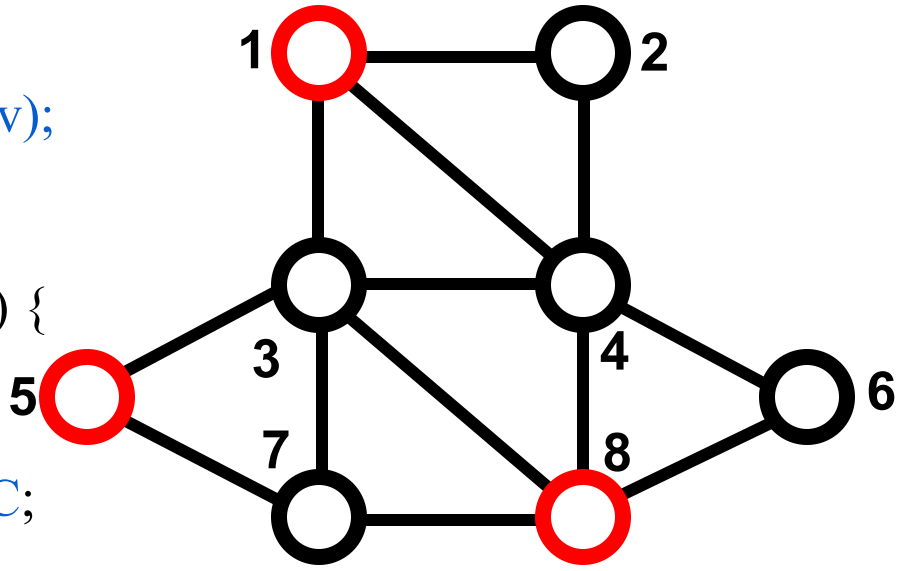


S = { 1, 5, 8 }

C = { }

Common Implementation of the Parallel, Randomized MIS Algorithm

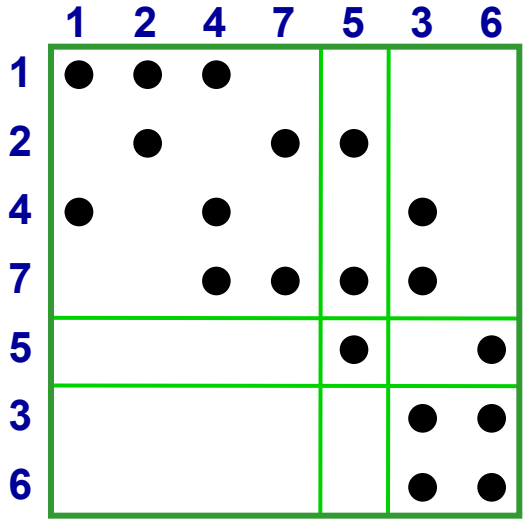
```
1.  S = empty set;  C = V;
2.  while  C  is not empty {
3.      label each v in C with a random r(v);
4.      for all v in C in parallel {
5.          if r(v) < min( r(neighbors of v) ) {
6.              move v from C to S;
7.              remove neighbors of v from C;
8.          }
9.      }
10. }
```

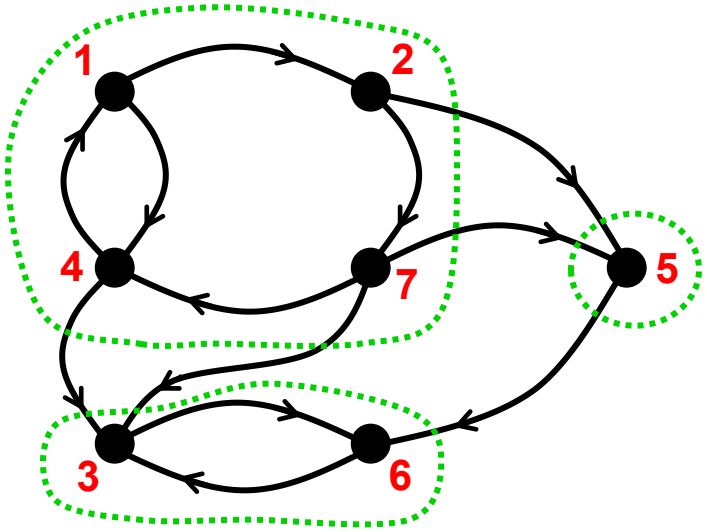


Theorem: This algorithm "very probably" finishes within O(log n) rounds.

work **~ O(n log n), but time ~O(log n)**

Strongly Connected Components (SCC)





- **Symmetric permutation to block triangular form**
- **Find P in linear time by depth-first search [Tarjan]**

J. Gilbert, "Applied Parallel Computing", UCSB CS240A, Spring 2011.

SCC of Directed Graph

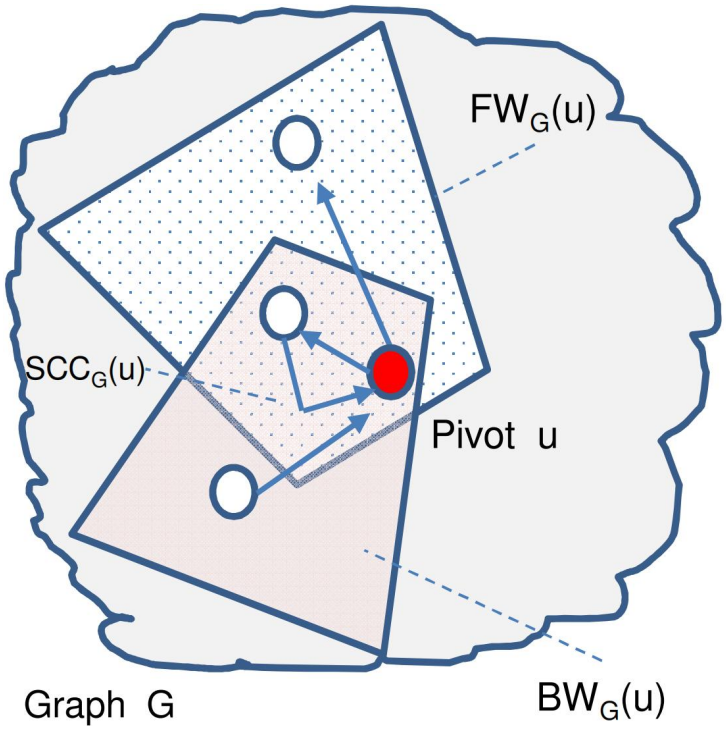
- **Sequential: depth-first search [Tarjan]; $O(n_V+n_E)$.**
- **DFS seems to be inherently sequential.**

- **Parallel: divide-and-conquer and BFS [Fleischer et al.]; worst-case span O(n) but good in practice on many graphs.**

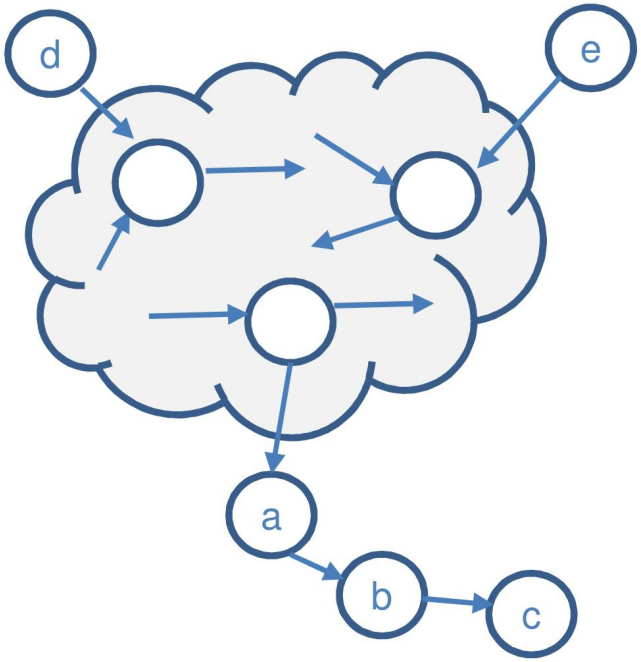
SCC: Conventional Parallel Algorithms

- **Basic idea: divide-and-conquer + BFS**
- **FW-BW algorithm**
  - L. Fleischer, B. Hendrickson, and A. Pınar. On identifying strongly connected components in parallel. Parallel and Distributed Processing, pages 505–511, 2000.
- **FW-BW-Trim algorithm**
  - W. McLendon III, B. Hendrickson, S. Plimpton, and L. Rauchwerger. Finding strongly connected components in distributed graphs. Journal of Parallel and Distributed Computing, 65(8):901–910, 2005.

SCC: Conventional Parallel Algorithms



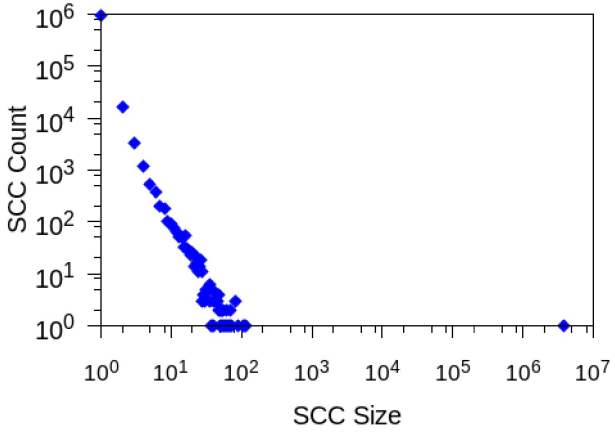
FW-BW



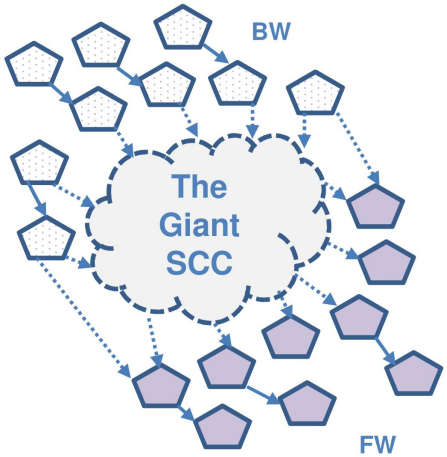
Trim

image source: S. Hong, N. C. Rodia, and K. Olukotun, "On fast parallel detection of strongly connected components (SCC) in small-world graphs," SC 2013.

Parallel SCC for Small-World Graphs

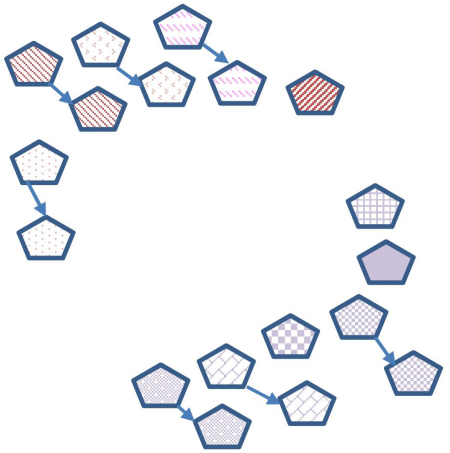


A small-world graph example



Phase 1

data-level parallelism



Phase 2

task-level parallelism

image source: S. Hong, N. C. Rodia, and K. Olukotun, "On fast parallel detection of strongly connected components (SCC) in small-world graphs," SC 2013.

Parallel SCC for Small-World Graphs

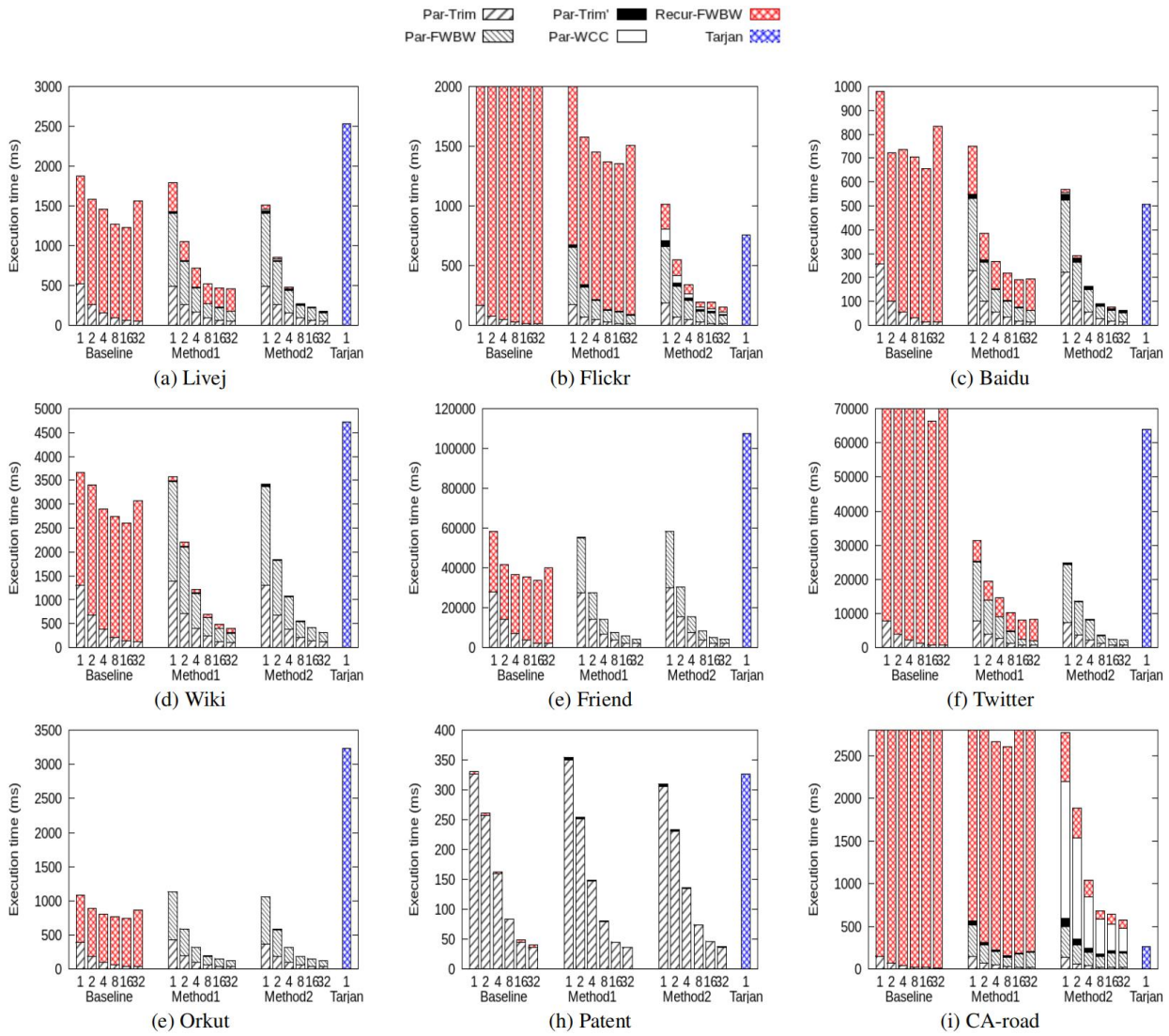


image source: S. Hong, N. C. Rodia, and K. Olukotun, SC 2013.

Single-Source Shortest Paths (SSSP)

- **For a weighted graph $G = (V,E,w)$, the *single-source all-sinks shortest paths* problem is to find the shortest paths from a vertex $v \in V$ to all other vertices in V.**
- **Dijkstra's algorithm is similar to Prim's algorithm. It maintains a set of nodes for which the shortest paths are known.**
- **It grows this set based on the node closest to source using one of the nodes in the current shortest path set.**

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

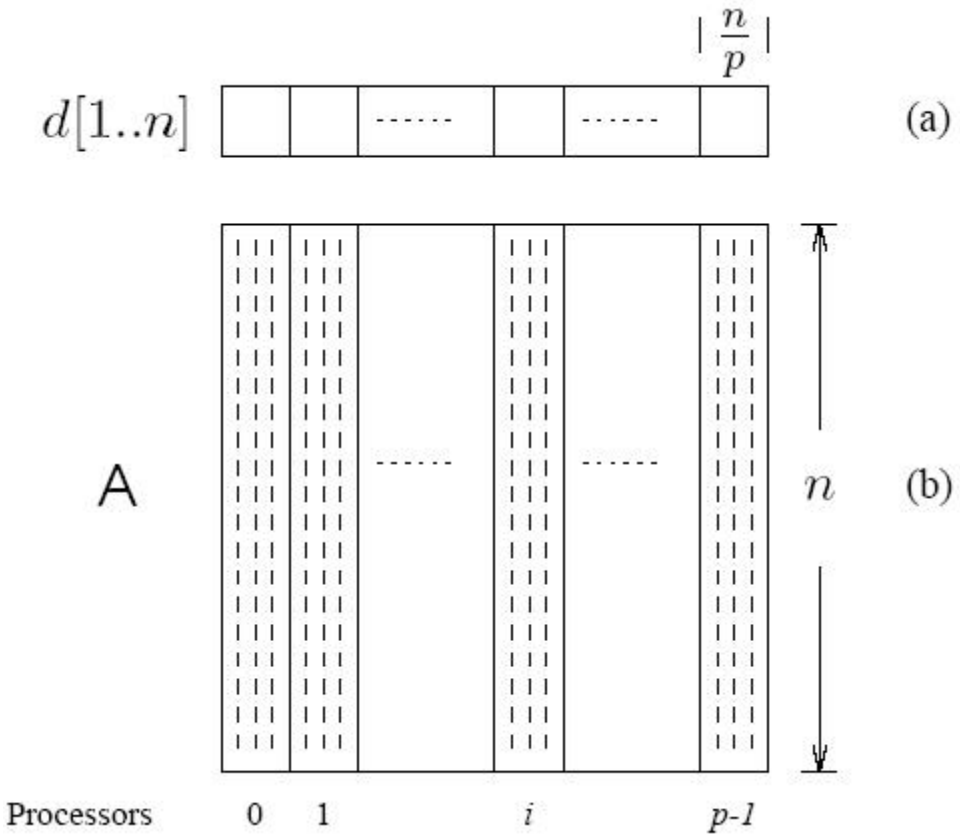
Single-Source Shortest Paths: Dijkstra's Algorithm

```
1.   procedure DIJKSTRA_SINGLE_SOURCE_SP(V, E, w, s)
2.   begin
3.       V_T := {s};
4.       for all v ∈ (V − V_T) do
5.           if (s, v) exists set l[v] := w(s, v);
6.           else set l[v] := ∞;
7.       while V_T ≠ V do
8.       begin
9.           find a vertex u such that l[u] := min{l[v]|v ∈ (V − V_T)};
10.          V_T := V_T ∪ {u};
11.          for all v ∈ (V − V_T) do
12.              l[v] := min{l[v], l[u] + w(u, v)};
13.      endwhile
14.  end DIJKSTRA_SINGLE_SOURCE_SP
```

Dijkstra's sequential single-source shortest paths algorithm.

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Dijkstra's Algorithm: Parallel Formulation



The partitioning of the distance array d and the adjacency matrix A among p processes.

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Dijkstra's Algorithm: Parallel Formulation

- **Parallel formulation**
  - The weighted adjacency matrix is partitioned using the 1-D block mapping.
  - Each process selects, locally, the node closest to the source, followed by a global reduction to select next node.
  - The node is broadcast to all processors and the *l*-vector updated.
- **Parallel performance**
  - The parallel time per iteration is $O(n/p + \log p)$.
    - The cost to select the minimum entry is $O(n/p + \log p)$.
    - The cost of a broadcast is $O(\log p)$.
    - The cost of local update of the *d* vector is $O(n/p)$.
  - The total parallel time is given by $O(n^2/p + n \log p)$.
  - ATTENSION
    - $O(n/p)$ for minimum selection and local update is only estimated for the algorithm we use here, which can be improved by a better algorithm and advance data structures (e.g., heap)

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

All-Pairs Shortest Paths

- **Given a weighted graph $G(V,E,w)$, the *all-pairs shortest paths* problem is to find the shortest paths between all pairs of vertices $v_i, v_j \in V$.**
- **A number of algorithms are known for solving this problem.**

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Dijkstra's Algorithm

- **Execute *n* instances of the single-source shortest path problem, one for each of the *n* source vertices.**
- **Complexity is $O(n^3)$.**
  - (without using priority queues and advanced data structures)

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Dijkstra's Algorithm: Parallel Formulation

- **Two parallelization strategies**
  - (source partitioned) execute each of the *n* shortest path problems on a different processor, or
  - (source parallel) use a parallel formulation of the shortest path problem to increase concurrency

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Dijkstra's Algorithm: Source Partitioned Formulation

- Use n processes, each process P_i finds the shortest paths from vertex v_i to all other vertices by executing Dijkstra's sequential single-source shortest paths algorithm.
- It requires no interprocess communication (provided that the adjacency matrix is replicated at all processes).
- The parallel run time of this formulation is: $\Theta(n^2)$.

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Dijkstra's Algorithm: Source Parallel Formulation

- **In this case, each of the shortest path problems is further executed in parallel**
- **Given p processes ($p > n$), each single source shortest path problem is executed by p/n processes.**
- **Using previous results, this takes total time:**

$$T_P = \overbrace{\Theta\left(\frac{n^3}{p}\right)}^{\text{computation}} + \overbrace{\Theta(n \log p).}^{\text{communication}}$$

  - ($log\ p$ is a pessimistic estimate of the broadcasting time)

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Floyd's Algorithm

- **For any pair of vertices v_i, $v_j \in V$, consider all paths from v_i to v_j whose intermediate vertices belong to the set $\{v_1, v_2, \ldots, v_k\}$.**
  - Let $p_{i,j}^{(k)}$ (of weight $d_{i,j}^{(k)}$) be the minimum-weight path among them.
- **If vertex v_k is not in the shortest path from v_i to v_j, then $p_{i,j}^{(k)}$ is the same as $p_{i,j}^{(k-1)}$.**
- **If v_k is in $p_{i,j}^{(k)}$, then we can break $p_{i,j}^{(k)}$ into two paths - one from v_i to v_k and one from v_k to v_j. Each of these paths uses vertices from $\{v_1, v_2, \ldots, v_{k-1}\}$.**

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Floyd's Algorithm

From our observations, the following recurrence relation follows:

$$d_{i,j}^{(k)} = \begin{cases} w(v_i, v_j) & \text{if } k = 0 \\ \min\left\{ d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)} \right\} & \text{if } k \geq 1 \end{cases}$$

This equation must be computed for each pair of nodes and for $k = 1, n$. The serial complexity is $O(n^3)$.

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Floyd's Algorithm

```
1.  procedure FLOYD_ALL_PAIRS_SP(A)
2.  begin
3.      D^(0) = A;
4.      for k := 1 to n do
5.          for i := 1 to n do
6.              for j := 1 to n do
7.                  d_{i,j}^(k) := min ( d_{i,j}^(k-1), d_{i,k}^(k-1) + d_{k,j}^(k-1) );
8.  end FLOYD_ALL_PAIRS_SP
```

Floyd's all-pairs shortest paths algorithm. This program computes the all-pairs shortest paths of the graph $G = (V,E)$ with adjacency matrix A.

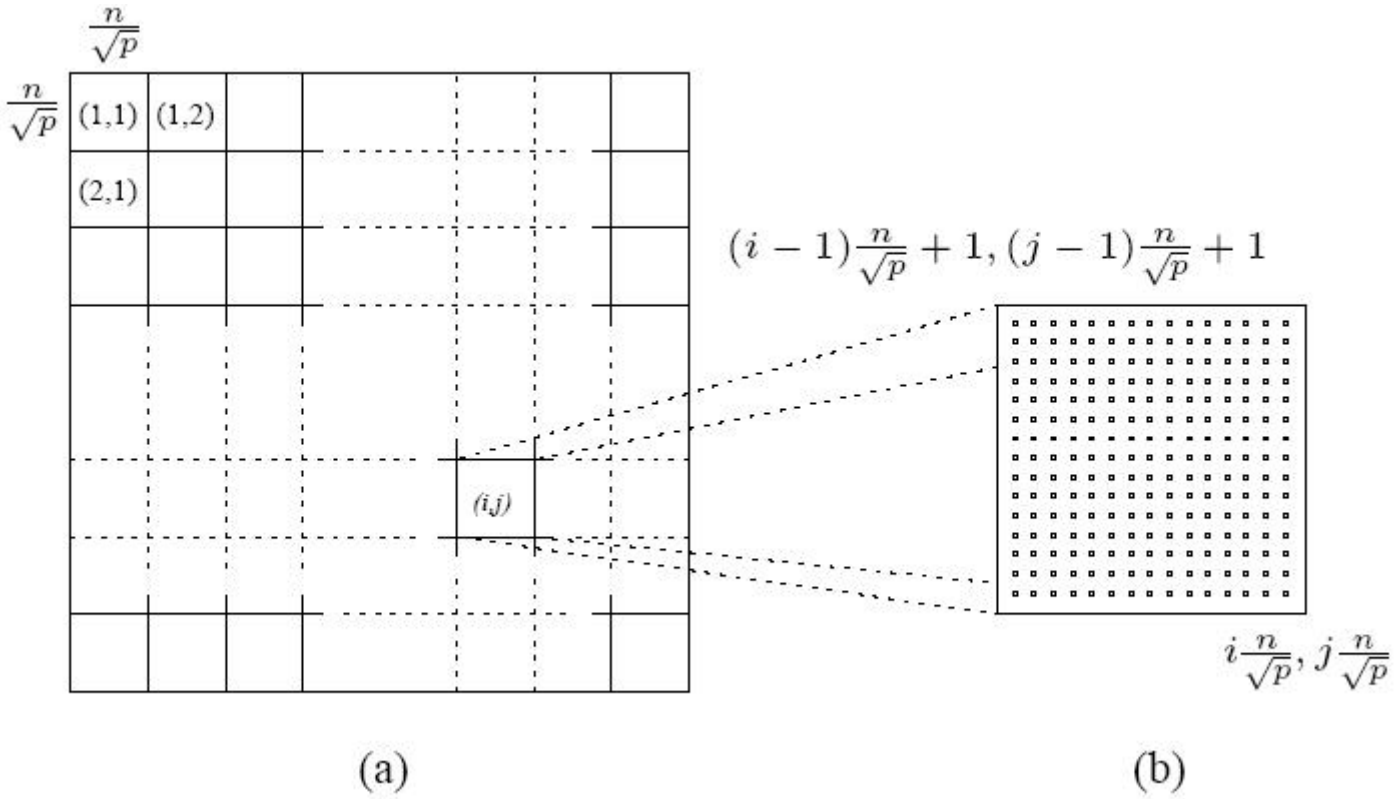
A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Floyd's Algorithm: Parallel Formulation Using 2-D Block Mapping

- **Matrix $D^{(k)}$ is divided into p blocks of size $(n/\sqrt{p}) \times (n/\sqrt{p})$.**
- **Each processor updates its part of the matrix during each iteration.**
- **To compute $d_{i,j}^{(k)}$ processor $P_{i,j}$ must get $d_{i,k}^{(k-1)}$ and $d_{k,j}^{(k-1)}$.**
- **In general, during the k^{th} iteration**
  - each of the $\sqrt{p}$ processes containing part of the k^{th} row send it to the $\sqrt{p}-1$ processes in the same column.
  - Similarly, each of the $\sqrt{p}$ processes containing part of the k^{th} column sends it to the $\sqrt{p}-1$ processes in the same row.

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Floyd's Algorithm: Parallel Formulation Using 2-D Block Mapping

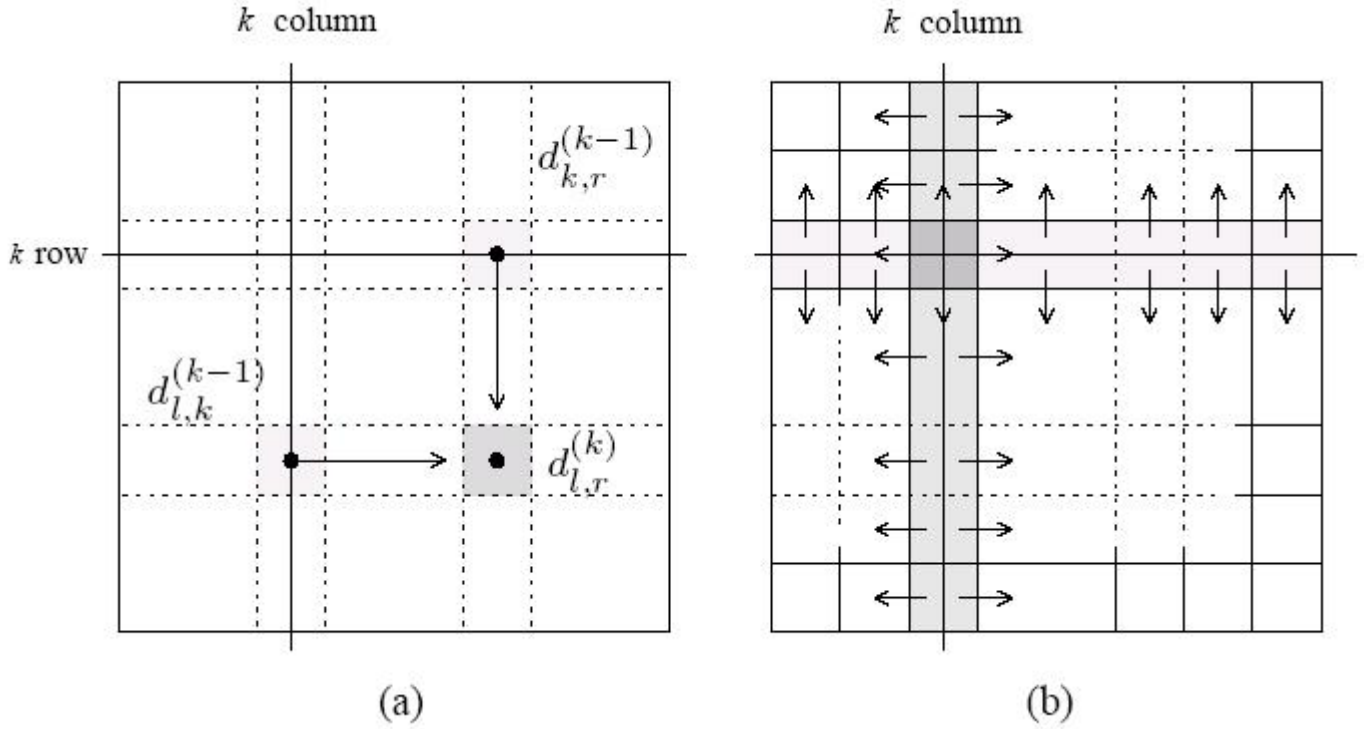


(a) Matrix $D^{(k)}$ distributed by 2-D block mapping into $\sqrt{p} \times \sqrt{p}$ subblocks,

(b) the subblock of $D^{(k)}$ assigned to process $P_{i,j}$.

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Floyd's Algorithm: Parallel Formulation Using 2-D Block Mapping



(a) Communication patterns used in the 2-D block mapping. When computing $d_{i,j}^{(k)}$, information must be sent to the highlighted process from two other processes along the same row and column.

(b) The row and column of $\sqrt{p}$ processes that contain the k^{th} row and column send them along process columns and rows.

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Floyd's Algorithm: Parallel Formulation Using 2-D Block Mapping

```
1.   procedure FLOYD_2DBLOCK(D^(0))
2.   begin
3.       for k := 1 to n do
4.       begin
5.           each process P_{i,j} that has a segment of the k^th row of D^(k-1);
                 broadcasts it to the P_{*,j} processes;
6.           each process P_{i,j} that has a segment of the k^th column of D^(k-1);
                 broadcasts it to the P_{i,*} processes;
7.           each process waits to receive the needed segments;
8.           each process P_{i,j} computes its part of the D^(k) matrix;
9.       end
10.  end FLOYD_2DBLOCK
```

Floyd's parallel formulation using the 2-D block mapping. $P_{*,j}$ denotes all the processes in the j^{th} column, and $P_{i,*}$ denotes all the processes in the i^{th} row. The matrix $D^{(0)}$ is the adjacency matrix.

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Floyd's Algorithm: Parallel Formulation Using 2-D Block Mapping

- **During each iteration of the algorithm, the k^{th} row and k^{th} column of processors perform a one-to-all broadcast along their rows/columns.**
- **The size of this broadcast is $n/\sqrt{p}$ elements, taking time $\Theta((n \log p)/\sqrt{p})$.**
- **The synchronization step takes time $\Theta(\log p)$.**
- **The computation time is $\Theta(n^2/p)$.**
- **The parallel run time of the 2-D block mapping formulation of Floyd's algorithm is**

$$T_P = \overbrace{\Theta\left(\frac{n^3}{p}\right)}^{\text{computation}} + \overbrace{\Theta\left(\frac{n^2}{\sqrt{p}}\log p\right)}^{\text{communication}}.$$

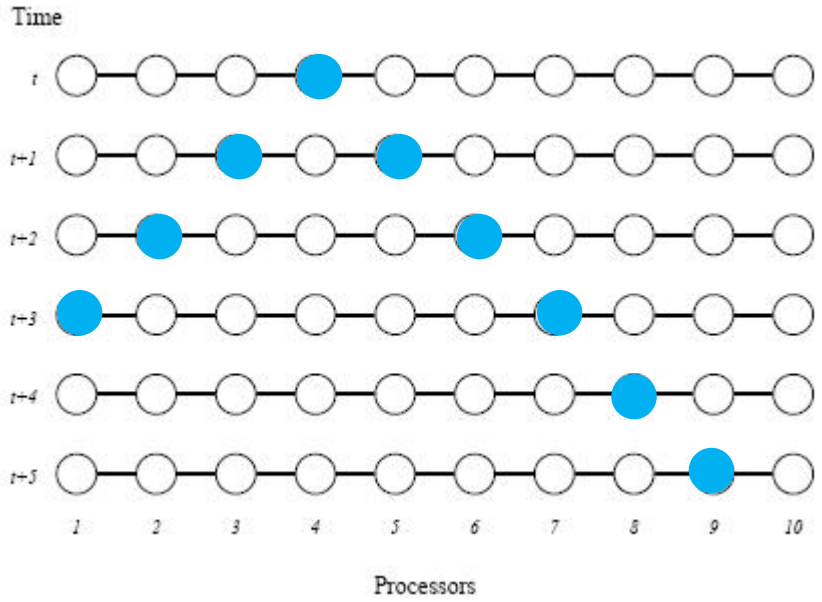
A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Floyd's Algorithm: Speeding Things Up by Pipelining

- **The synchronization step in parallel Floyd's algorithm can be removed without affecting the correctness of the algorithm.**
- **A process starts working on the k^{th} iteration as soon as it has computed the $(k-1)^{th}$ iteration and has the relevant parts of the $D^{(k-1)}$ matrix.**

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Floyd's Algorithm: Speeding Things Up by Pipelining



Communication protocol followed in the pipelined 2-D block mapping formulation of Floyd's algorithm. Assume that process 4 at time *t* has just computed a segment of the k^{th} column of the $D^{(k-1)}$ matrix. It sends the segment to processes 3 and 5. These processes receive the segment at time *t+1* (where the time unit is the time it takes for a matrix segment to travel over the communication link between adjacent processes). Similarly, processes farther away from process 4 receive the segment later. Process 1 (at the boundary) does not forward the segment after receiving it.

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Floyd's Algorithm: Speeding Things Up by Pipelining

- In each step, $n/\sqrt{p}$ elements of the first row are sent from process $P_{i,j}$ to $P_{i+1,j}$.
- Similarly, elements of the first column are sent from process $P_{i,j}$ to process $P_{i,j+1}$.
- Each such step takes time $\Theta(n/\sqrt{p})$.
- After $\Theta(\sqrt{p})$ steps, process $P_{\sqrt{p},\sqrt{p}}$ gets the relevant elements of the first row and first column in time $\Theta(n)$.
- The values of successive rows and columns follow after time $\Theta(n^2/p)$ in a pipelined mode.
- Process $P_{\sqrt{p},\sqrt{p}}$ finishes its share of the shortest path computation in time $\Theta(n^3/p) + \Theta(n)$.
- When process $P_{\sqrt{p},\sqrt{p}}$ has finished the $(n-1)^{th}$ iteration, it sends the relevant values of the n^{th} row and column to the other processes.

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Floyd's Algorithm: Speeding Things Up by Pipelining

- **The overall parallel run time of this formulation is**

$$T_P = \overbrace{\Theta\left(\frac{n^3}{p}\right)}^{\text{computation}} + \overbrace{\Theta(n).}^{\text{communication}}$$

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

All-pairs Shortest Path: Comparison

- **The performance and scalability of the all-pairs shortest paths algorithms on various architectures with bisection bandwidth. Similar run times apply to all cube architectures, provided that processes are properly mapped to the underlying processors.**

	Maximum Number of Processes for $E = \Theta(1)$	Corresponding Parallel Run Time
Dijkstra source-partitioned	$\Theta(n)$	$\Theta(n^2)$
Dijkstra source-parallel	$\Theta(n^2/\log n)$	$\Theta(n \log n)$
Floyd 1-D block	$\Theta(n/\log n)$	$\Theta(n^2 \log n)$
Floyd 2-D block	$\Theta(n^2/\log^2 n)$	$\Theta(n \log^2 n)$
Floyd pipelined 2-D block	$\Theta(n^2)$	$\Theta(n)$

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Single-Source Shortest Path, again

- **Famous serial algorithms**
  - Dijkstra : label setting – requires nonnegative edge weights
  - Bellman-Ford : label correcting - works on any graph
- **Example of share-memory parallel SSSP**
  - Parallel Bellman-Ford
  - Delta-stepping

SSSP: Serial Bellman-Ford

```
void BellmanFord(Graph g, Vertex s) {
  queue<Vertex> qi, qo;
  for each Vertex v { v.dist = INF; }
  s.dist = 0;  qi.enqueue(s);
  while (!qi.empty()) {
    Vertex v = qi.dequeue();
    for each edge (v,w) do
      if (v.dist + c(v,w) < w.dist) {
        w.dist = v.dist + c(v,w);
        w.prev = v;
        if (w is not in qo) qo.enqueue(w);
      }
    swap(qi, qo);
}}
```

SSSP: Parallel Bellman-Ford v1

```
void BellmanFord(Graph g, Vertex s) {
  queue<Vertex> qi, qo; // concurrent queues
  for each Vertex v { v.dist = INF; }
  s.dist = 0;  qi.enqueue(s);
  while (!qi.empty()) {
    for each v=qi.dequeue() in parallel do
      for each edge (v,w) do {
        lock(w);
        if (v.dist + c(v,w) < w.dist) {
          w.dist = v.dist + c(v,w);  w.prev = v;  qo.enqueue(w);
        }
        unlock(w);
      }
    swap(qi, qo);
}}
```

SSSP: Parallel Bellman-Ford v2

```
void BellmanFord(Graph g, Vertex s) {
  queue<Vertex> qi, qo; // concurrent queues
  for each Vertex v { v.dist = INF; }
  s.dist = 0; qi.enqueue(s);
  while (!qi.empty()) {
    for each v=qi.dequeue() in parallel do // generate requests
      requests.insert((v, w, d=v.dist+c(v,w), act=(d<w.dist)));
    for each (v,w,d,act) in requests in parallel do { // execute requests
      if (act) {
          lock(w);
          if (act) { w.dist = d;  w.prev = v;  qo.enqueue(w); }
          unlock(w);
        }
      }
    swap(qi, qo);
}}
```

ideas similar to D. Merrill, et al., “Scalable GPU graph traversal,” PPoPP ’12.

Single-Source Shortest Path, again

- **Famous serial algorithms**
  - Dijkstra : label setting – requires nonnegative edge weights
  - Bellman-Ford : label correcting - works on any graph
- **Meyer and Sanders, Δ - stepping algorithm**

  U. Meyer and P. Sanders, Δ - stepping: a parallelizable shortest path algorithm. Journal of Algorithms 49 (2003)
- **Chakaravarthy et al., clever combination of Δ - stepping and direction optimization (BFS) on supercomputer-scale graphs**

  V. T. Chakaravarthy, F. Checconi, F. Petrini, Y. Sabharwal
  "Scalable Single Source Shortest Path Algorithms for Massively Parallel Systems ", IPDPS'14

Δ - stepping algorithm

- **The rough idea**
  - Set up the value of Δ
    - Classify edges as heavy (weight≥Δ) and light (weight<Δ)
    - Conceptually, divide all vertices into buckets
      - B[0] = { v : d(s,v) ∈ [0, Δ) }
      - B[1] = { v : d(s,v) ∈ [Δ, 2Δ) }
      - B[2] = { v : d(s,v) ∈ [2Δ, 3Δ) }
      - …
      - B[i] = { v : d(s,v) ∈ [iΔ, (i+1)Δ) }
  - phase i of Δ-stepping algorithm: relax vertices in B[i]

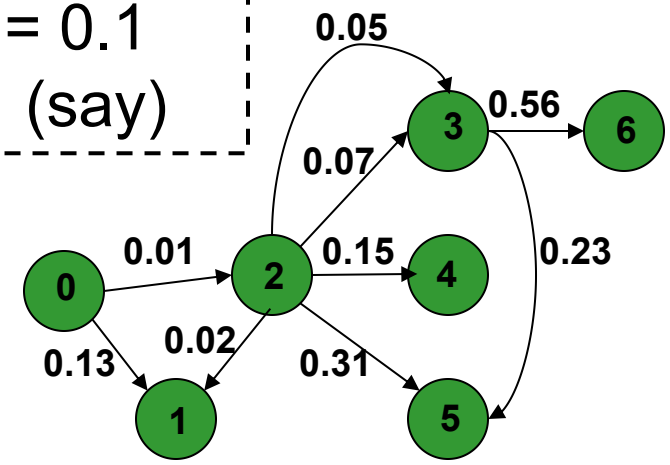
Δ - stepping algorithm

- **Label-correcting algorithm: Can relax edges from unsettled vertices**
  - also "approximate bucket implementation of Dijkstra"
- **For random edge weighs [0,1], runs in $O(n+m+D \cdot L)$**
  - where L = max distance from source to any node, and D = 1/Δ
- **Vertices are ordered using buckets of width Δ**
- **Each bucket may be processed in parallel**
- **Basic operation: Relax (e(u,v))**
  - d(v) = min { d(v), d(u) + w(u, v) }
- **Δ < min w(e) : Degenerates into Dijkstra**
- **Δ > max w(e) : Degenerates into Bellman-Ford**

Buluc, Demmel, Yelick, "Applications of Parallel Computers," Berkeley CS267 Spring 2018.

Δ - stepping algorithm: illustration

Δ = 0.1 (say)



d array

0	1	2	3	4	5	6
0	∞	∞	∞	∞	∞	∞

Buckets

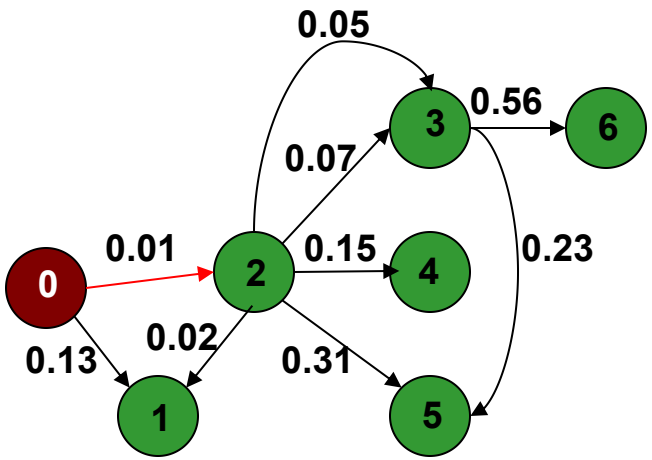
0	0

One parallel **phase**
while (bucket is non-empty)
- i) Inspect light (w < Δ) edges
- ii) Construct a set of "requests" (R)
- iii) Clear the current bucket
- iv) Remember deleted vertices (S)
- v) Relax request pairs in R

Relax heavy request pairs (from S)
Go on to the next bucket

Initialization:
Insert *s* into bucket, d(s) = 0

Δ - stepping algorithm: illustration



d array

0	1	2	3	4	5	6
0	∞	∞	∞	∞	∞	∞

Buckets

0	0						

One parallel **phase**
while (bucket is non-empty)
- i) Inspect light (w < Δ) edges
- ii) Construct a set of "requests" (R)
- iii) Clear the current bucket
- iv) Remember deleted vertices (S)
- v) Relax request pairs in R

Relax heavy request pairs (from S)
Go on to the next bucket

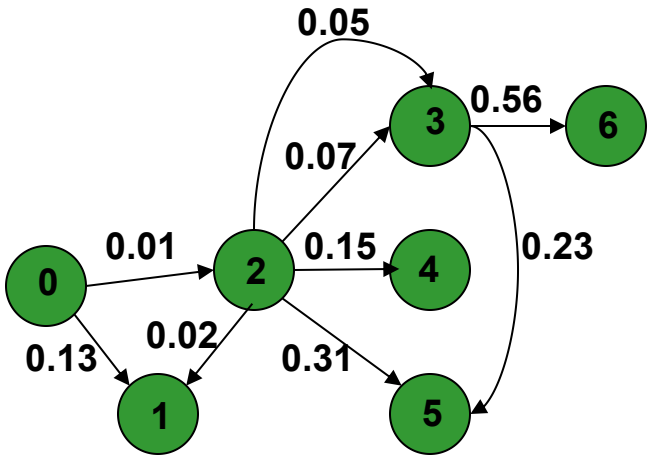
R

2						
.01						

S

Buluc, Demmel, Yelick, "Applications of Parallel Computers," Berkeley CS267 Spring 2018.

Δ - stepping algorithm: illustration



d array

0	1	2	3	4	5	6
0	∞	∞	∞	∞	∞	∞

Buckets

0							

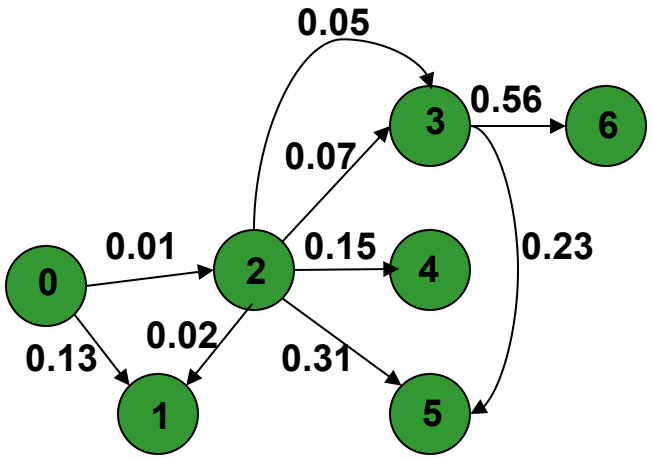
One parallel **phase**
while (bucket is non-empty)
- i) Inspect light (w < Δ) edges
- ii) Construct a set of "requests" (R)
- iii) Clear the current bucket
- iv) Remember deleted vertices (S)
- v) Relax request pairs in R

Relax heavy request pairs (from S)
Go on to the next bucket

R	2						
	.01						
S	0						

Buluc, Demmel, Yelick, "Applications of Parallel Computers," Berkeley CS267 Spring 2018.

Δ - stepping algorithm: illustration



d array

0	1	2	3	4	5	6
0	∞	.01	∞	∞	∞	∞

Buckets

0	2						

One parallel **phase**
while (bucket is non-empty)
- i) Inspect light (w < Δ) edges
- ii) Construct a set of "requests" (R)
- iii) Clear the current bucket
- iv) Remember deleted vertices (S)
- v) Relax request pairs in R

Relax heavy request pairs (from S)
Go on to the next bucket

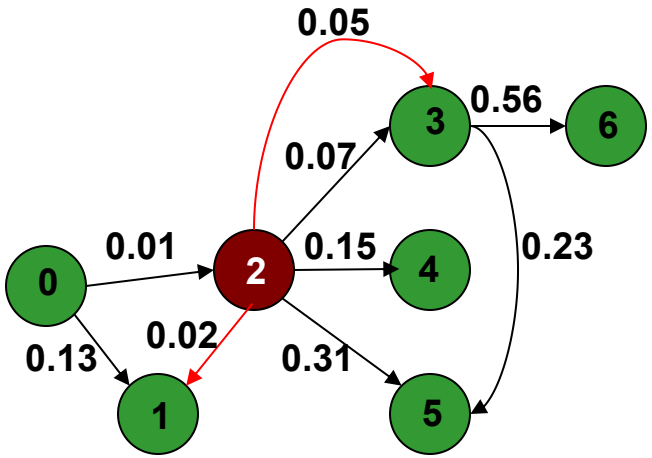
R

S

0						

Buluc, Demmel, Yelick, "Applications of Parallel Computers," Berkeley CS267 Spring 2018.

Δ - stepping algorithm: illustration



d array

0	1	2	3	4	5	6
0	∞	.01	∞	∞	∞	∞

Buckets

0	2						

One parallel **phase**
while (bucket is non-empty)
- i) Inspect light (w < Δ) edges
- ii) Construct a set of "requests" (R)
- iii) Clear the current bucket
- iv) Remember deleted vertices (S)
- v) Relax request pairs in R

Relax heavy request pairs (from S)
Go on to the next bucket

R

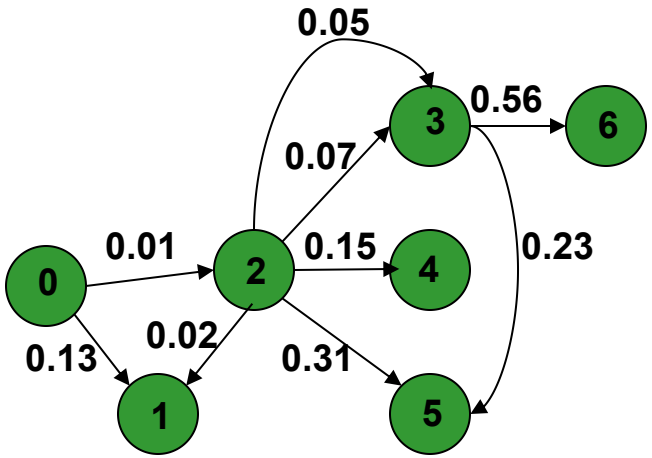
1	3					
.03	.06					

S

0						

Buluc, Demmel, Yelick, "Applications of Parallel Computers," Berkeley CS267 Spring 2018.

Δ - stepping algorithm: illustration



d array

0	1	2	3	4	5	6
0	∞	.01	∞	∞	∞	∞

Buckets

0							

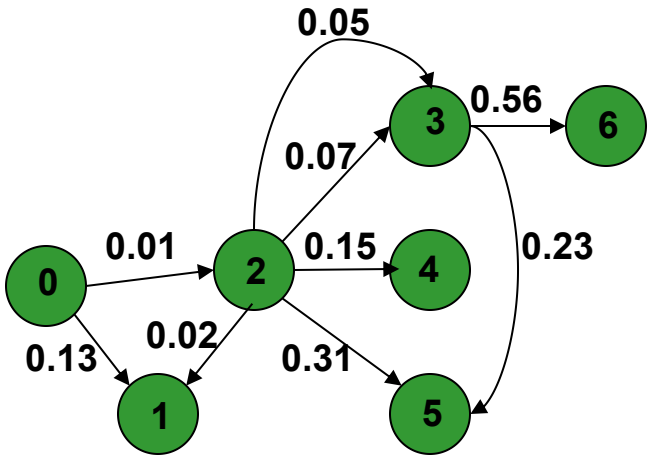
One parallel **phase**
while (bucket is non-empty)
- i) Inspect light (w < Δ) edges
- ii) Construct a set of "requests" (R)
- iii) Clear the current bucket
- iv) Remember deleted vertices (S)
- v) Relax request pairs in R

Relax heavy request pairs (from S)
Go on to the next bucket

R	1	3					
	.03	.06					
S	0	2					

Buluc, Demmel, Yelick, "Applications of Parallel Computers," Berkeley CS267 Spring 2018.

Δ - stepping algorithm: illustration



d array

0	1	2	3	4	5	6
0	.03	.01	.06	∞	∞	∞

Buckets

0	1	3					

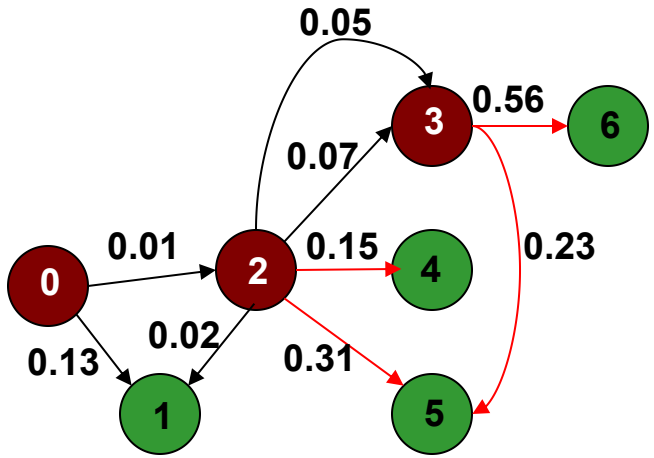
One parallel **phase**
while (bucket is non-empty)
- i) Inspect light (w < Δ) edges
- ii) Construct a set of "requests" (R)
- iii) Clear the current bucket
- iv) Remember deleted vertices (S)
- v) Relax request pairs in R

Relax heavy request pairs (from S)
Go on to the next bucket

R							
S	0	2					

Buluc, Demmel, Yelick, "Applications of Parallel Computers," Berkeley CS267 Spring 2018.

Δ - stepping algorithm: illustration



d array

0	1	2	3	4	5	6
0	.03	.01	.06	.16	.29	.62

Buckets

1	4						
2	5						
6	6						

One parallel **phase**
while (bucket is non-empty)
- i) Inspect light ($w < \Delta$) edges
- ii) Construct a set of "requests" (R)
- iii) Clear the current bucket
- iv) Remember deleted vertices (S)
- v) Relax request pairs in R

Relax heavy request pairs (from S)
Go on to the next bucket

R							
S	0	2	1	3			

Algorithm 1: Δ-stepping algorithm

Input: $G(V, E)$, source vertex s, length function $l : E \to \mathbb{R}$

Output: $\delta(v), v \in V$, the weight of the shortest path from s to v

```
1  foreach v ∈ V do
2      heavy(v) ⟵ {⟨v, w⟩ ∈ E : l(v, w) > Δ};
3      light(v) ⟵ {⟨v, w⟩ ∈ E : l(v, w) ≤ Δ};
4      d(v) ⟵ ∞;
5  relax(s, 0);
6  i ⟵ 0;
7  while B is not empty do
8      S ⟵ φ;
9      while B[i] ≠ φ do
10         Req ⟵ {(w, d(v) + l(v, w)) : v ∈ B[i] ∧ ⟨v, w⟩ ∈ light(v)};
11         S ⟵ S ∪ B[i];
12         B[i] ⟵ φ;
13         foreach (v, x) ∈ Req do
14             relax(v, x);
15     Req ⟵ {(w, d(v) + l(v, w)) : v ∈ S ∧ ⟨v, w⟩ ∈ heavy(v)};
16     foreach (v, x) ∈ Req do
17         relax(v, x);
18     i ⟵ i + 1;
19 foreach v ∈ V do
20     δ(v) ⟵ d(v);
```

Algorithm 1: Δ-stepping algorithm

Input: $G(V, E)$, source vertex s, length function $l : E \to \mathbb{R}$

Output: $\delta(v), v \in V$, the weight of the shortest path from s to v

```
1  foreach v ∈ V do
2      heavy(v) ⟵ {⟨v, w⟩ ∈ E : l(v, w) > Δ};
3      light(v) ⟵ {⟨v, w⟩ ∈ E : l(v, w) ≤ Δ};
4      d(v) ⟵ ∞;
5  relax(s, 0);
6  i ⟵ 0;
7  while B is not empty do
8      S ⟵ φ;
9      while B[i] ≠ φ do
10         Req ⟵ {(w, d(v) + l(v, w)) : v ∈ B[i] ∧ ⟨v, w⟩ ∈ light(v)};
11         S ⟵ S ∪ B[i];
12         B[i] ⟵ φ;
13         foreach (v, x) ∈ Req do
14             relax(v, x);
15     Req ⟵ {(w, d(v) + l(v, w)) : v ∈ S ∧ ⟨v, w⟩ ∈ heavy(v)};
16     foreach (v, x) ∈ Req do
17         relax(v, x);
18     i ⟵ i + 1;
19 foreach v ∈ V do
20     δ(v) ⟵ d(v);
```

Classify edges as "heavy" and "light"

Algorithm 1: Δ-stepping algorithm

Input: $G(V, E)$, source vertex s, length function $l : E \to \mathbb{R}$

Output: $\delta(v), v \in V$, the weight of the shortest path from s to v

```
1  foreach v ∈ V do
2      heavy(v) ⟵ {⟨v, w⟩ ∈ E : l(v, w) > Δ};
3      light(v) ⟵ {⟨v, w⟩ ∈ E : l(v, w) ≤ Δ};
4      d(v) ⟵ ∞;
5  relax(s, 0);
6  i ⟵ 0;
7  while B is not empty do
8      S ⟵ φ;
9      while B[i] ≠ φ do
10         Req ⟵ {(w, d(v) + l(v, w)) : v ∈ B[i] ∧ ⟨v, w⟩ ∈ light(v)};
11         S ⟵ S ∪ B[i];
12         B[i] ⟵ φ;
13         foreach (v, x) ∈ Req do
14             relax(v, x);
15     Req ⟵ {(w, d(v) + l(v, w)) : v ∈ S ∧ ⟨v, w⟩ ∈ heavy(v)};
16     foreach (v, x) ∈ Req do
17         relax(v, x);
18     i ⟵ i + 1;
19 foreach v ∈ V do
20     δ(v) ⟵ d(v);
```

Relax light edges (phase)
Repeat until B[i] Is empty

Algorithm 1: Δ-stepping algorithm

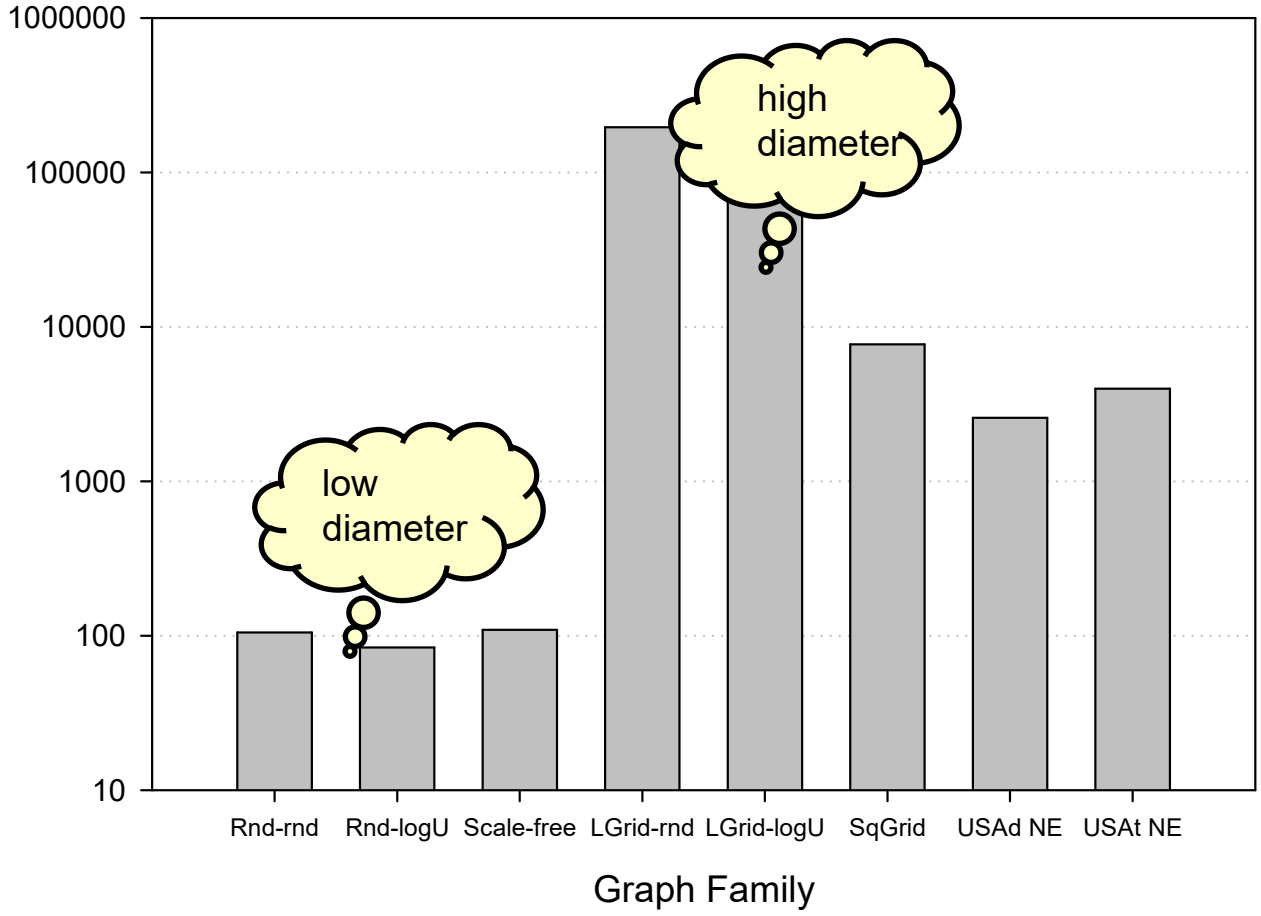
Input: $G(V, E)$, source vertex s, length function $l : E \to \mathbb{R}$

Output: $\delta(v), v \in V$, the weight of the shortest path from s to v

```
1  foreach v ∈ V do
2      heavy(v) ⟵ {⟨v, w⟩ ∈ E : l(v, w) > Δ};
3      light(v) ⟵ {⟨v, w⟩ ∈ E : l(v, w) ≤ Δ};
4      d(v) ⟵ ∞;
5  relax(s, 0);
6  i ⟵ 0;
7  while B is not empty do
8      S ⟵ φ;
9      while B[i] ≠ φ do
10         Req ⟵ {(w, d(v) + l(v, w)) : v ∈ B[i] ∧ ⟨v, w⟩ ∈ light(v)};
11         S ⟵ S ∪ B[i];
12         B[i] ⟵ φ;
13         foreach (v, x) ∈ Req do
14             relax(v, x);
15     Req ⟵ {(w, d(v) + l(v, w)) : v ∈ S ∧ ⟨v, w⟩ ∈ heavy(v)};
16     foreach (v, x) ∈ Req do
17         relax(v, x);
18     i ⟵ i + 1;
19 foreach v ∈ V do
20     δ(v) ⟵ d(v);
```

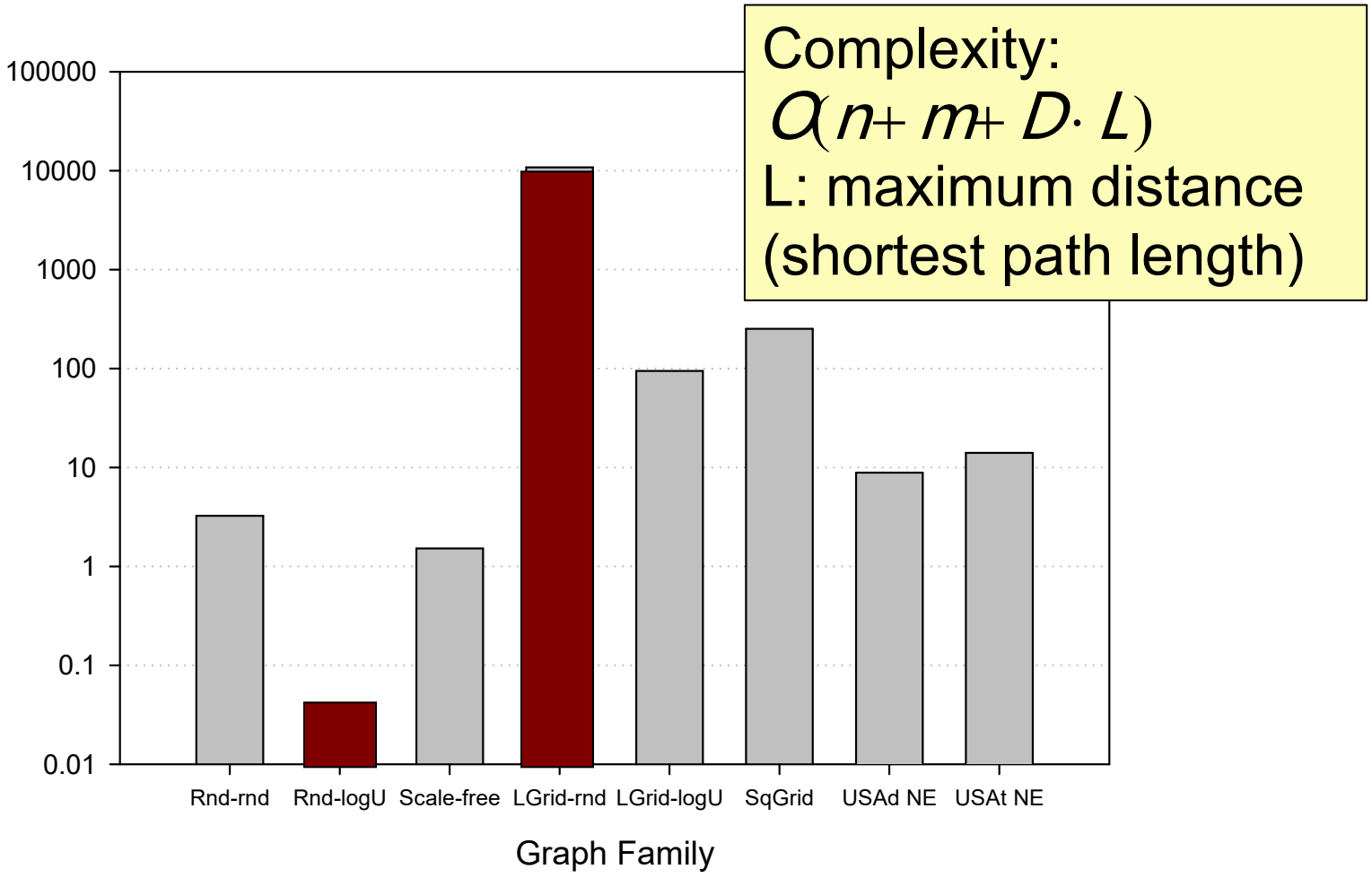
Relax heavy edges. No reinsertions in this step.

No. of phases (machine-independent performance count)



Too many phases in high diameter graphs:
Level-synchronous breadth-first search has the same problem.

Average shortest path weight for various graph families
~ 2^{20} vertices, 2^{22} edges, directed graph, edge weights normalized to [0,1]



Buluc, Demmel, Yelick, "Applications of Parallel Computers," Berkeley CS267 Spring 2018.

Summary

- **Selected parallel graph algorithms**
  - Graph and sparse matrix
  - Independent set
  - Strongly-connected components
  - Shortest path and Δ-stepping