THE UNIVERSITY OF WARWICK

Modern Parallel Algorithms

Lecture 3

Artur Czumaj

DIMAP and Department of Computer Science

University of Warwick

Warszawa, November 2022

Sorting

- We discussed sorting for small values of N (with respect to s)
- One can do "similarly" for arbitrary N and s: to sort in $O(\log_s N)$ rounds

- This can be done even deterministically!

Corollary: Sorting of N numbers on an MPC with local space $s = N^{\delta}$ and linear total space $O(N)$ can be done deterministically in $O(1)$ rounds

PRAM: sorting lower bound of $\Omega(\log N)$ time (with any $\mathrm{poly}(N)$ number of processors)

Example: Sorting

A O(1)-rounds deterministic sorting for $s = O(n^{\delta})$

Input: distinct numbers $x_1, \ldots, x_N$; machine M_i stores $x_{(i-1)\cdot \mathfrak{s}+1}, \ldots, x_{i\cdot \mathfrak{s}}$, $1 \le i \le \mathfrak{m} = \frac{N}{\mathfrak{s}}$

Output: machine M_i stores $x_{\pi((i-1)\cdot \mathfrak{s}+1)}, \ldots, x_{\pi(i\cdot \mathfrak{s})}$, $1 \le i \le \frac{N}{\mathfrak{s}}$, where π is an N-permutation such that $x_{\pi(1)} < x_{\pi(2)} < \cdots \le x_{\pi(N)}$

$\mathfrak{m}$ – number of machines used
$\mathfrak{m} = O(N/\mathfrak{s})$

1 Partition the input (arbitrarily) into $\mathfrak{s}^{1/3}$ groups of the same size
2 ▶ each group has $\frac{N}{\mathfrak{s}^{1/3}}$ elements and has allocated $\frac{\mathfrak{m}}{\mathfrak{s}^{1/3}}$ machines
3 Recursively sort each group
4 In each group take as *pivots* all elements with ranks $j \cdot \mathfrak{s}^{2/3}$ for $1 \le j \le \frac{N}{\mathfrak{s}}$
5 // the set of $\frac{N}{\mathfrak{s}^{2/3}}$ pivots from all groups form a $\mathfrak{s}$-pivot
6 Each group sends all its pivots to the first group
7 First group recursively sort all $\frac{N}{\mathfrak{s}^{2/3}}$ pivots; denote the pivots[a] as $p_0 < p_1 < \cdots < p_{1+N/\mathfrak{s}^{2/3}}$.
8 Distribute the sorted order of all pivots to each group
9 ▶ machines in the first group send relative ranks of all pivots to every group
10 Partition all input elements in each group into sets $\Pi_j^{\langle i \rangle}$, with $1 \le i \le \mathfrak{s}^{1/3}$, $0 \le j \le \frac{N}{\mathfrak{s}^{2/3}}$, where $\Pi_j^{\langle i \rangle} := \{x \text{ input from group } i : p_j < x \le p_{j+1}\}$
11 Each group i sends the sizes of all sets $\Pi_j^{\langle i \rangle}$ to the $\frac{\mathfrak{m}}{\mathfrak{s}^{1/3}}$ machines in the first group
12 Machines in the first group compute the rank of each pivot $rank(p_j) = \sum_{k=0}^{j-1} \sum_{i=1}^{\mathfrak{s}^{1/3}} |\Pi_j^{\langle i \rangle}|$
13 Machines from the first group send ranks of all pivots to all groups
14 Machines in the first group allocate pivots to $2\mathfrak{s}^{1/3}$ groups with $\frac{\mathfrak{m}}{\mathfrak{s}^{1/3}}$ machines each, so that
15 ▶ the j-th smallest pivot p_j is allocated to group $\tau(j)$
16 ▶ $\tau(0) \le \tau(1) \le \cdots \le \tau(\frac{N}{\mathfrak{s}^{1/3}})$
17 ▶ if group k is allocated to pivots $j, \ldots, j'$ then $\sum_{r=j}^{j'} \sum_{i=1}^{\mathfrak{s}^{1/3}} |\Pi_r^{\langle i \rangle}| \le \frac{N}{\mathfrak{s}^{1/3}}$
18 Machines in the first group distribute all values $\tau(0), \ldots, \tau(\frac{N}{\mathfrak{s}^{2/3}})$ to all groups
19 Each group i, for every $0 \le j \le \frac{N}{\mathfrak{s}^{1/3}}$, sends $\Pi_j^{\langle i \rangle}$ to group $\tau(j)$
20 Each group k $(1 \le k \le 2\mathfrak{s}^{1/3})$ recursively sorts all elements in sets $\bigcup_{j:\tau(j)=k} \bigcup_{i=1}^{\mathfrak{s}^{1/3}} \Pi_j^{\langle i \rangle}$
21 Each group k determines the rank of its t-th largest element to be $t + \sum_{j:\tau(j)<k} \sum_{i=1}^{\mathfrak{s}^{1/3}} |\Pi_\ell^{\langle i \rangle}|$
22 Each machine, for each its element with rank ϱ, sends that element and its rank to $M_{\lfloor \varrho/\mathfrak{s} \rfloor}$

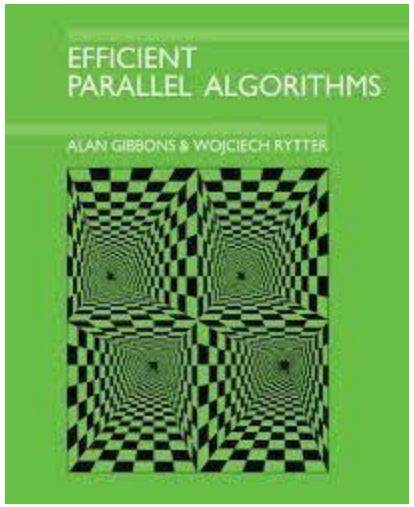
[a] We assume that the 0-th smallest pivot is $p_0 = -\infty$ and the $(\frac{N}{\mathfrak{s}^{2/3}}+1)$-st smallest pivot is $p_{1+N/\mathfrak{s}^{2/3}} = +\infty$.

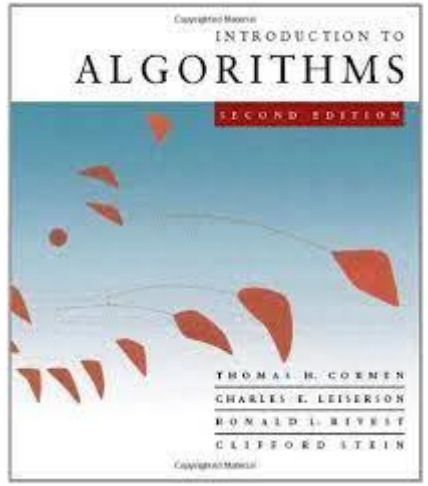
Number of rounds is constant for any positive constant δ
It's not $O(\log_s N)$ though

PRAM Simulations

- Why do we care?

- There are many PRAM algorithms + some textbooks (was even in CLRS)

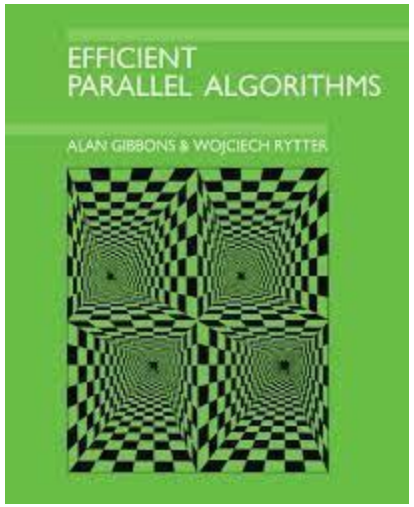


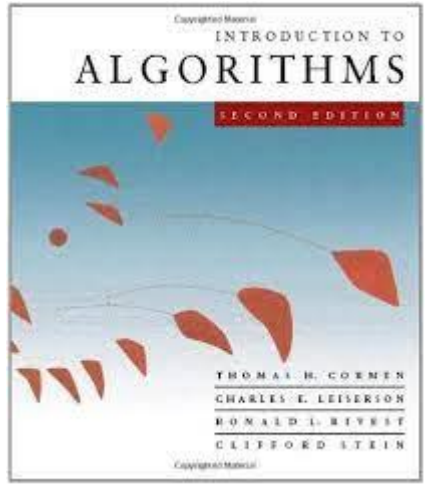


PRAM Simulations

Theorem: Let ALG be an CRCW PRAM algorithm that runs in T steps using P processors and Q memory cells.

Then one can simulate ALG in $O(T \log_s P)$ rounds on an MPC with local space s and $M = O(\frac{P+Q}{s})$ machines.

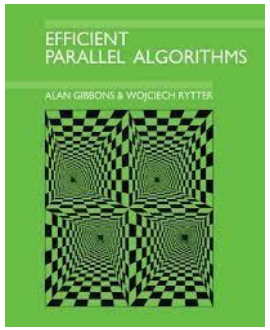


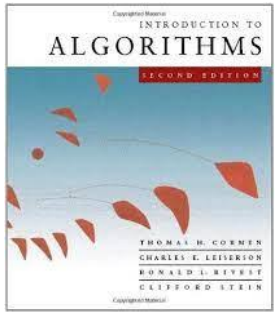


PRAM Simulations

- PRAM simulations yield many MPC algorithms
  - We have a great library of PRAM algorithms for graph problems, for computational geometry problems, etc
  - For example, linear programming (for constant-dimension) can be solved in $O(1)$ rounds on an MPC with local space $s = O(N^{\delta})$

- Main interest in the algorithmic community:
  - can we do better?





BSP Simulations

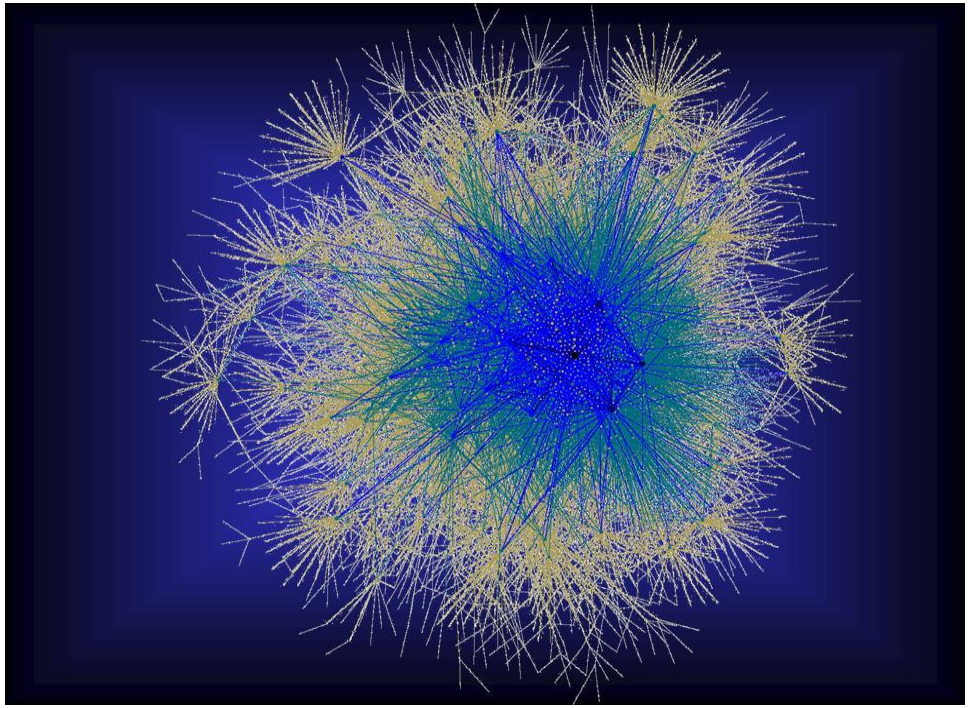
- Similar results

Congested clique

- Is almost equivalent to MPC with $s = O(n)$

MPC for graphs

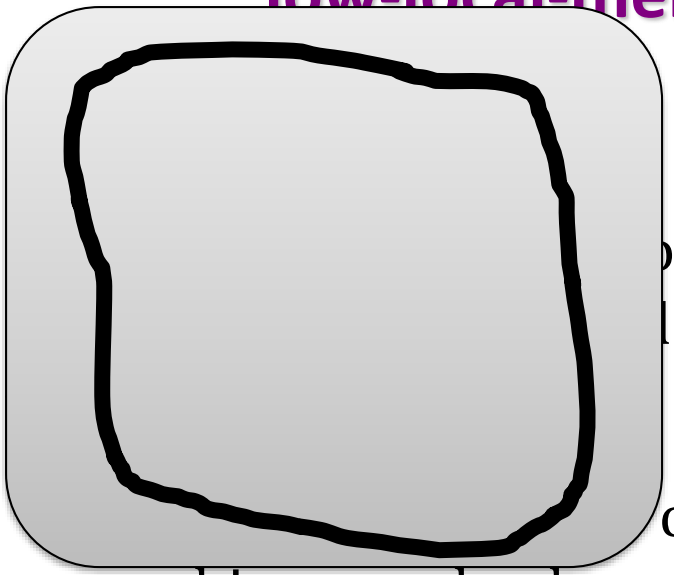
Input: Edges of an m-edge graph on n vertices

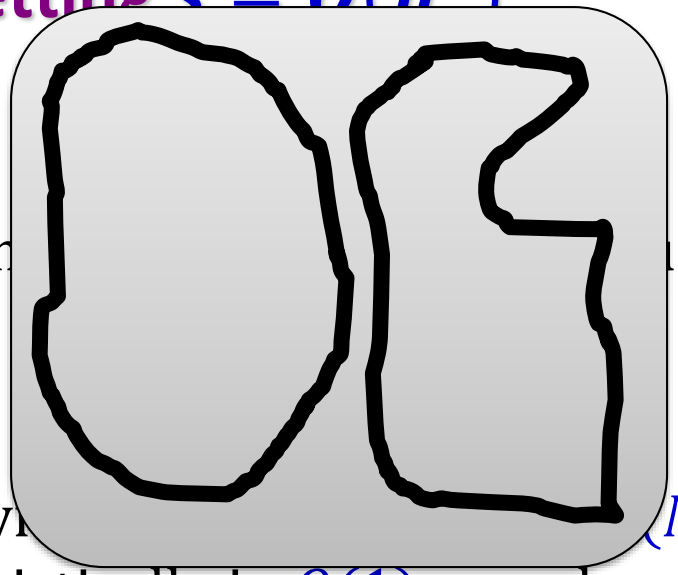


MPC algorithms for graphs
low-local-memory setting $s = O(n^{\delta})$

- PRAM simulation:
  - any t-steps PRAM algorithm can be simulated in $O(t)$ rounds on MPC with low-local space
- Basic primitive:
  - sorting of N numbers on MPC with $s = O(n^{\delta})$ on $M = O(N/s)$ machines can be done deterministically in $O(1)$ rounds
- Basic "obstacle":
  - 1-vs-2-cycles conjecture:
    - distinguishing between a cycle on n vertices and two cycles on $n/2$ vertices requires $\Omega(\log n)$ rounds on an MPC with $s = O(n^{\delta})$

MPC algorithms for graphs
low-local-memory setting $s = O(n^{\delta})$





- orithm can nds
  l space
- on MPC wi (N/s)
  machines can be done deterministically in $O(1)$ rounds
- Basic "obstacle":
  - 1-vs-2-cycles conjecture:
    - distinguishing between a cycle on n vertices and two cycles on $n/2$ vertices requires $\Omega(\log n)$ rounds on an MPC with $s = O(n^{\delta})$

MPC algorithms for graphs
low-local-memory setting $s = O(n^{\delta})$

- PRAM simulation:
  - any t-steps PRAM algorithm can be simulated in $O(t)$ rounds on MPC with low-local space
- Basic primitive:
  - sorting of N numbers on MPC with $s = O(n^{\delta})$ on $M = O(N/s)$ machines can be done deterministically in $O(1)$ rounds
- Basic "obstacle":
  - 1-vs-2-cycles conjecture:
    - distinguishing between a cycle on n vertices and two cycles on $n/2$ vertices requires $\Omega(\log n)$ rounds on an MPC with $s = O(n^{\delta})$

MPC algorithms for graphs low-local-memory setting

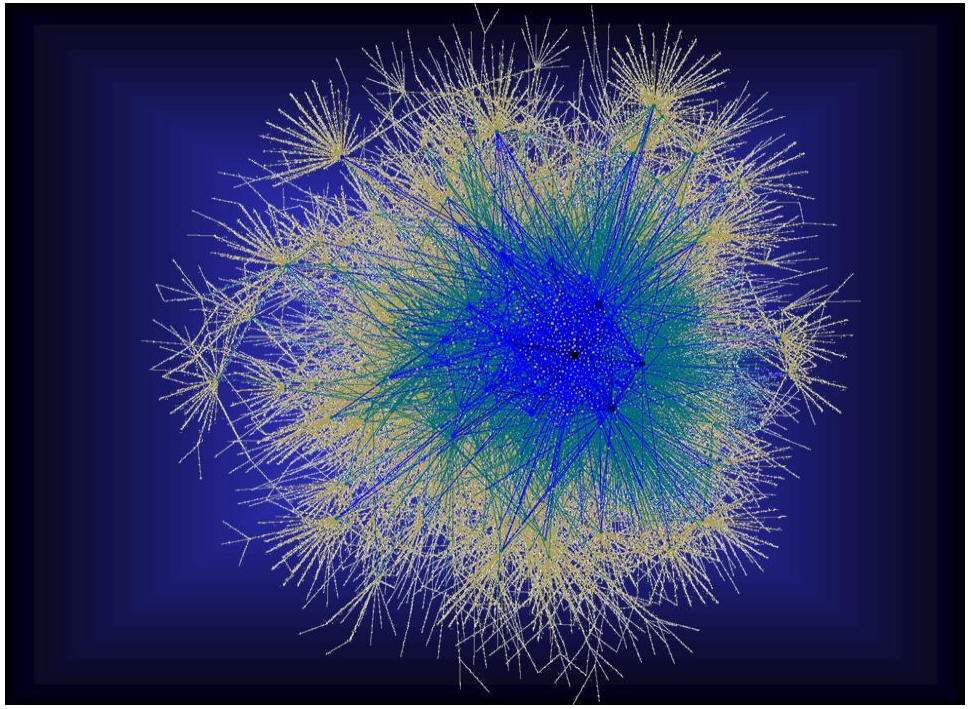
What can be done:

- PRAM simulations yields many MPC algorithms
- Main interest in the algorithmic community:
  - can we do better?

MPC algorithms for graphs
low-local-memory setting

Most fundamental graph problem:

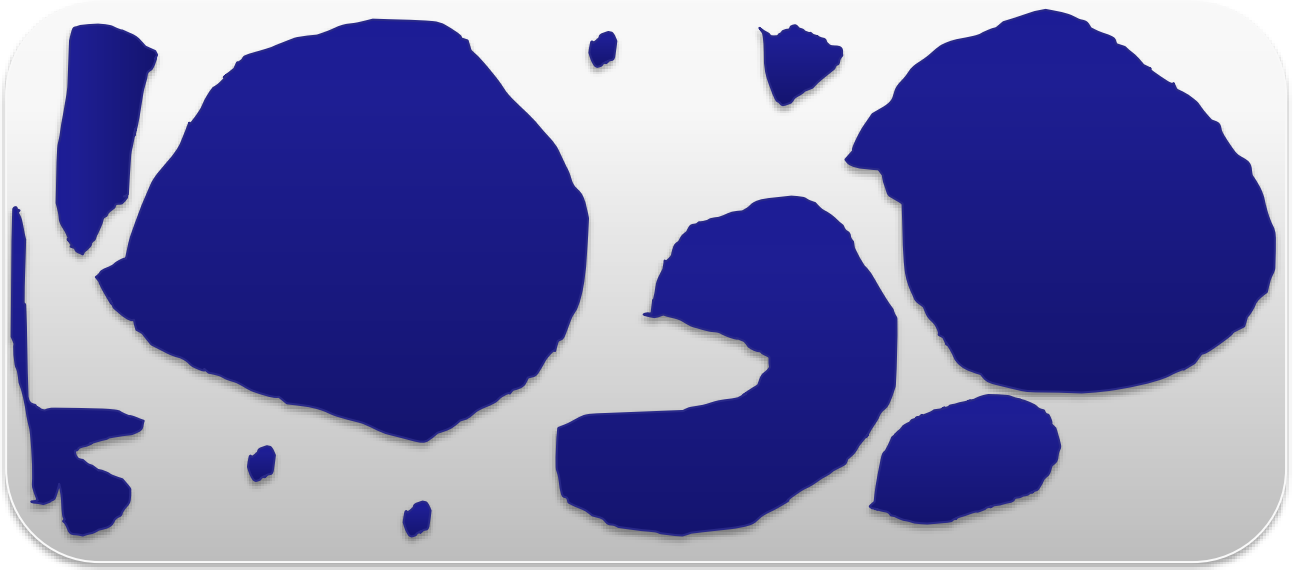
- is graph G connected?
- determine all connected components



MPC algorithms for graphs
low-local-memory setting

Most fundamental graph problem:

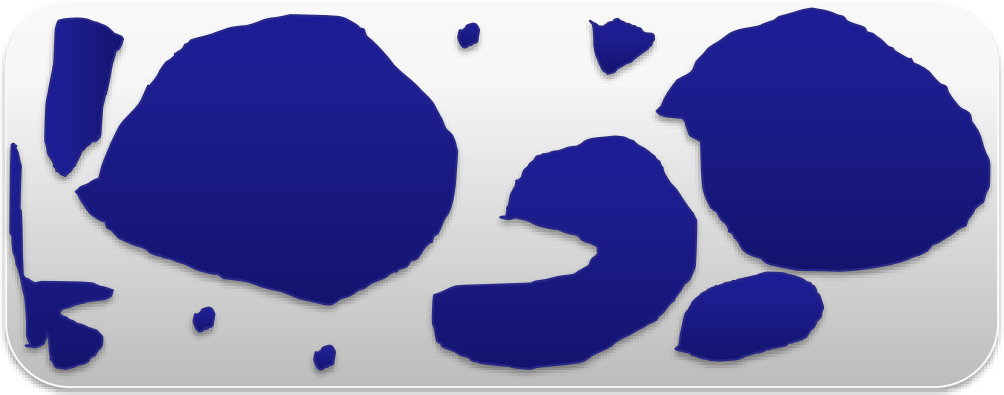
- is graph G connected?
- determine all connected components



Connectivity

- Input: graph $G = (V, E)$
  - $n = |V|, m = |E|$

- Determine all connected components of G
  - Compute $cc: V \longrightarrow \mathbb{N}$ that satisfies the following:
    - if $u, v \in V$ are connected then $cc(u) = cc(v)$
    - if $u, u \in V$ are not connected then $cc(u) \neq cc(v)$



Connectivity

- In the 90s, PRAM algorithms that solve the problem in $O(\log n)$ time
  - seems like the best we can hope for
  - extends to MPC

- 1-vs-2-cycles conjecture:
  - distinguishing between a cycle on n vertices and two cycles on $n/2$ vertices requires $\Omega(\log n)$ rounds on an MPC
  - we don't know how to prove such bound
  - proving it would imply some "hard" complexity bounds

$O(\log n)$-rounds connectivity on low-space MPC

Basic graph connectivity

Input: undirected graph $G = (V, E)$
Output: for each $u \in V$, $cc(u)$ is an ID of the connected component of u in G

1 mark every vertex $u \in V$ as *active* and label $cc(u) := u$
2 **while** G has an edge $\{x, y\} \in E$ with $cc(x) \neq cc(y)$ **do**
3 `/* Invariant: if a vertex` u `is active then` $cc(u) = u$`; if` u `is non-active then` $cc(u) = v$ `for some active vertex` v `in the` u`'s connected component */`
4 **for all** active vertices $u \in V$ **do in parallel** call u a *leader* with probability $\frac{1}{2}$
5 **for all** leaders $u \in V$ **do in parallel** mark all vertices in C_u as leaders
6 **for all** active non-leaders $u \in V$ **do in parallel**
7 **if** $\mathcal{N}(C_u)$ contains a leader **then**
8 find the smallest (with respect to the ID) leader vertex $v \in \mathcal{N}(C_u)$
9 mark u *non-active*
10 relabel each vertex with label $cc(u)$ by $cc(v)$
11 **end**
12 `/* Invariant: if a vertex` u `is active then` $cc(u) = u$`; if` u `is non-active then` $cc(u) = v$ `for some active vertex` v `in the` u`'s connected component */`
13 **end**
14 **end**

C_u – vertices v with $cc(v) = u$

$N(C_u)$ – neighbors of vertices in C_u

Connectivity

- In the 90s, PRAM algorithms that solve the problem in $O(\log n)$ time
  - seems like the best we can hope for
  - extends to MPC

- What about graphs with low diameter?
  - D = maximum diameter of any connected component
  - many "practical" graphs have low diameter
    - (often $D = O(\log n)$, for example for "random graphs")

MPC algorithms for graphs
low-local-memory setting

MPC algorithm:

in $O(\log D + \log\log n)$ rounds

using $O((n+m)/s)$ machines (optimal utilization)

MPC algorithms for graphs
low-local-memory setting

MPC algorithm:

in $O(\log D + \log\log n)$ rounds

using $O((n+m)/s)$ machines (optimal utilization)

Using matrix multiplication approach (computing transitive closure) we can solve the problem in $O(\log D)$ rounds using $O(n^3)$ machines

Main result: connectivity can be solved in (about) the same number of rounds with optimal number of machines

Connectivity

- Why can we do $O(\log D)$ rounds?
- If enough total space (or # machines) then it's "trivial"
- Let A be an adjacency matrix; I identity $n \times n$ matrix
  - $(A + I)^D$ determines all connected components!
  - $(A + I)^D[i, j] \neq 0$ iff i and j are in the same connected component
  - in fact $(A + I)^t[i, j] \neq 0$ iff i and j are at distance at most t

Connectivity

- Let A be an adjacency matrix; I identity $n \times n$ matrix
  - $(A + I)^D$ determines all connected components!
  - $(A + I)^D[i, j] \neq 0$ iff i and j are in the same connected component

- Since a square of a matrix can be computed in a constant number of rounds ➔

  deterministic $O(\log D)$ rounds MPC algorithm

Connectivity

- Let A be an adjacency matrix; I identity $n \times n$ matrix
  - $(A + I)^D$ determines all connected components!
  - $(A + I)^D[i, j] \neq 0$ iff i and j are in the same connected component

- Since a square of a matrix can be computed in a constant number of rounds ➔

  deterministic $O(\log D)$ rounds MPC algorithm

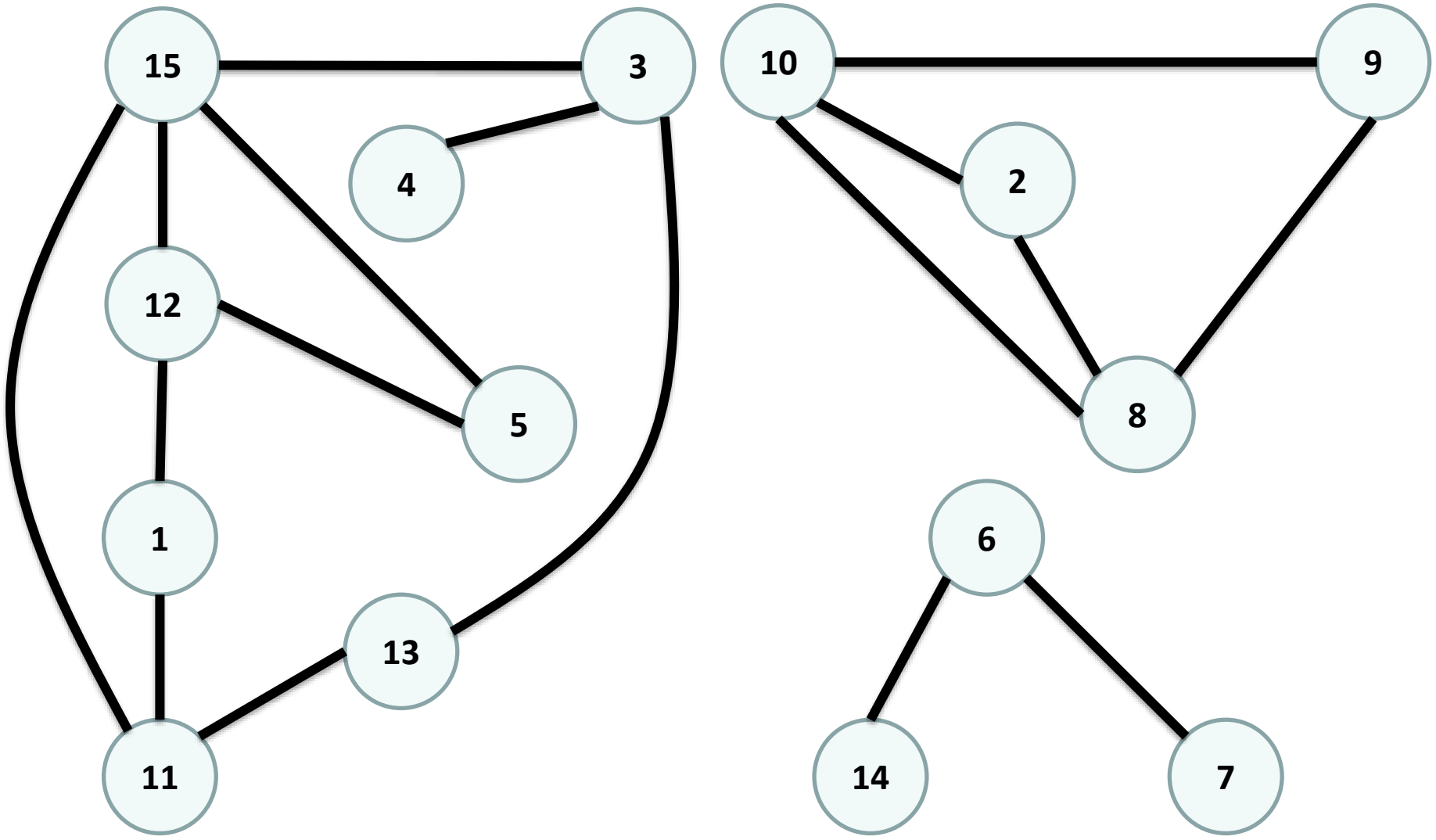
Requires a lot of global space (~ #machines) – $\Omega(n^3)$

Connectivity

- A tool to reduce total space to $O(n + m)$

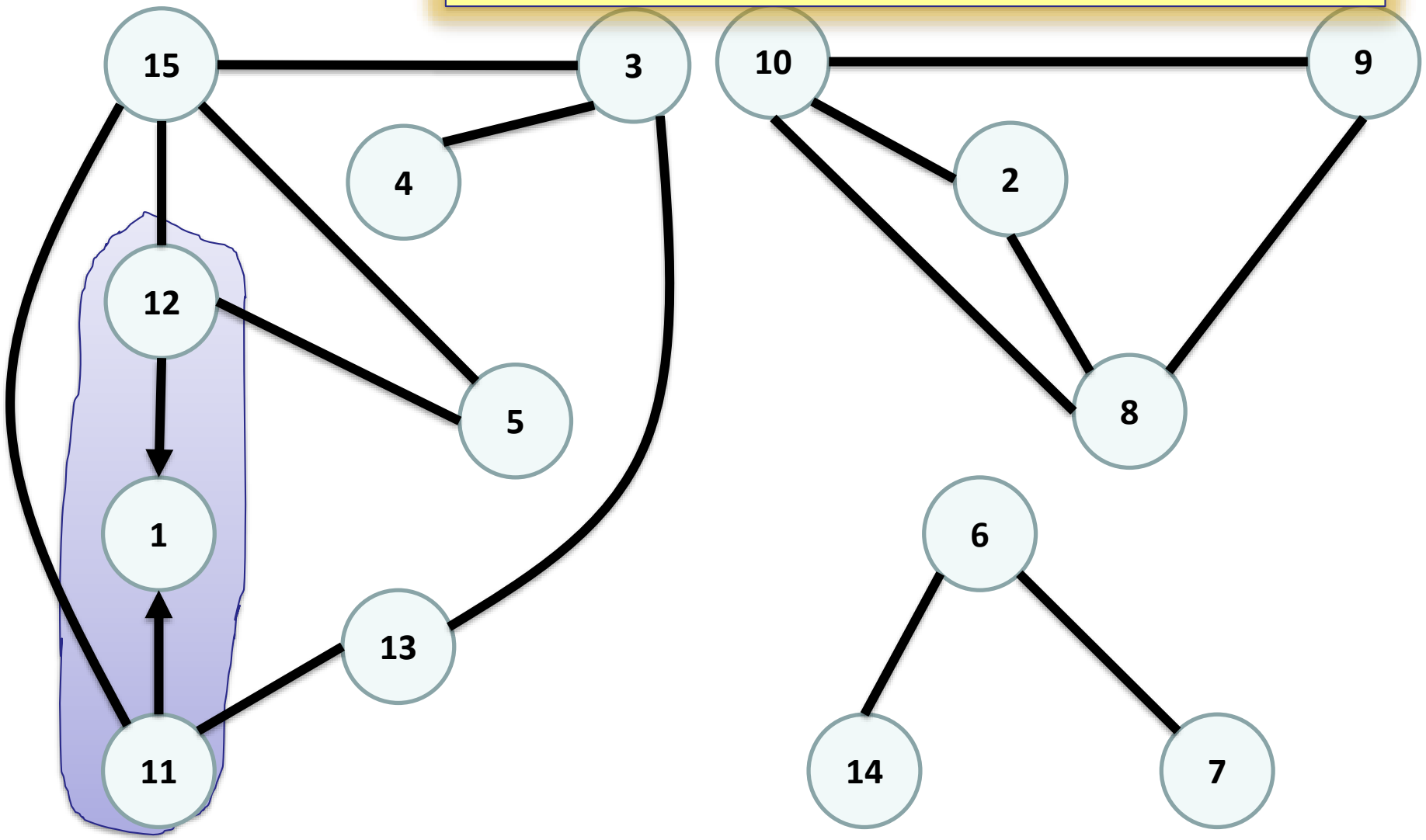
- Standard idea:
  - Take the input graph
  - Contract some edges to make the graph smaller while maintaining connectivity
  - Repeat until the graph fits on a single machine

Connectivity via contractions

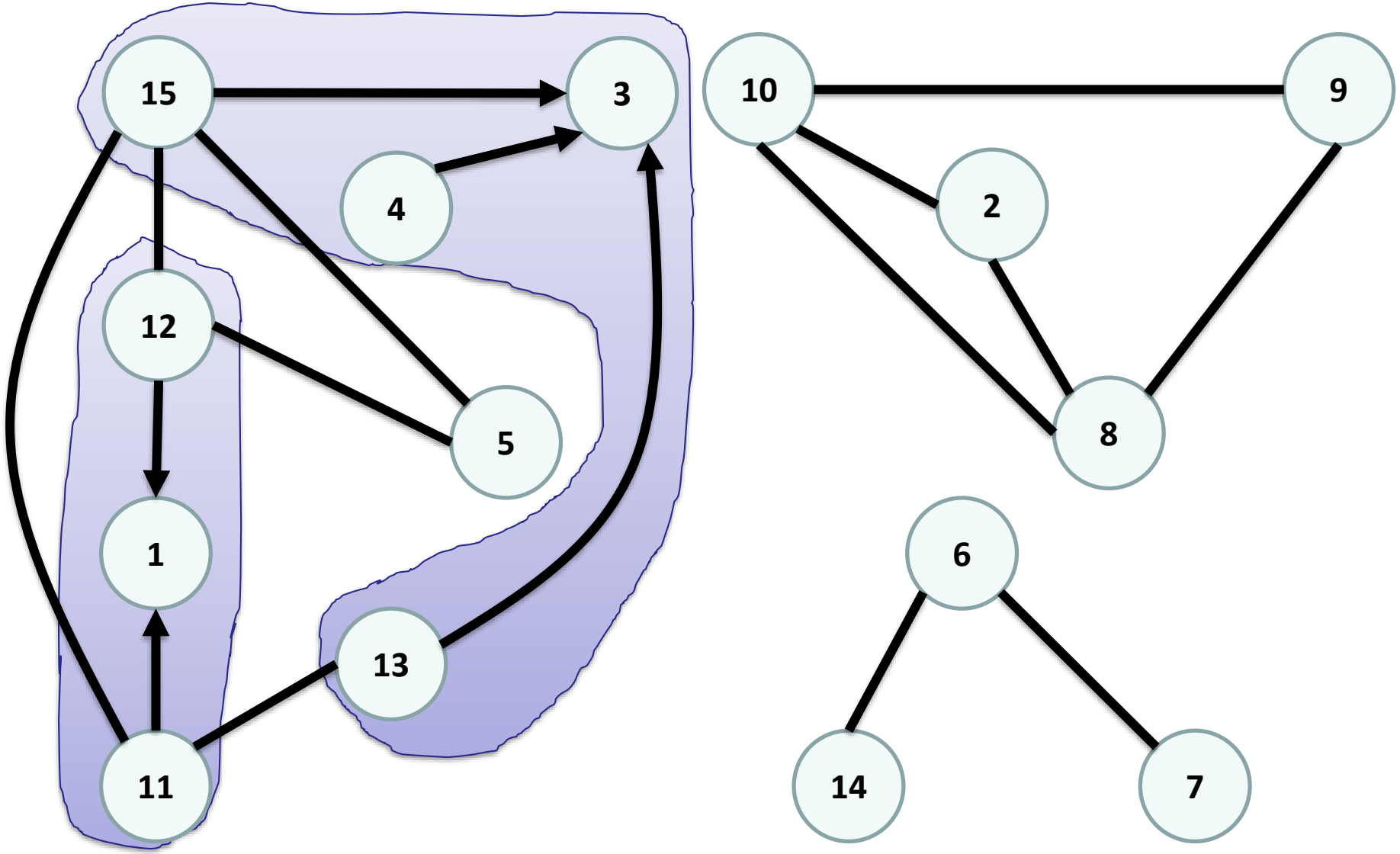


Connectivity via contractions

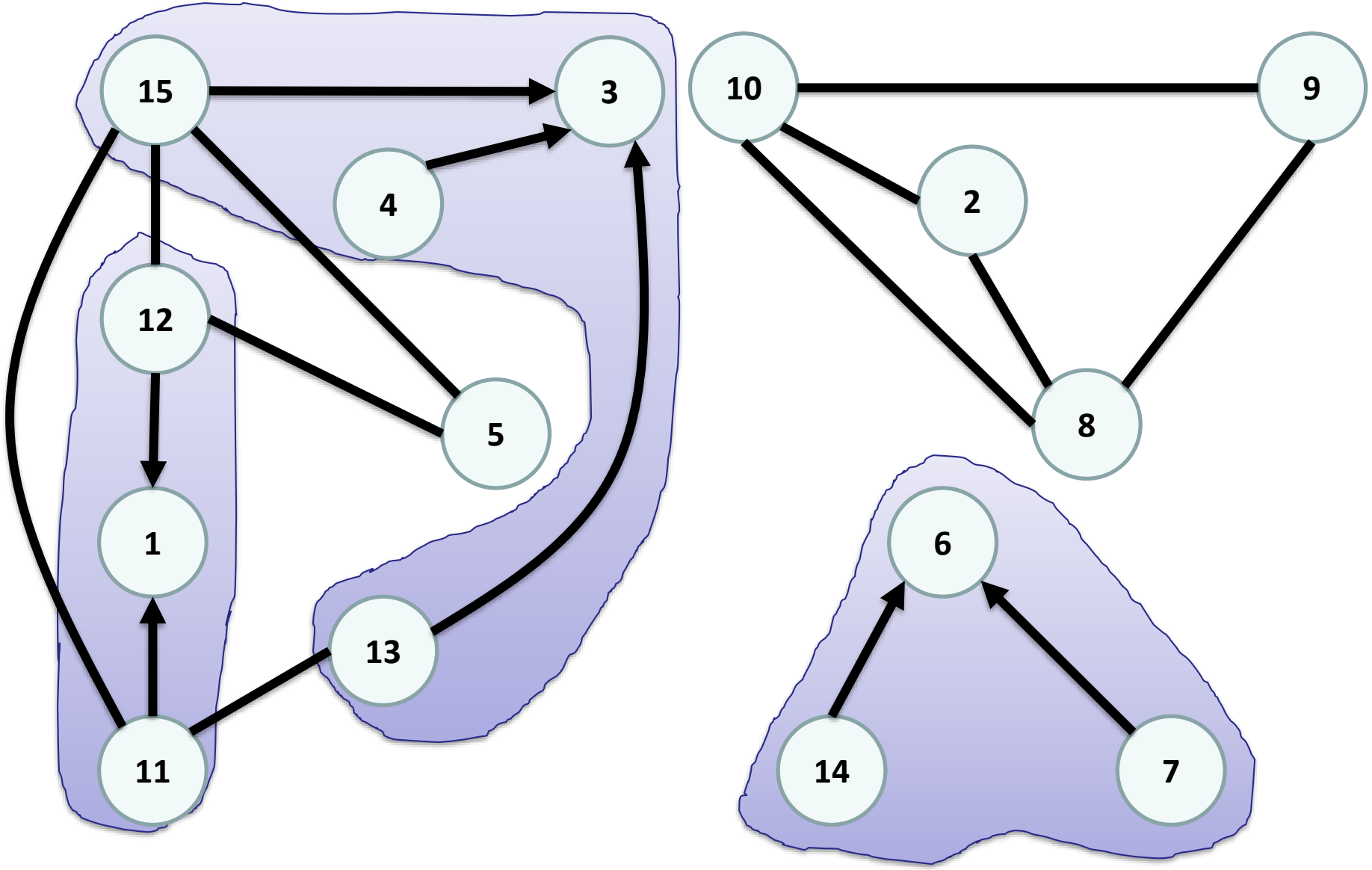
11 and 12 are in the same connected component as 1



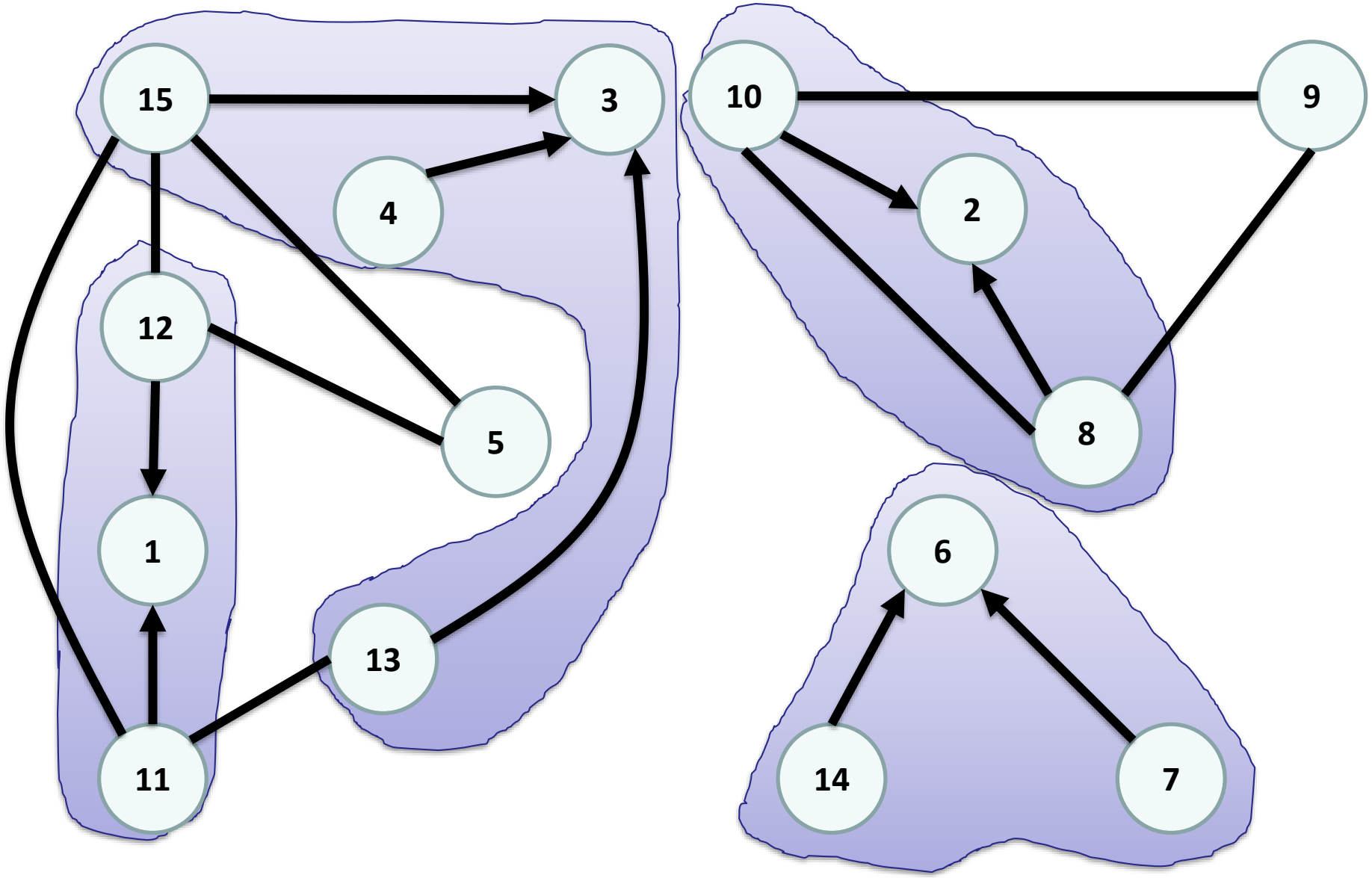
Connectivity via contractions



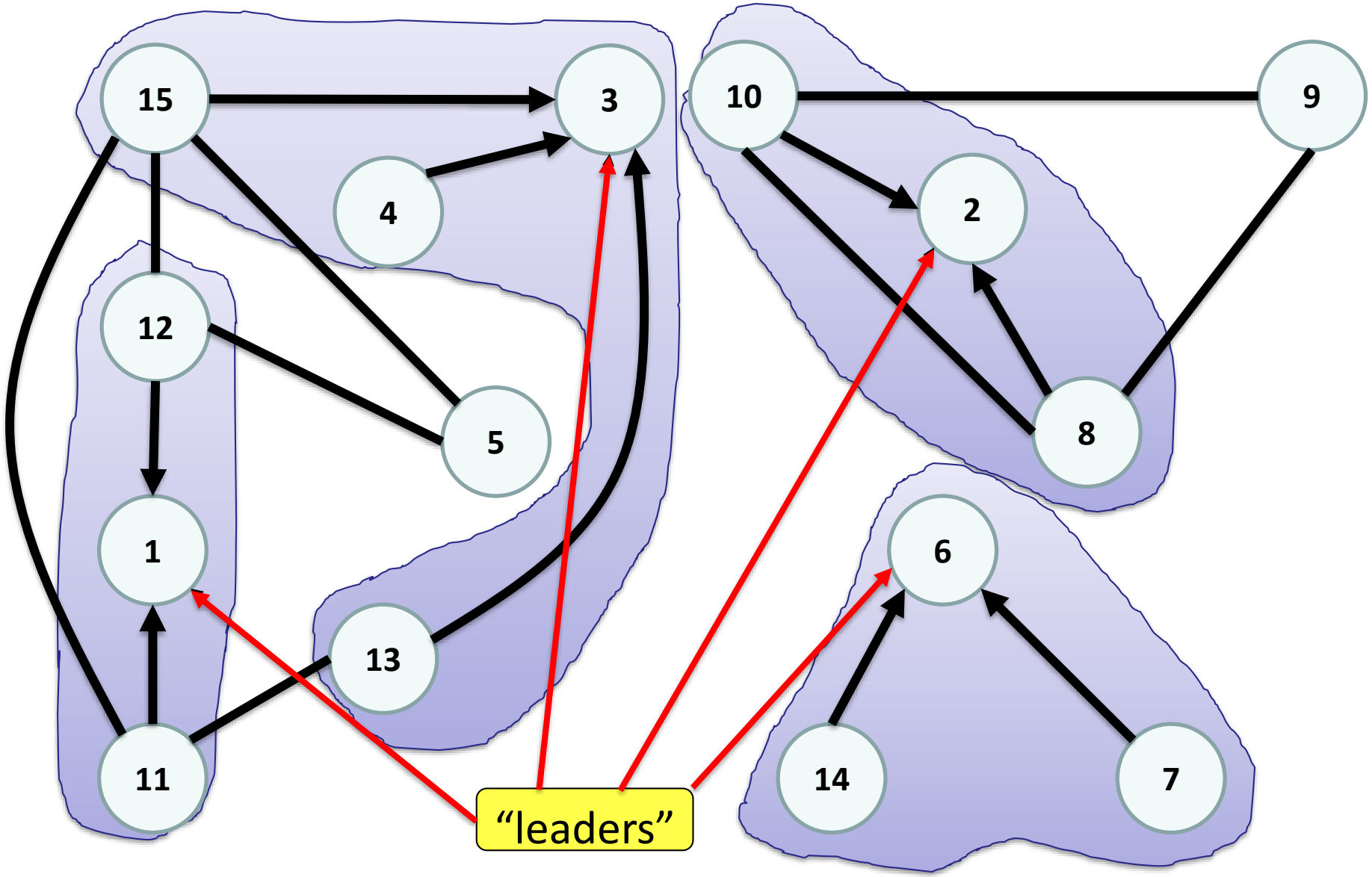
Connectivity via contractions



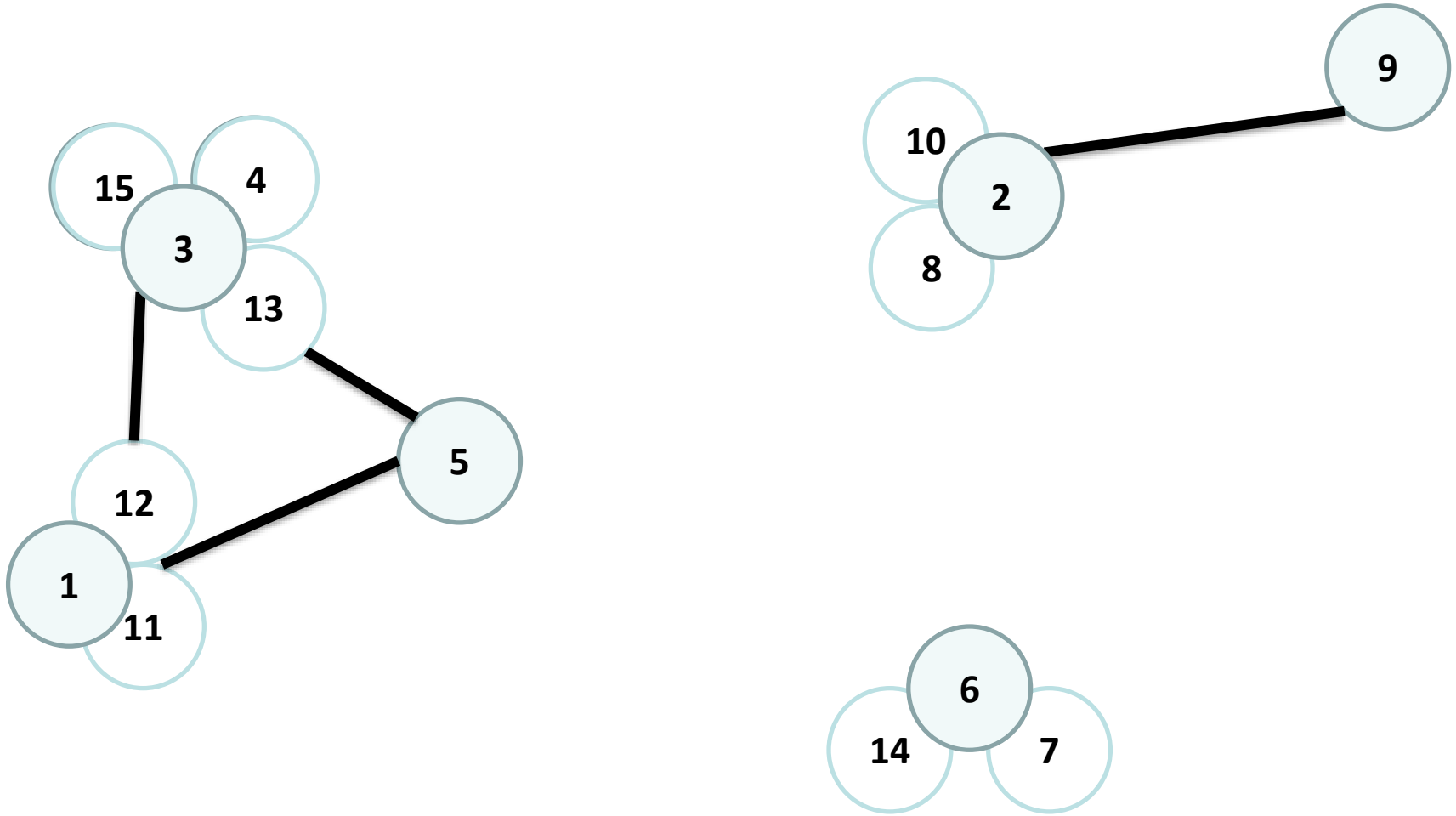
Connectivity via contractions



Connectivity via contractions



Connectivity via contractions



Connectivity

- A tool to reduce total space to $O(n+m)$

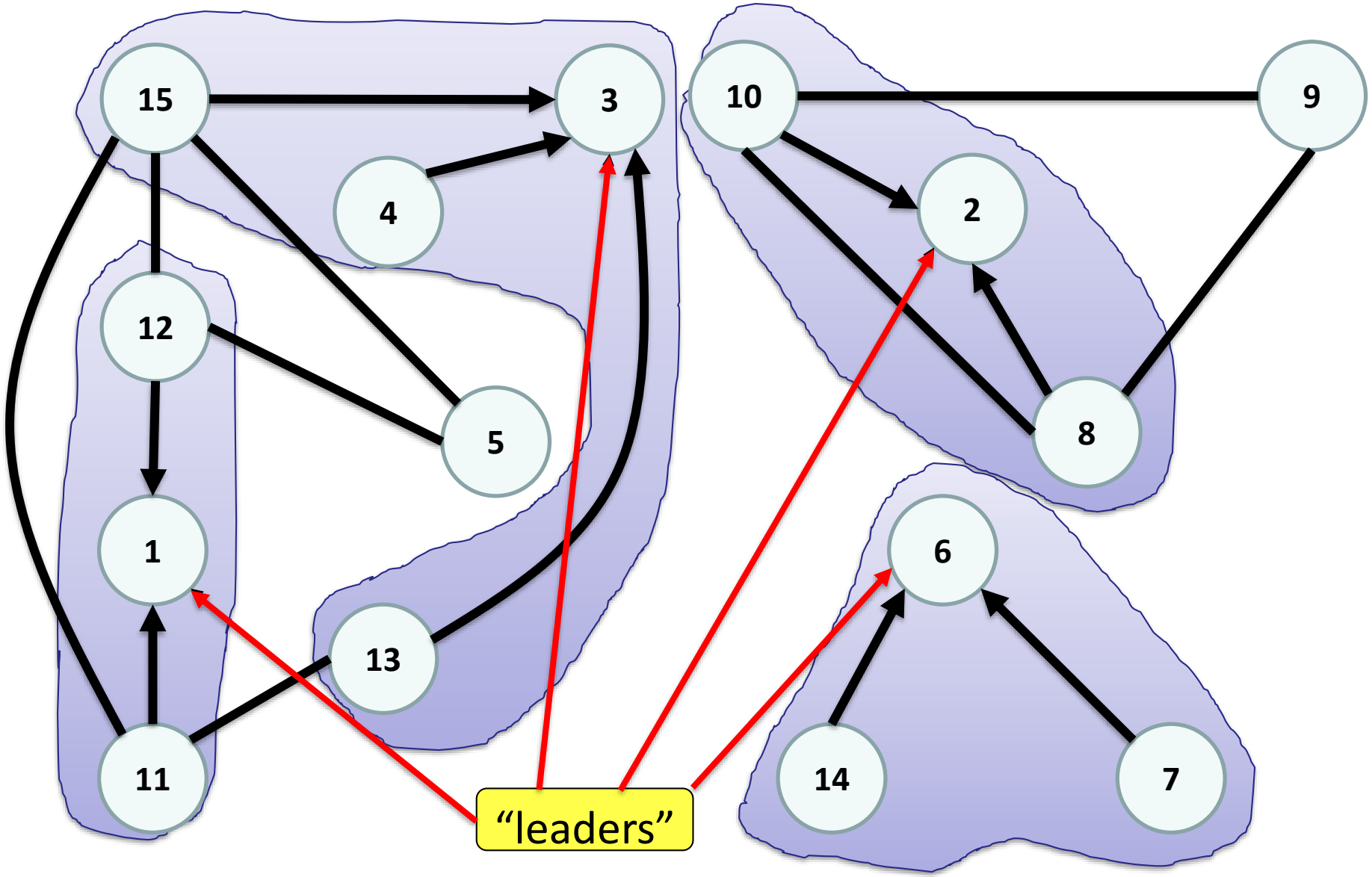
- Standard idea:
  - Take the input graph
  - Contract some edges to make the graph smaller while maintaining connectivity
  - Repeat until the graph fits on a single machine

Once we have smaller graph – we can use more space per vertex/edge!
We will use extra space to “enlarge” the graph

Connectivity via contractions

- Key trick:
- *Dominating set* Q:
  - Every vertex u is either in Q or has a neighbour in Q
- Idea:
  - Find a small dominating set Q
  - Contract each vertex to a neighbour (“leader”) in Q
  - Reduces number of vertices to $|Q|$

Connectivity via contractions



Connectivity via contractions

- Key trick:
- *Dominating set* Q:
  - Every vertex u is either in Q or has a neighbour in Q
- Idea:
  - Find a small dominating set Q
  - Contract each vertex to a neighbour ("leader") in Q
  - Reduces number of vertices to $|Q|$

- If *minimum degree* is d then one can "easily" find a dominating set of size $n \log n/d$
  - put a vertex to Q with probability $2 \cdot \log n/d$

Connectivity via contractions

This d will be going up in every iteration of the loop

- Take input graph G
- Reduce number of vertices to $n/\log^2 n$
- ➔ if we have linear total space, then each vertex has "space" for $\log^2 n$ vertices in its connected component
- "Expand" degree of each vertex to $\log^2 n$ (or d = whatever extra space per vertex is available)
- Find a small dominating set (of size $n/\log^3 n$ – or $\frac{n}{d \cdot \log n}$)
- Contract all vertices to the dominating set
- Repeat until entire graph fits a single machine

Expand degree to $\geq d$:

- If vertex is in connected component with $> d$ vertices ➔ connect it to $\geq d$ vertices in its connected component
- else ➔ detect entire connected component

Connectivity via contractions

- Take input graph G
- Reduce number of vertices to $n/\log^2 n$
- ➔ if we have linear total space, then each vertex has "space" for $\log^2 n$ vertices in its connected component
- "Expand" degree of each vertex to $\log^2 n$ (or $d =$ whatever extra space per vertex is available)
- Find a small dominating set (of size $n/\log^3 n$ – or $\frac{n}{d \cdot \log n}$)
- Contract all vertices to the dominating set
- Repeat until entire graph fits a single machine

- $O(\log \log n)$ rounds will suffice!

- Start with $N = n/\log^2 n$ and $d = \log^2 n$
- Reduce to $N = N \log n/d = n/\log^3 n$ and $d = \log^3 n$
- Reduce to $N = N \log n/d = n/\log^5 n$ and $d = \log^5 n$
- Reduce to $N = N \log n/d = n/\log^9 n$ and $d = \log^9 n$
- Reduce to $N = N \log n/d = n/\log^{17} n$ and $d = \log^{17} n$
- Reduce to $N = N \log n/d = n/\log^{33} n$ and $d = \log^{33} n$
- …

- Take input graph G
- Reduce number of vertices to $n/\log^2 n$
- ➔ if we have linear total space, then each vertex has "space" for $\log^2 n$ vertices in its connected component
- "Expand" degree of each vertex to $\log^2 n$ (or $d =$ whatever extra space per vertex is available)
- Find a small dominating set (of size $n/\log^3 n$ – or $\frac{n}{d \cdot \log n}$)
- Contract all vertices to the dominating set
- Repeat until entire graph fits a single machine

- $O(\log \log n)$ rounds will suffice!

Connectivity via contractions

- Take input graph G
- Reduce number of vertices to $n/\log^2 n$
- ➔ if we have linear total space, then each vertex has "space" for $\log^2 n$ vertices in its connected component
- "Expand" degree of each vertex to $\log^2 n$ (or $d =$ whatever extra space per vertex is available)
- Find a small dominating set (of size $n/\log^3 n$ – or $\frac{n}{d \cdot \log n}$)
- Contract all vertices to the dominating set
- Repeat until entire graph fits a single machine

- $O(\log \log n)$ rounds will suffice!

Can be performed in $O(\log \log n)$ rounds!

Expansion can be performed in $O(\log D)$ rounds!

Connectivity via contractions

- Take input graph G
- Reduce number of vertices to $n/\log^2 n$
- ➔ if we have linear total space, then each vertex has "space" for $\log^2 n$ vertices in its connected component
- "Expand" degree of each vertex to $\log^2 n$ (or $d =$ whatever extra space per vertex is available)
- Find a small dominating set (of size $n/\log^3 n$ – or $\frac{n}{d \cdot \log n}$)
- Contract all vertices to the dominating set
- Repeat until entire graph fits a single machine
- $O(\log \log$

Theorem: One can determine connected components in $O(\log D \log \log n)$ rounds on an MPC

Theorem: One can determine connected components in $O(\log D + \log \log n)$ rounds on an MPC

Connectivity via contractions

- Take input graph G
- Reduce number of vertices to $n/\log^2 n$
- ➔ if we have linear total space, then each vertex has "space" for $\log^2 n$ vertices in its connected component
- "Expand" degree d ... ertex is available)
- Find a small domi...
- Contract all vertices to the dominating set
- Repeat until entire graph fits a single machine
- $O(\log \log$ ...

These algorithms are ***randomized***:
- Reduction to $n/\log^2 n$ vertices is randomized
- Finding a small dominating set is randomized

Theorem: One can determine connected components in $O(\log D \log\log n)$ rounds on an MPC

Theorem: One can determine connected components in $O(\log D + \log\log n)$ rounds on an MPC

Connectivity via contractions

- Take input graph G
- Reduce number of vertices to $n/\log^2 n$
- ➔ if we have linear total space, then each vertex has "space" for $\log^2 n$ vertices in its connected component
- "Expand" degree of each vertex to $\log^2 n$ (or $d =$ whatever extra space per vertex is available)
- Find a small dominating set (of size $n/\log^3 n$ – or $\frac{n}{d \cdot \log n}$)
- Contract all vertices to the dominating set
- Repeat until entire graph fits a single machine

- $O(\log \log n)$ rounds will suffice!

Reduction to $n/\log^2 n$ vertices

- In a constant number of rounds reduce to $n/2$ vertices
  - Not very difficult

- Central step:
  - Find a matching of size $O(N)$ on a path of length N
  - Easy using randomization:
    - remove each edge with probability ½
    - a constant fraction of edges will become isolated ➔ form a matching of size $O(N)$

Matching on a path

Matching on a path

remove each edge with prob. $1/2$

Matching on a path

Reduction to $n/\log^2 n$ vertices

- In a constant number of rounds reduce to $n/2$ vertices
  - Not very difficult with randomization

- Central step:
  - Find a matching of size $O(N)$ on a path of length N
  - Easy using randomization:
    - remove each edge with probability ½
    - a constant fraction of edges will become isolated ➔ form a matching of size $O(N)$

Reduction to $n/\log^2 n$ vertices

- In a constant number of rounds reduce to $n/2$ vertices
  - Not very difficult with randomization

- Central step:
  - Find a matching of size $O(N)$ on a path of length N
  - Easy using randomization:
    - remove each edge with probability ½
    - a constant fraction of edges will become isolated ➔ form a matching of size $O(N)$

We can do it deterministically in a constant number of rounds!

Matching on a path

- Matching algorithm relies on **3-wise independent random variables** (random choices for the edges)

Random variables are **k-wise independent** on if any k of them are independent

Matching on a path

Matching on a path

remove each edge with prob. $1/2$

Matching on a path

Matching on a path

- Matching algorithm relies on 3-wise independent random variables (random choices for the edges)

- There are families of 3-wise independent random hash functions of size $O(n^3)$
- Take such a family $\mathcal{H}$
- There is some $h \in \mathcal{H}$ that corresponds to a matching of linear size

- We can find one such $h \in \mathcal{H}$ in a constant number of rounds on an MPC (with total linear space)

Matching on a path

- We can find one such $h^* \in \mathcal{H}$ in a constant number of rounds on an MPC (with total linear space)

- $|\mathcal{H}| \leq n^3$ ➔ $\mathcal{H}$ is represented using $3 \log n$ bits

Method of conditional probabilities

For us: F = size of the matching generated by randomly chosen edges

Method of conditional probabilities by Erdös and Selfridge (1973) and Raghavan (1988):

Find $h^* \in \mathcal{H}$ by a sequence of repeatedly refining $\mathcal{H}$ into smaller subsets, $\mathcal{H} = H_0 \supseteq H_1 \supseteq \cdots$, until we end up with a one element set defining h^*.

The way of refining $\mathcal{H}$ is led by not decreasing (or not increasing) the conditional expectation of some target function F on $\mathcal{H}$, that is, to ensure that

$$E_{h \in H_{i+1}}[F(h)] \geq E_{h \in H_i}[F(h)]$$

Main difficulty: to efficiently compute conditional expectations, and to bound the number of iterations needed to find a one element subset of $\mathcal{H}$.

Method of conditional probabilities

- We can find one such $h^* \in \mathcal{H}$ in a constant number of rounds on an MPC (with total linear space)
- $|\mathcal{H}| \leq n^3$ ➔ $\mathcal{H}$ is represented using $3 \log n$ bits
- Split the bits into chunks of size $t = \log_2 s = \delta \log_2 n = O(\log n)$, such that $2^t = s$ words fits into a single machine
- We run in $3 \log n / t = 3 \log_s n = 3/\delta$ phases:
  - phase i determines bits $(i-1) \cdot t + 1, \ldots, i \cdot t$ of h^*

Method of conditional probabilities

- We can find one such $h^* \in \mathcal{H}$ in a constant number of rounds on an MPC (with total linear space)

- $|\mathcal{H}| \leq n^3$ ➔ $\mathcal{H}$ is represented using $3 \log n$ bits

- Split the bits into chunks of size $t = \log_2 s = \delta \log_2 n = O(\log n)$, such that $2^t = s$ words fits into a single machine

- We run in $3 \log n / t = 3 \log_s n = 3/\delta$ phases:
  - phase i determines bits $(i-1) \cdot t + 1, \ldots, i \cdot t$ of h^*

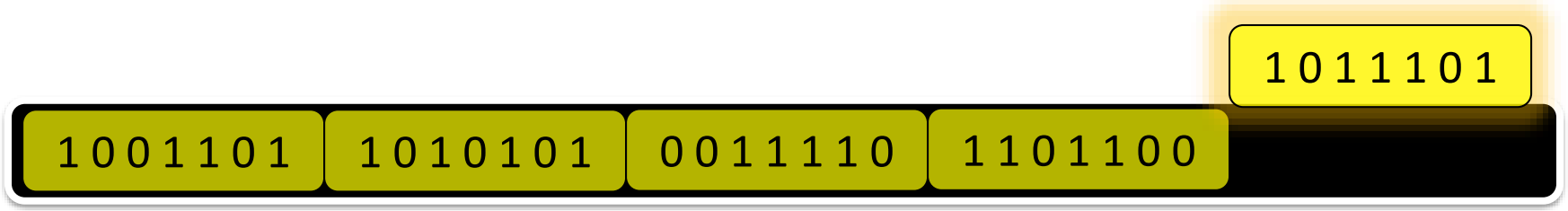
Method of conditional probabilities

For that, at the beginning of phase i:

- each machine knows the bits of the first $i-1$ parts of h^* (that is, prefix of $(i-1)t = (i-1)\log_2 s$ bits).

Next, each machine considers extension of the current bits by all possible seeds of length $t = \log_2 s$.

- Since $2^t = s$ fits local memory, this operation can be performed on every machine individually.

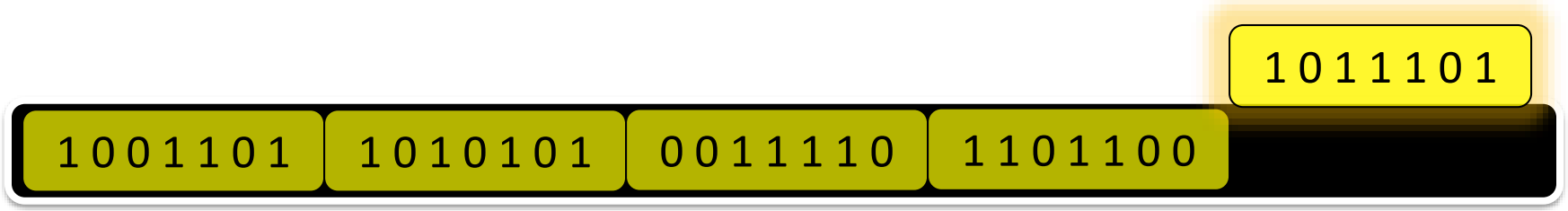


Method of conditional probabilities

For every possible $\log s$-bit extension of h^*:

- every machine computes locally the expected cost of the target function F (which is the size of the matching) with this seed, producing s different values for the target function, one for each partial seed

Then, one aggregates these values from all machines and chooses the $\log s$-bit extension of h^* which maximizes the target function (matching size) using the method of conditional probabilities.



Method of conditional probabilities ➔ linear matching on a path

- If we sum this up, then we obtain the following:

Theorem: In a constant number of MPC-rounds, one can deterministically find a linear-size matching on a path

Deterministically finding a small dominating set

- Similar approach
- Using significantly more sophisticated machinery of ε-approximate $O(\log n)$-wise independent hash functions

❑ ε-approximate $O(\log n)$-wise independent hash functions:

- ➢ requires ε to be tiny ➔ $O(\log n)$-bits representation
- ➢ requires the use of strong concentration bounds for (slightly) biased random variables
- ➢ MPC computation more tricky (since the progress function F is more complex)

Connectivity via contractions

Theorem: One can determine connected components in $O(\log D \log\log n)$ rounds on an MPC

Theorem: One can determine connected components in $O(\log D + \log\log n)$ rounds on an MPC

Theorem: These algorithms can be derandomized without any asymptotic loss:
deterministically in $O(\log D + \log\log n)$ rounds

MPC algorithms for graphs
low-local-memory setting

Algorithm/approach for connectivity can be applied to other related problems:

- Find a spanning tree
- Biconnected components
- Minimum spanning forest
- Shortest paths

MPC algorithms for graphs
low-local-memory setting

In $O(\log D \cdot \log \log_{m/n} n)$ rounds we can compute a *rooted spanning forest* of G

If we have more machines (say, $\left(O(n+m)\right)^{1+\gamma}/s$), then we can solve this task in $O(\log D \cdot \log(\frac{\log n}{2+\gamma \log n}))$ rounds

- when $\gamma = \Omega(1)$ then it's just $O(\log D)$ rounds

Minimum spanning forest with more machines – in $O(\log D_{MFS})$ rounds, where D_{MFS} = diameter of a minimum spanning forest

Some more related research

- Recent advances
  - approximating maximum matching,
  - maximal independent set,
  - graph $(\Delta + 1)$-coloring
  - random walks (and PageRanking)

- Some of these bounds use similar techniques

Final thoughts

- Fundamental study of MPC computation necessary to understand modern parallel computing
  - To develop tools for fast parallel algorithms
- One can design efficient deterministic MPC algorithms
- MPC can do more than PRAM
  - even with low space
  - even deterministically

THANK YOU!