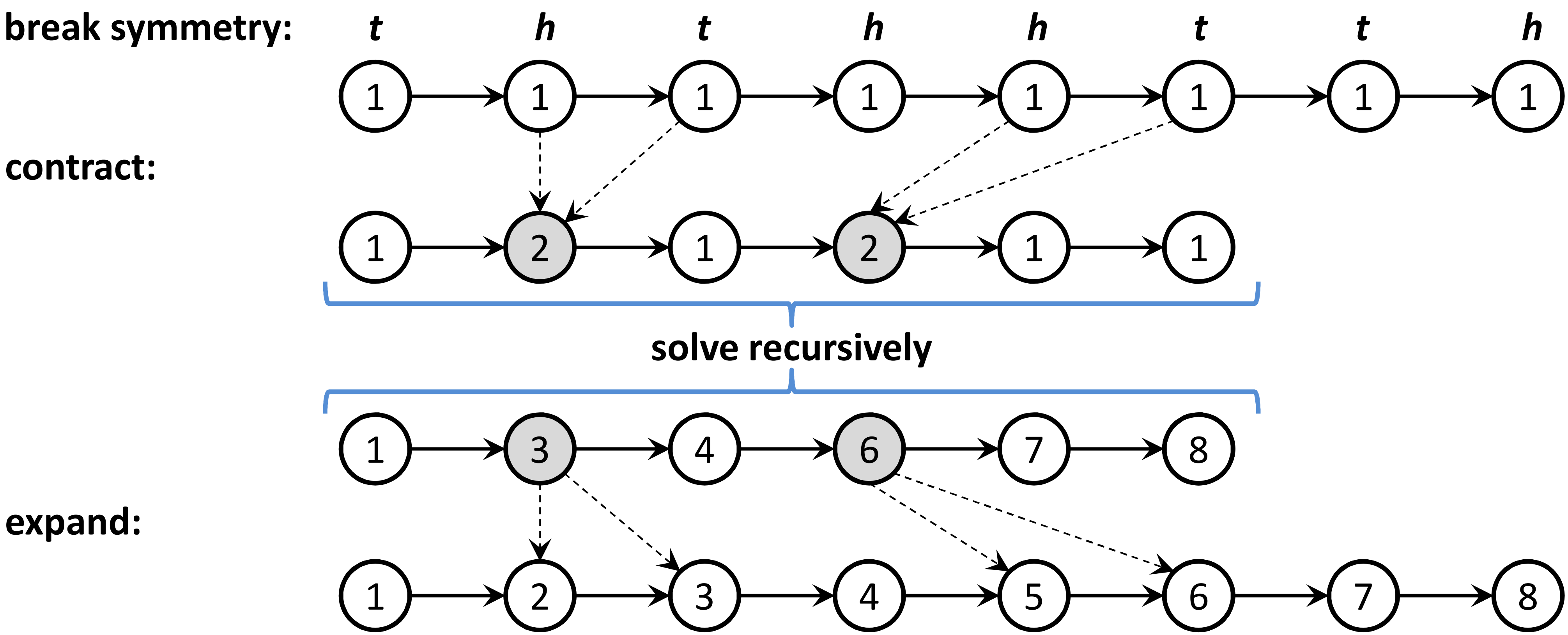
CSE 638: Advanced Algorithms

Lectures 10 & 11
(Parallel Connected Components)

Rezaul A. Chowdhury

Department of Computer Science
SUNY Stony Brook
Spring 2013

Symmetry Breaking: List Ranking



1. Flip a coin for each list node
2. If a node u points to a node v, and u got a head while v got a tail, combine u and v
3. Recursively solve the problem on the contracted list
4. Project this solution back to the original list

Symmetry Breaking: List Ranking

In every iteration a node gets removed with probability $\frac{1}{4}$

(as a node gets head with probability $\frac{1}{2}$ and the next node gets tail with probability $\frac{1}{2}$).

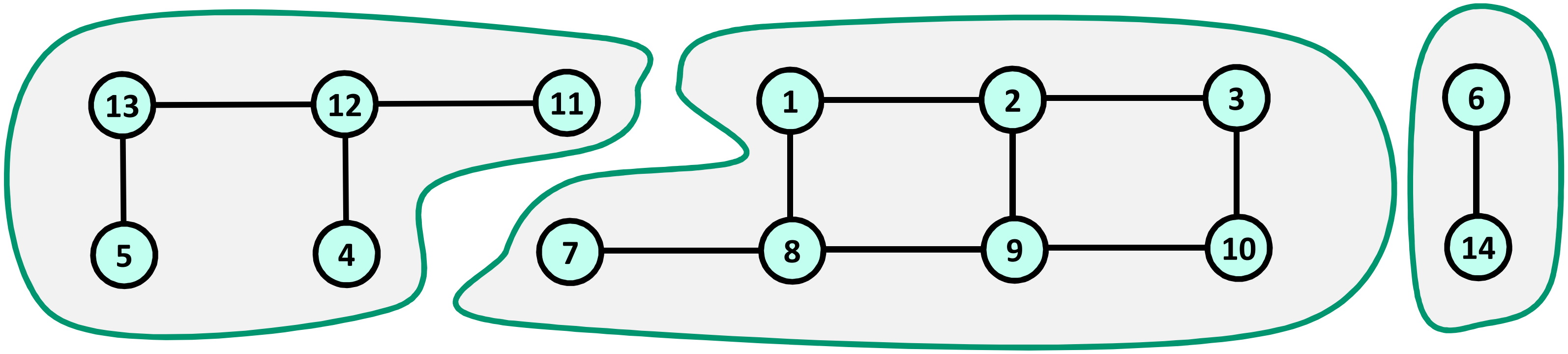
Hence, a quarter of the nodes get removed in each iteration (expected number).

Thus the expected number of iterations is $\Theta(\log n)$.

In fact, it can be shown that with high probability,

$$T_1(n) = O(n) \text{ and } T_\infty(n) = O(\log n)$$

Graph Connectivity

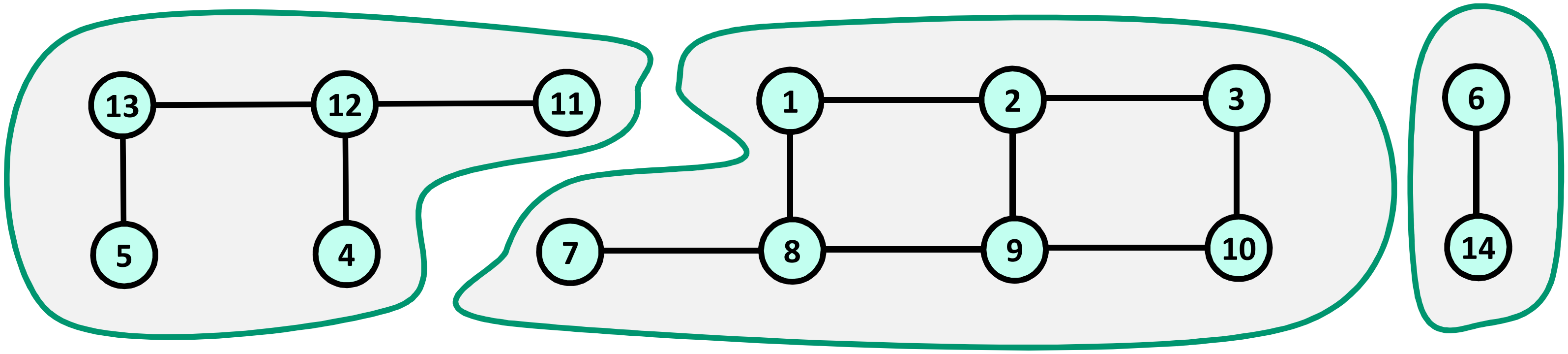


Connected Components: A *connected component C* of an undirected graph *G* is a maximal subgraph of *G* such that every vertex in *C* is reachable from every other vertex in *C* following a path in *G.*

Problem: Given an undirected graph identify all its connected components.

Suppose *n* is the number of vertices in the graph, and
m is the number of edges.

Graph Connectivity



Problem: Identify All connected components of an undirected graph.

Suppose n is the number of vertices in the graph, and

m is the number of edges.

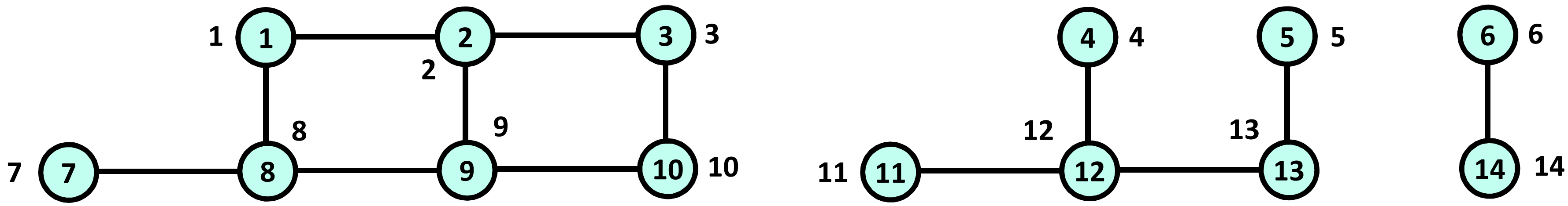
Serial Algorithms: Easy to solve in $\Theta(m + n)$ time using

- Depth First Search (DFS)
- Breadth First Search (BFS)

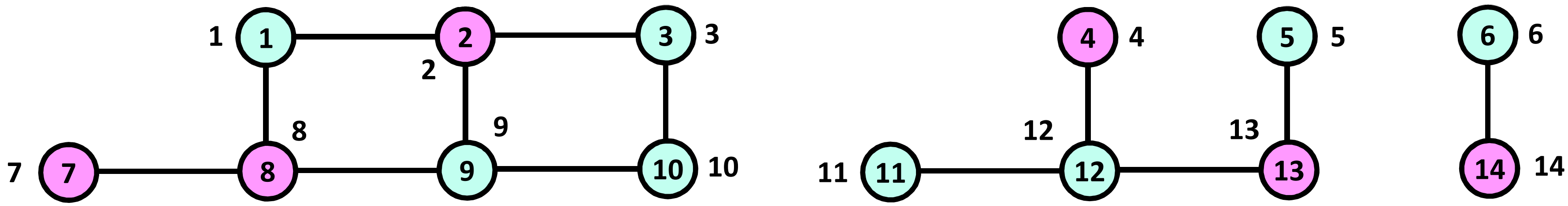
Parallel Algorithms:

- **BFS & DFS:** most efficient polylogarithmic depth algorithms are terribly work inefficient
- **Graph Contraction:** Can reach polylogarithmic depth without giving up too much or even anything at all in work-efficiency

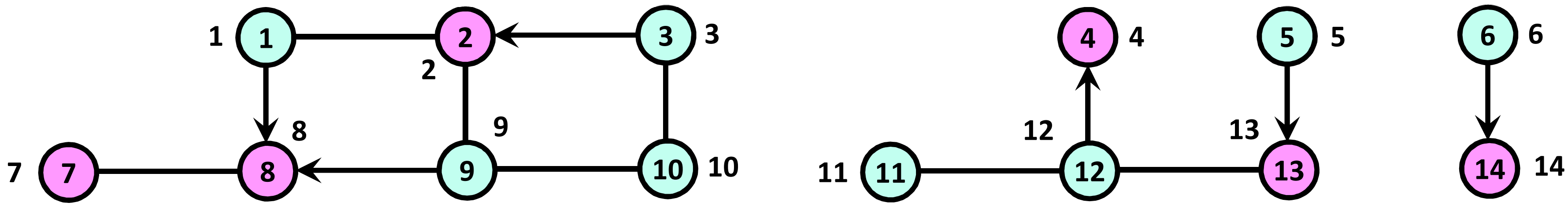
Randomized Parallel Connected Components



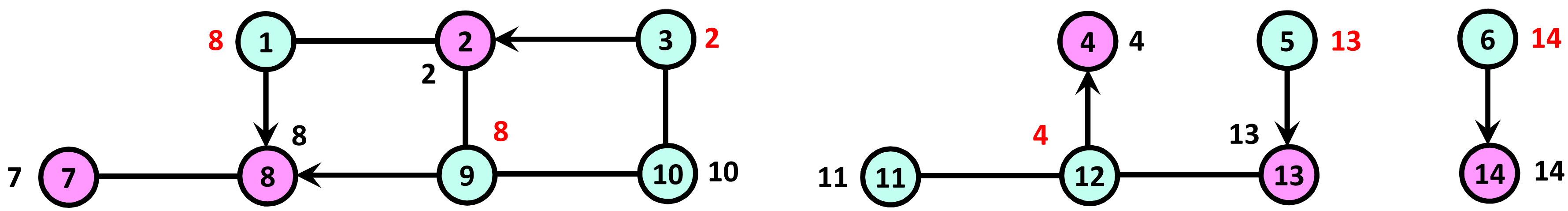
Randomized Parallel Connected Components



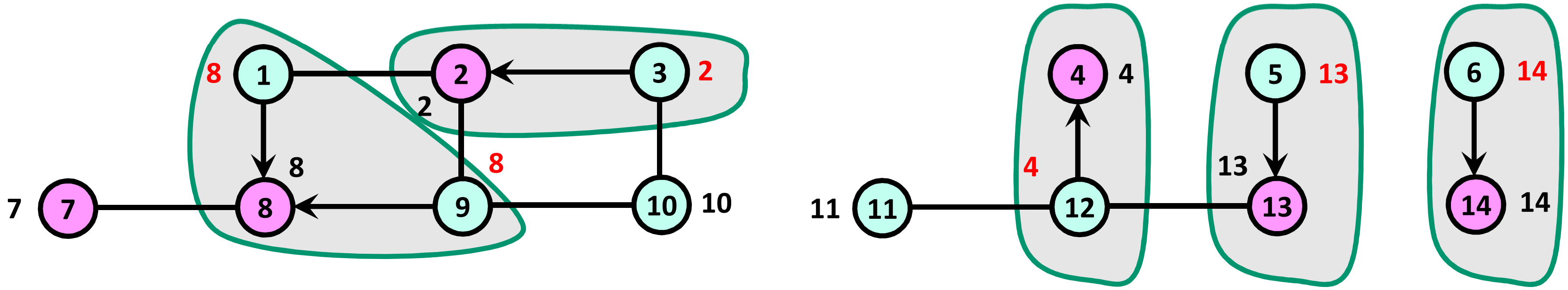
Randomized Parallel Connected Components



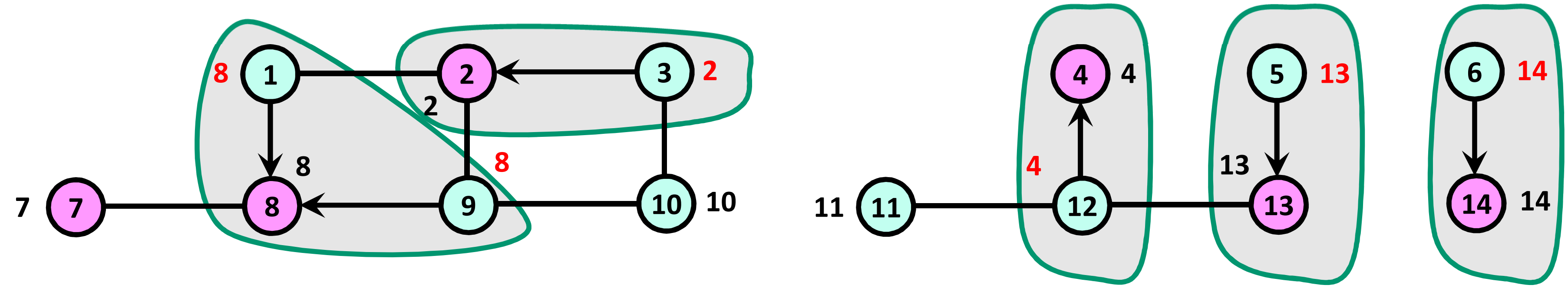
Randomized Parallel Connected Components



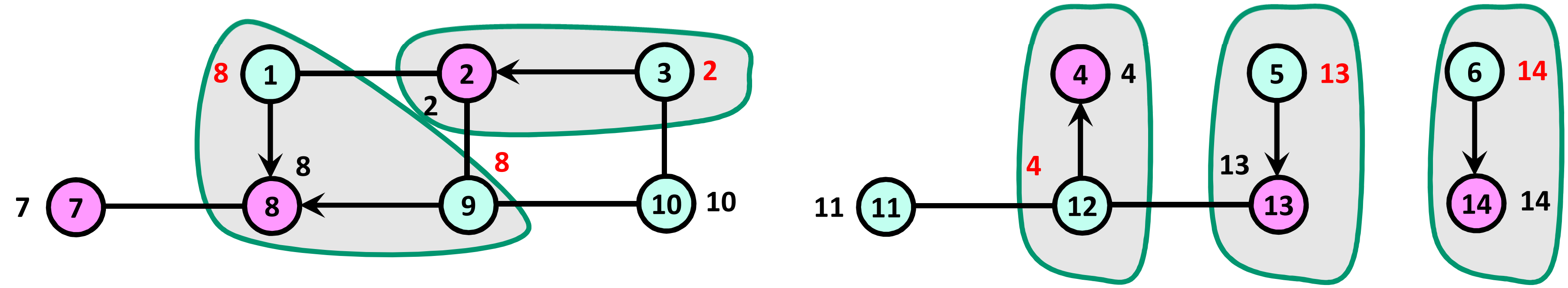
Randomized Parallel Connected Components

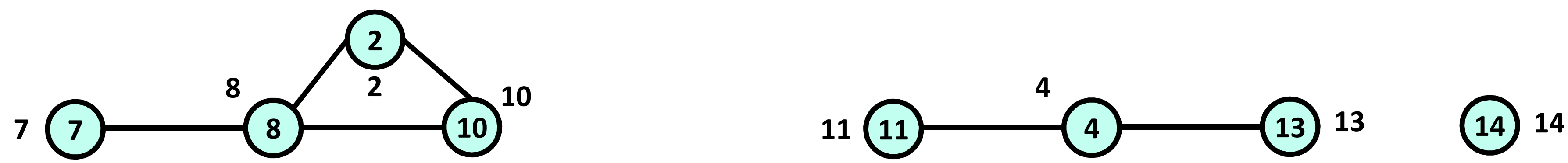


Randomized Parallel Connected Components

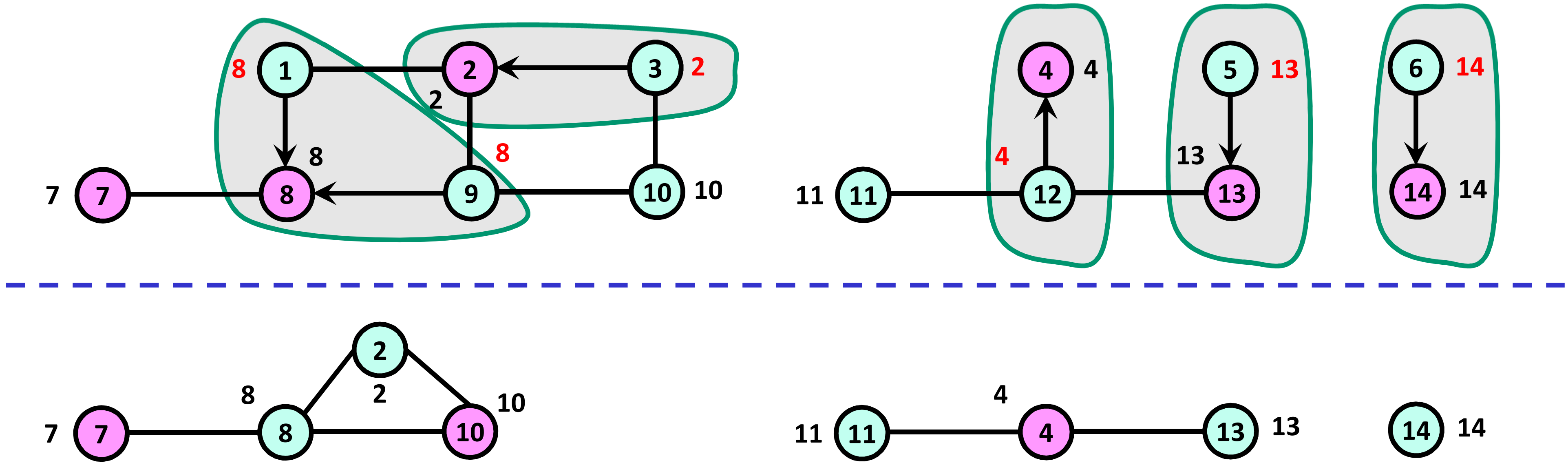


Randomized Parallel Connected Components

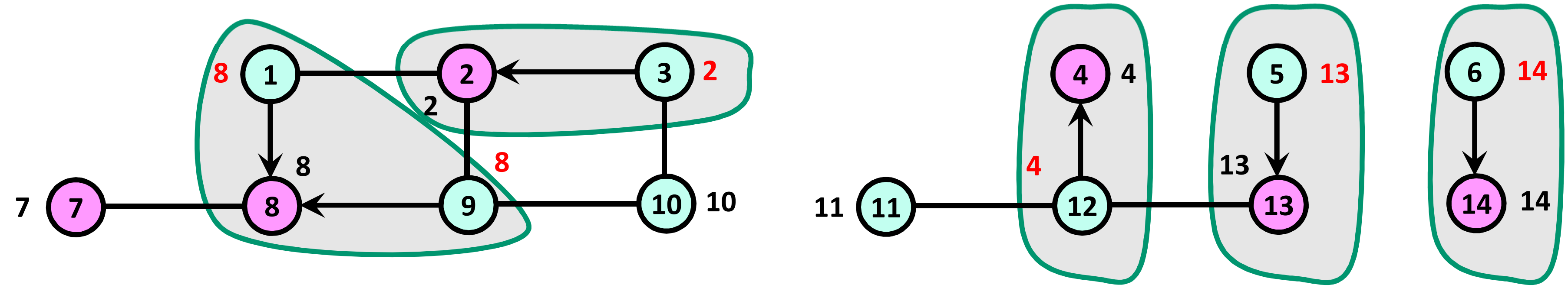


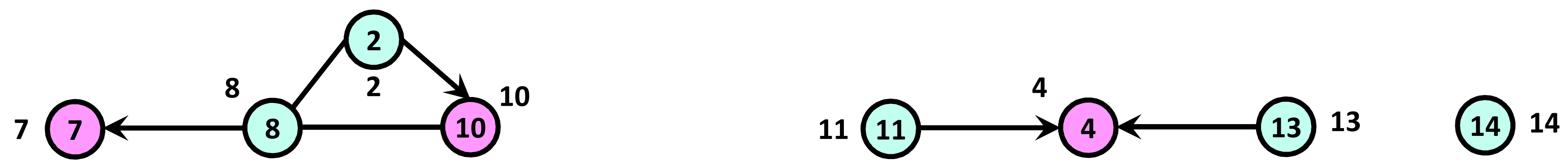


Randomized Parallel Connected Components

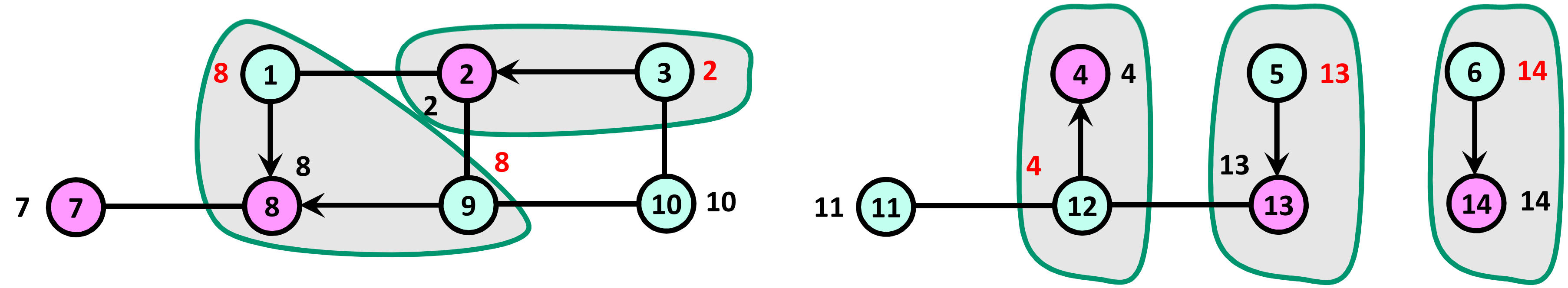


Randomized Parallel Connected Components

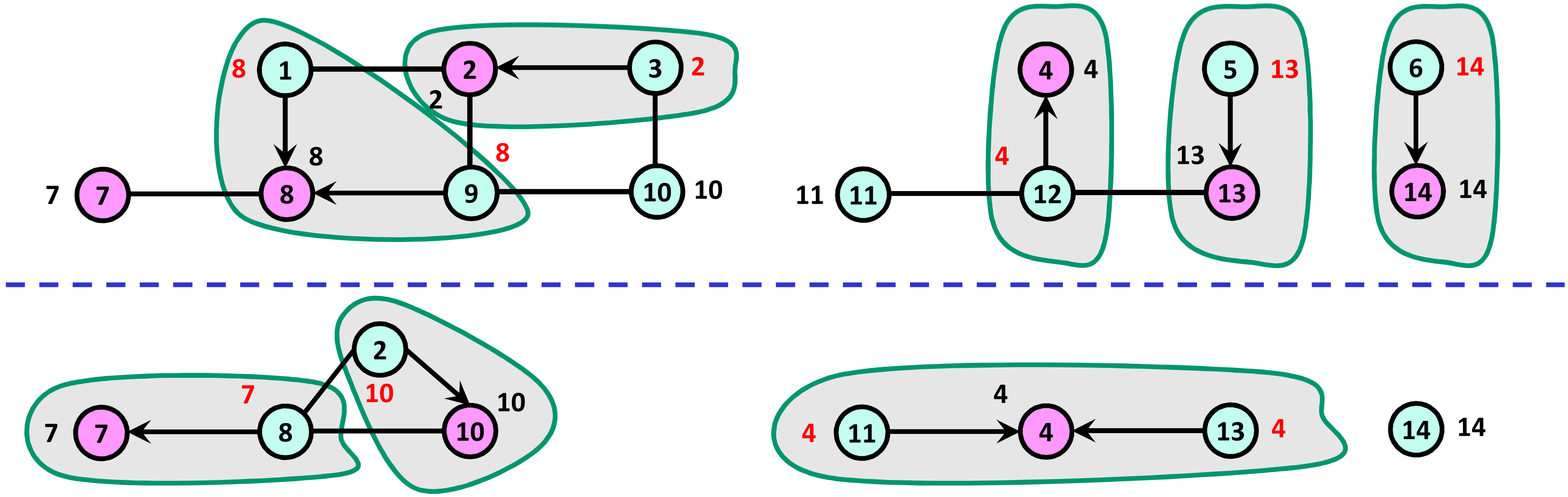




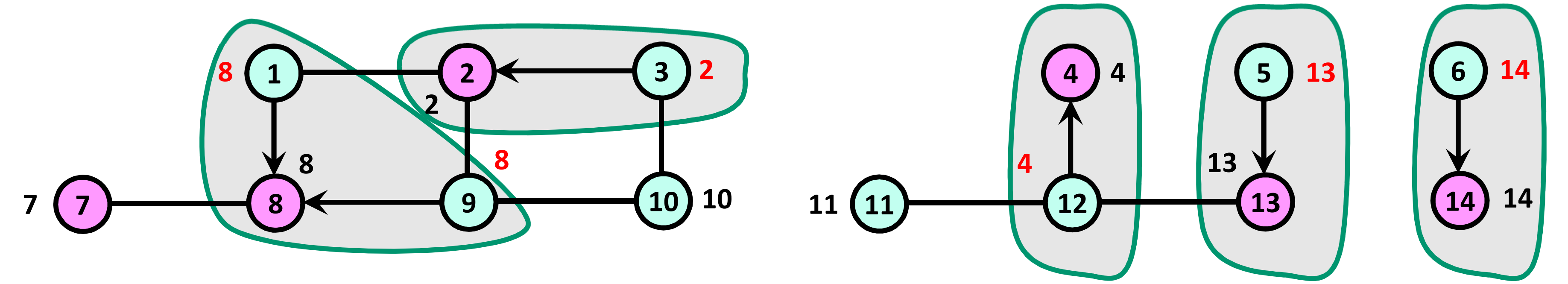
Randomized Parallel Connected Components

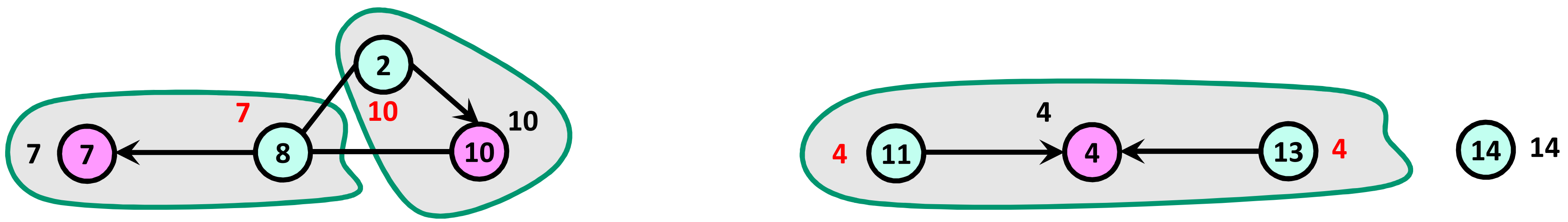


Randomized Parallel Connected Components

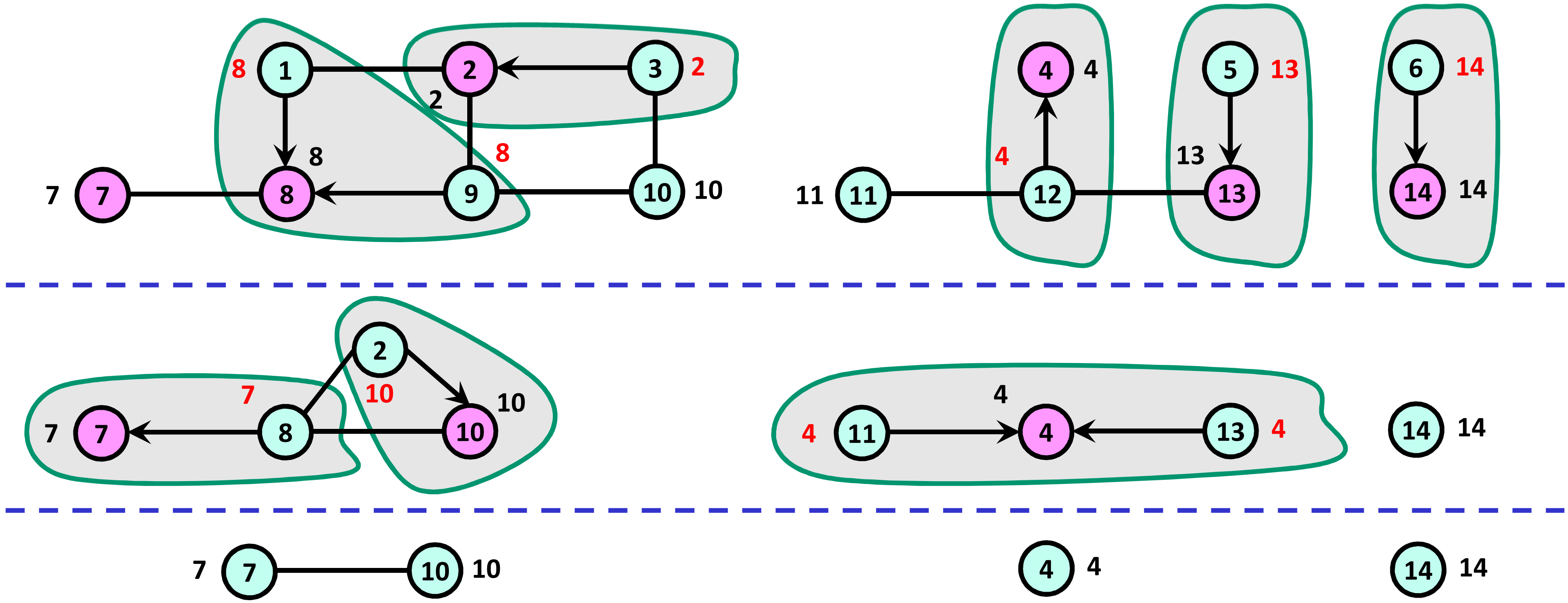


Randomized Parallel Connected Components

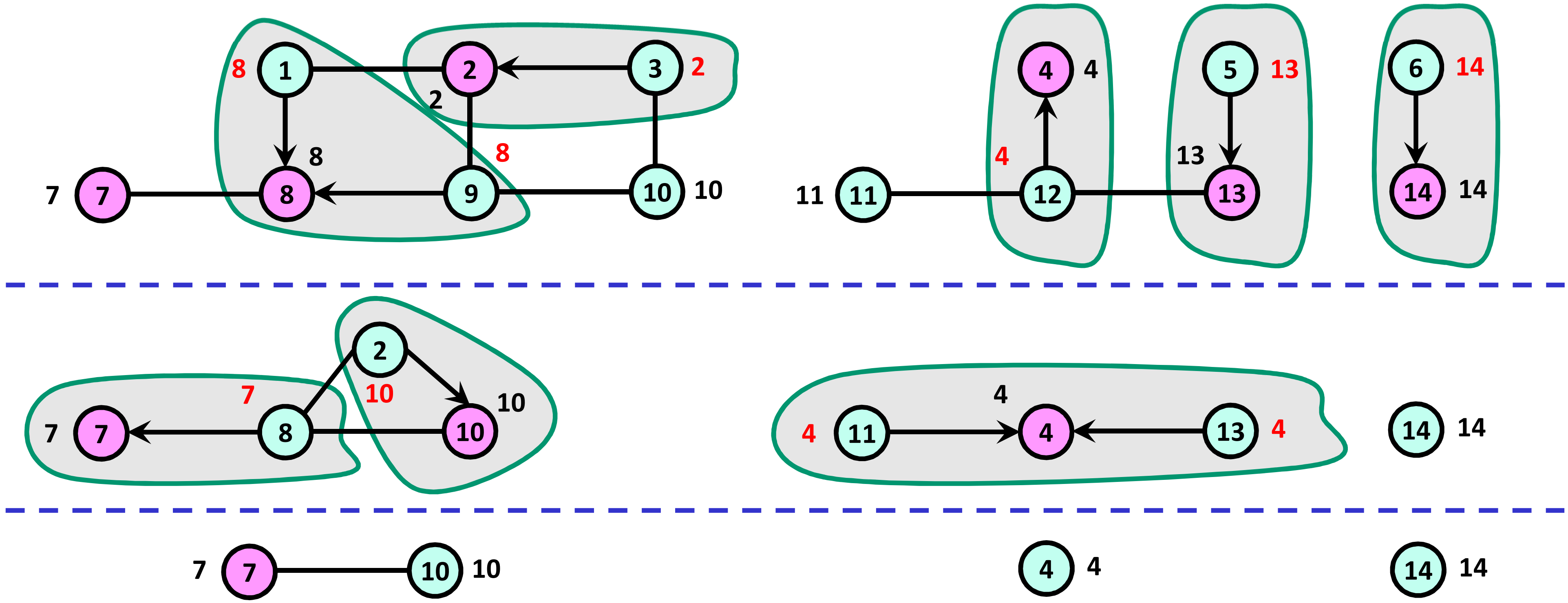




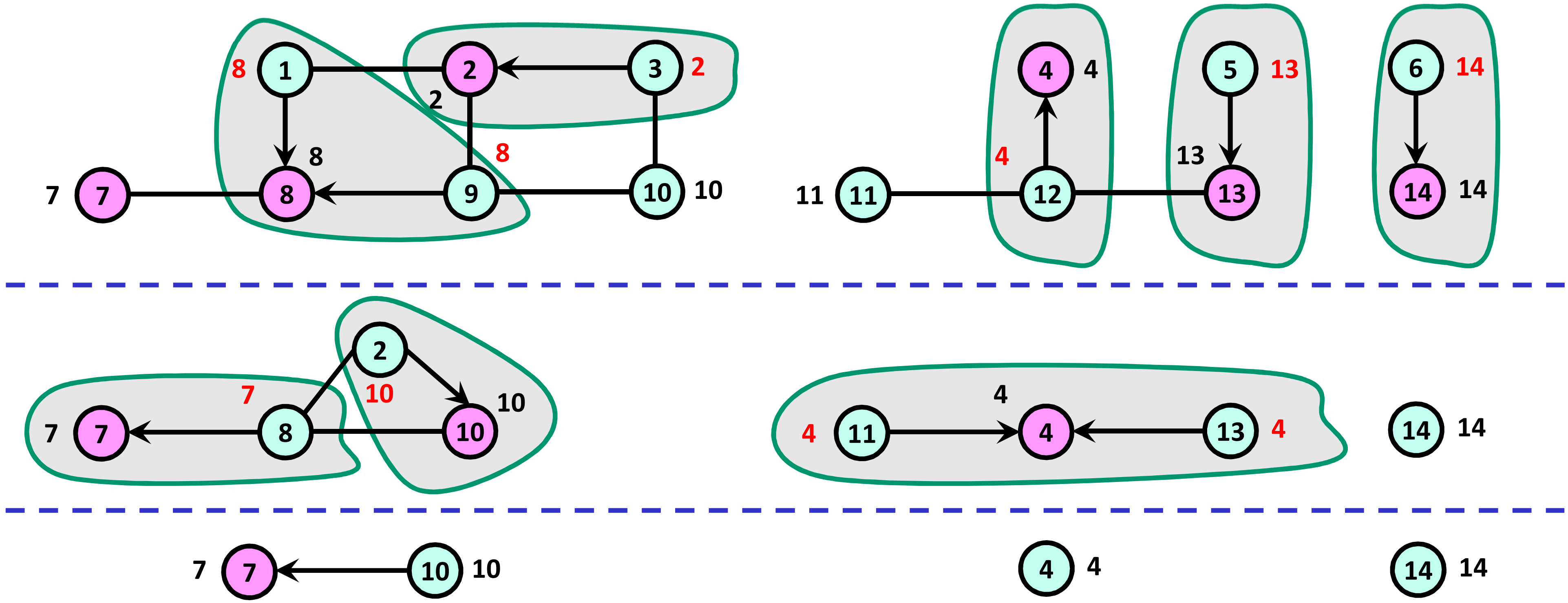
Randomized Parallel Connected Components



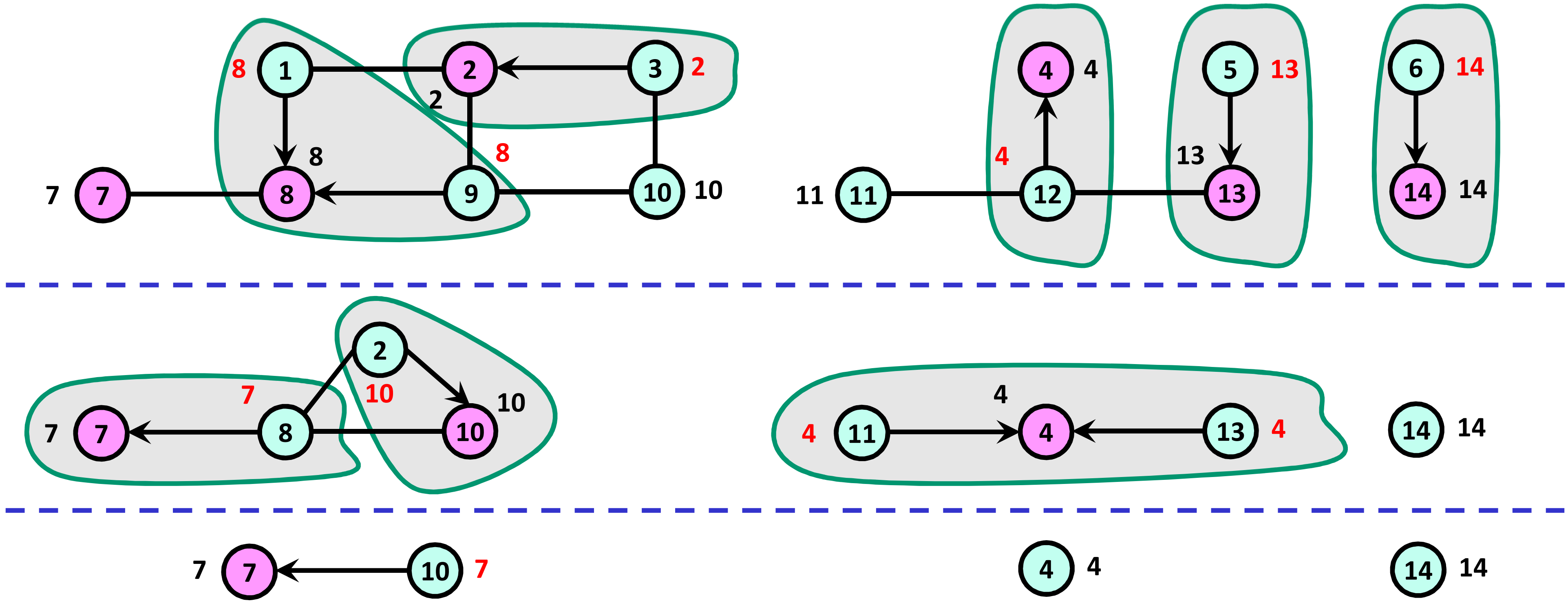
Randomized Parallel Connected Components



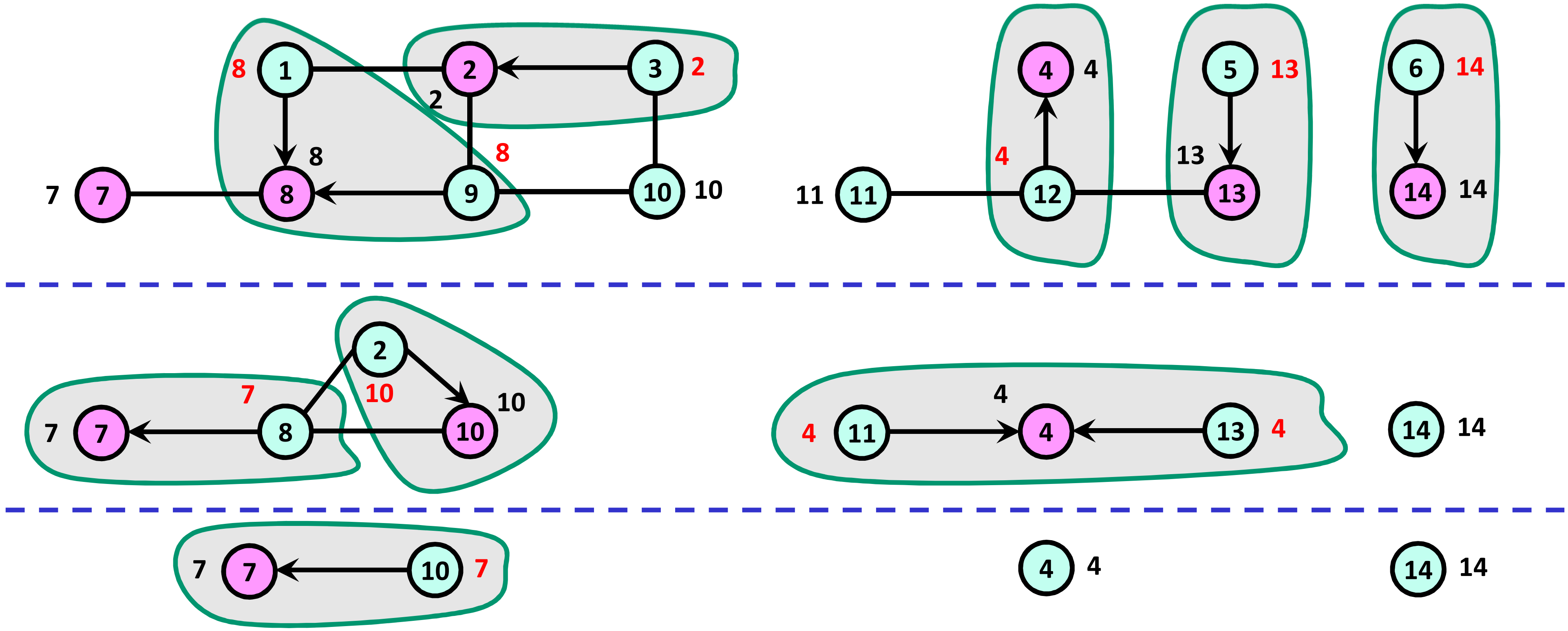
Randomized Parallel Connected Components



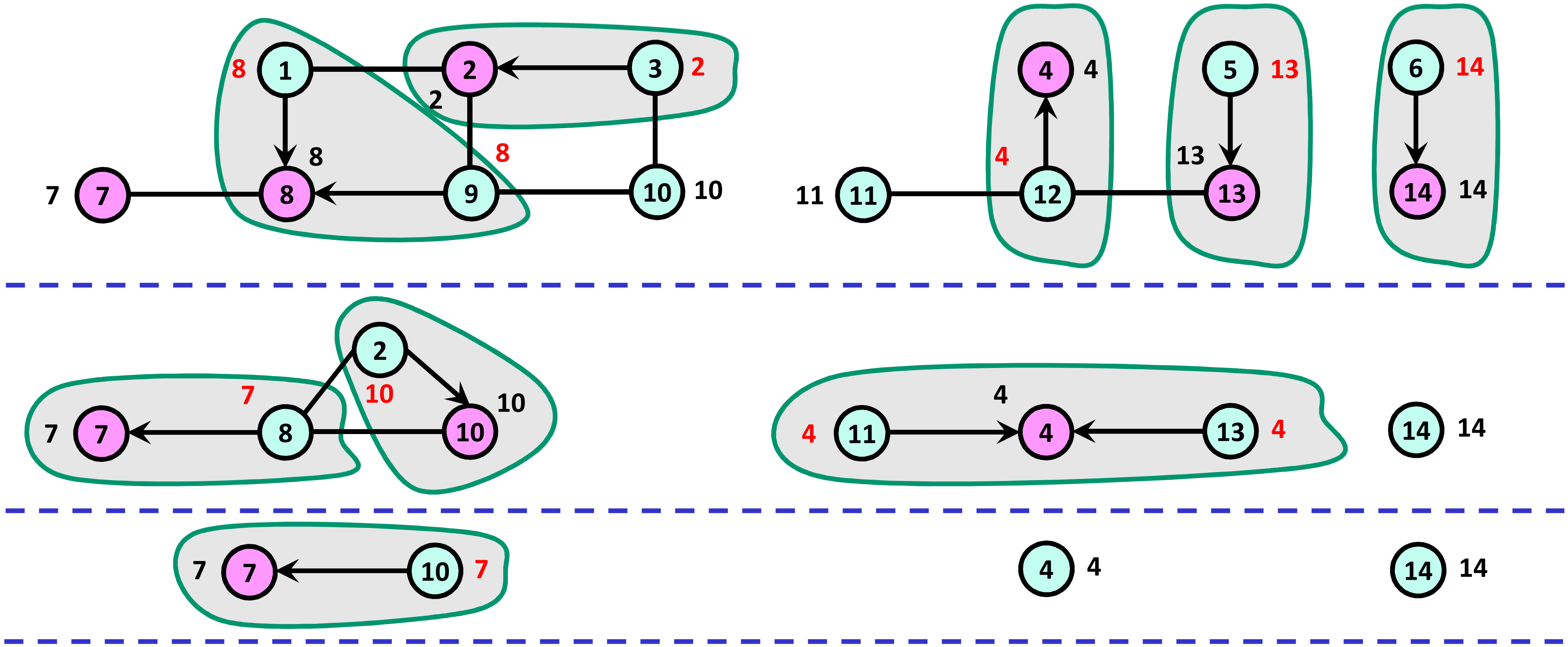
Randomized Parallel Connected Components



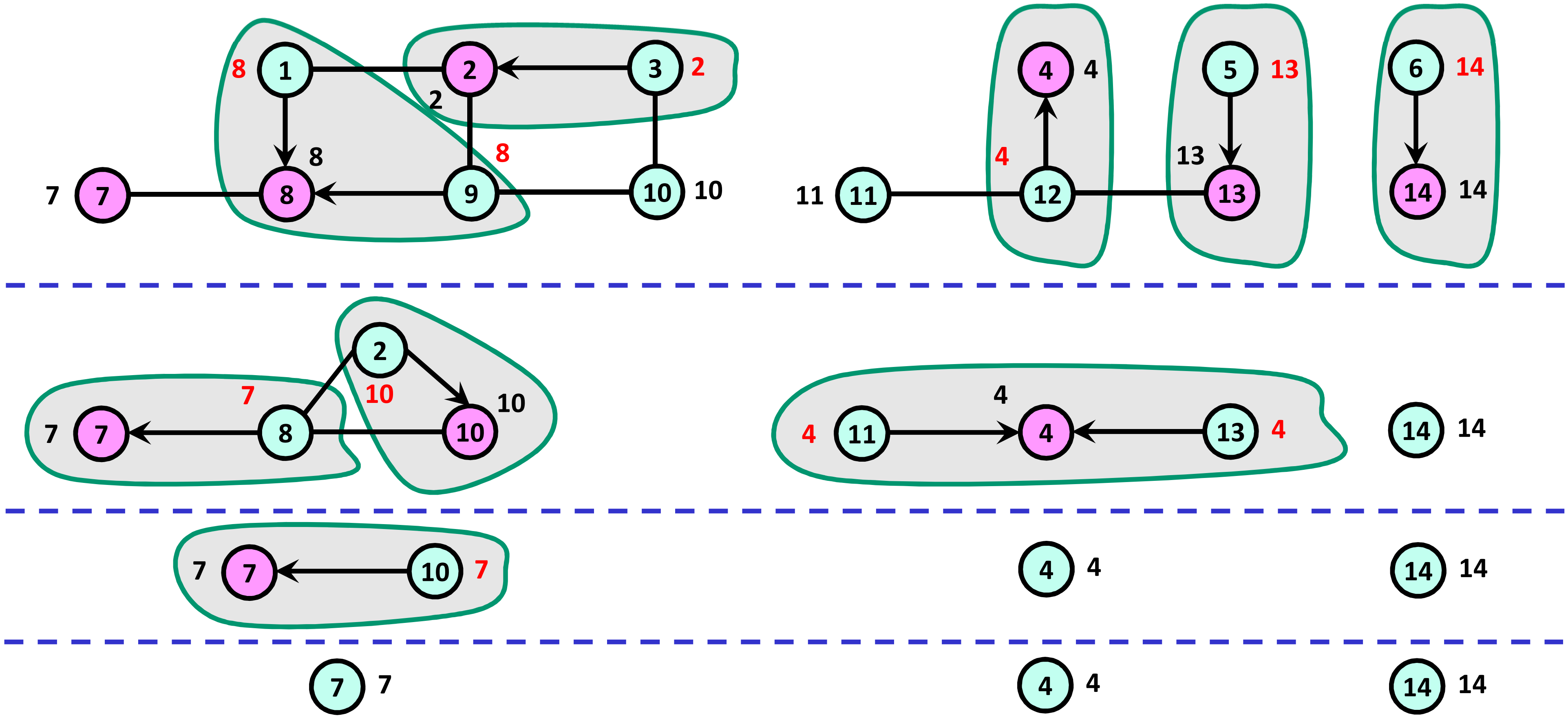
Randomized Parallel Connected Components



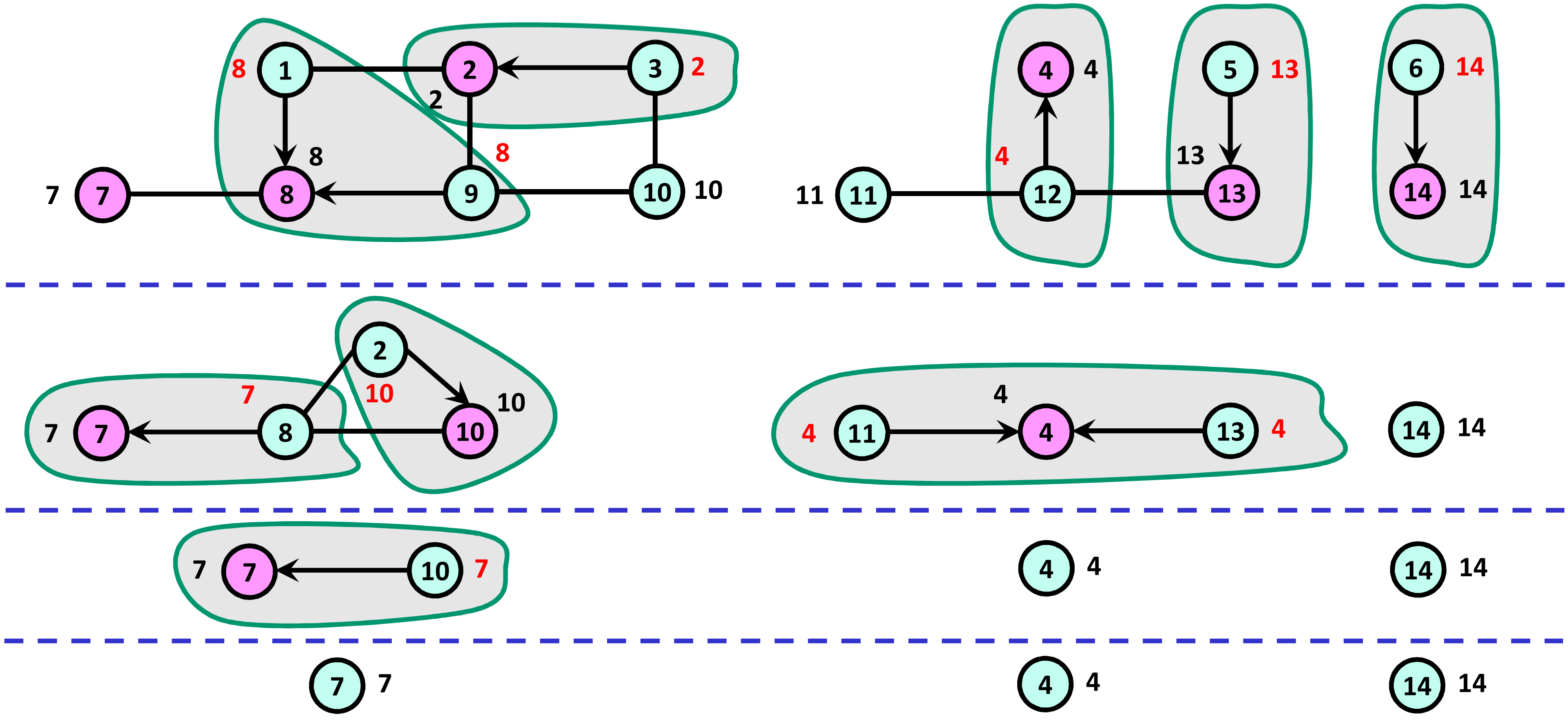
Randomized Parallel Connected Components



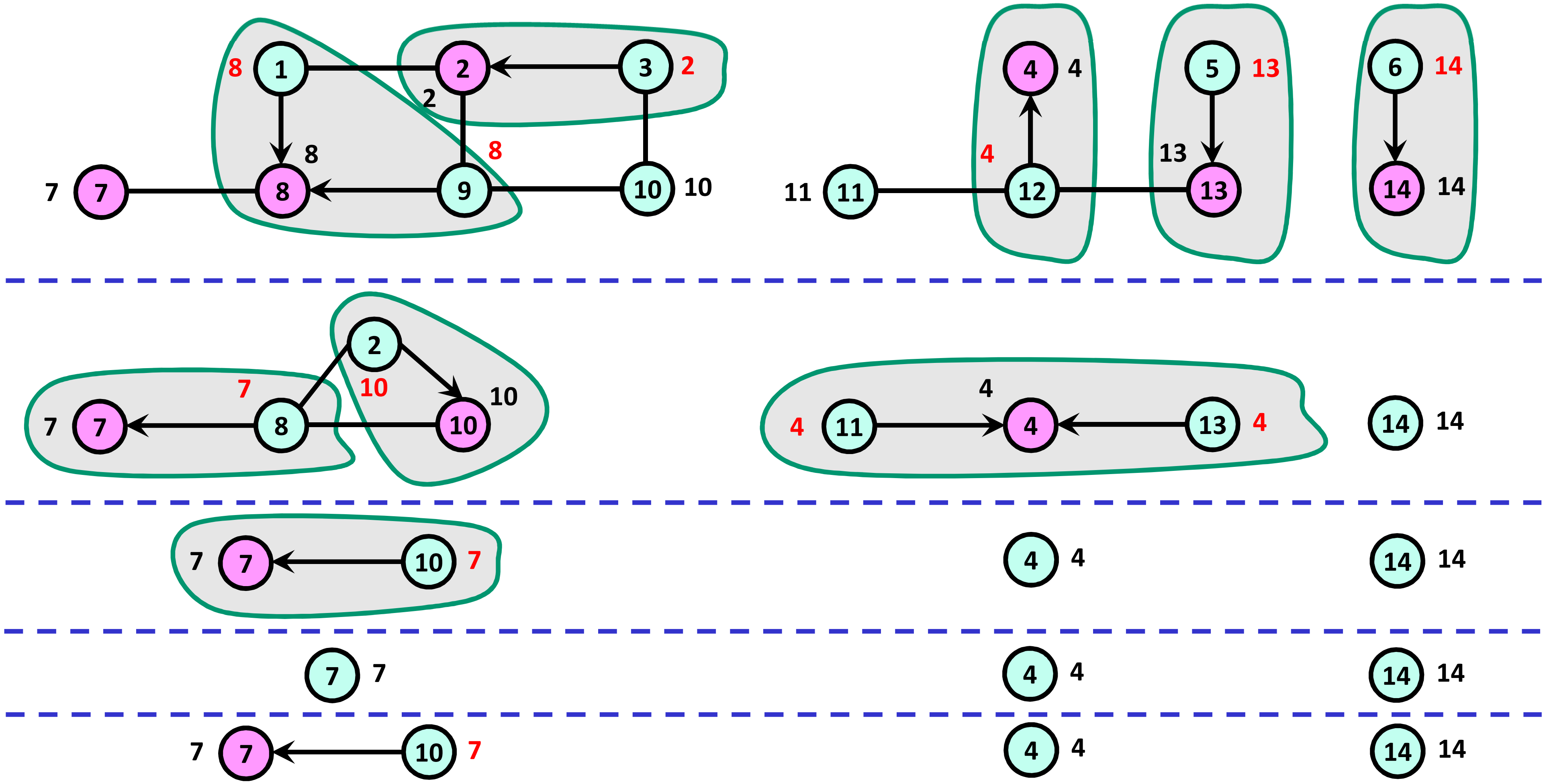
Randomized Parallel Connected Components



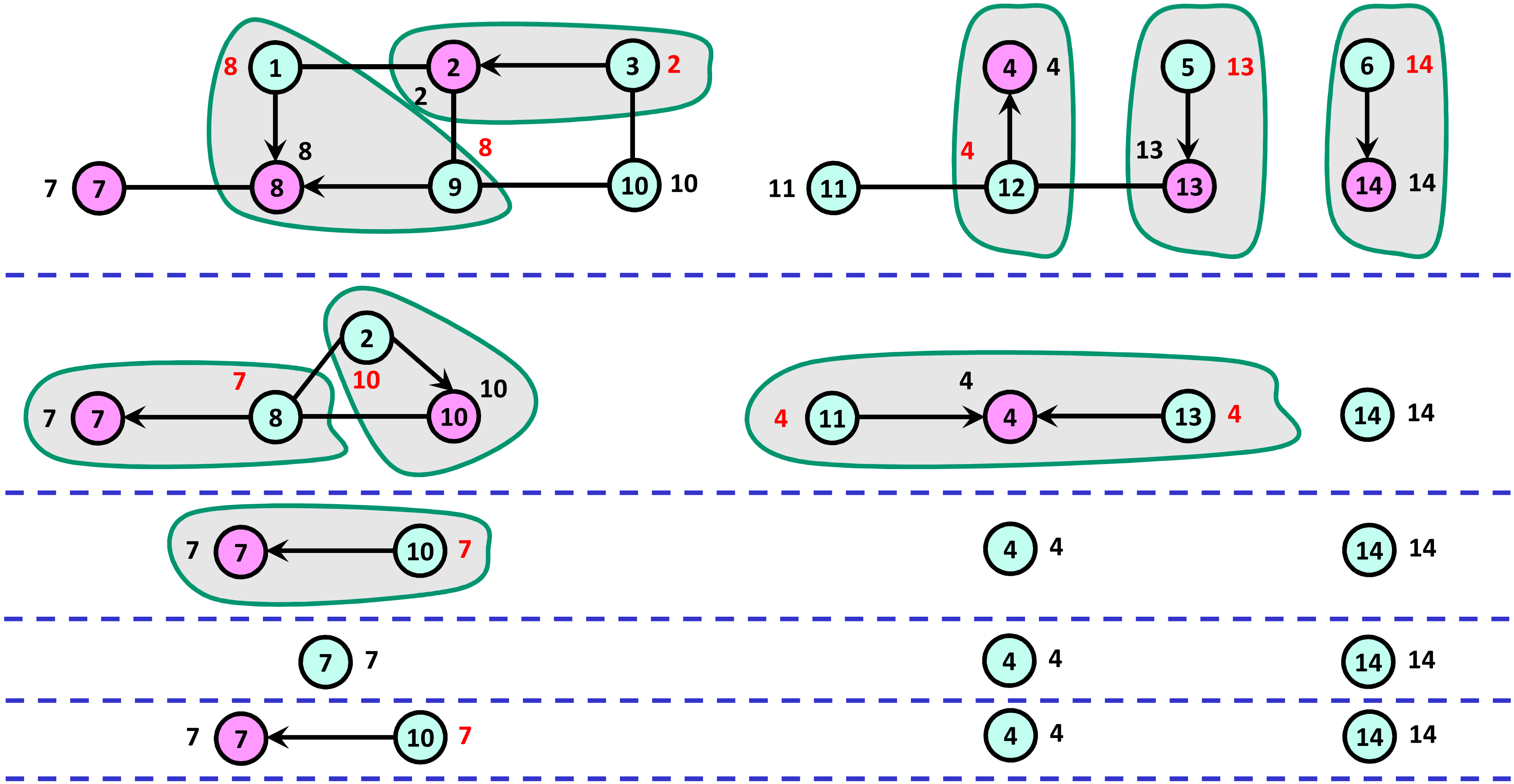
Randomized Parallel Connected Components



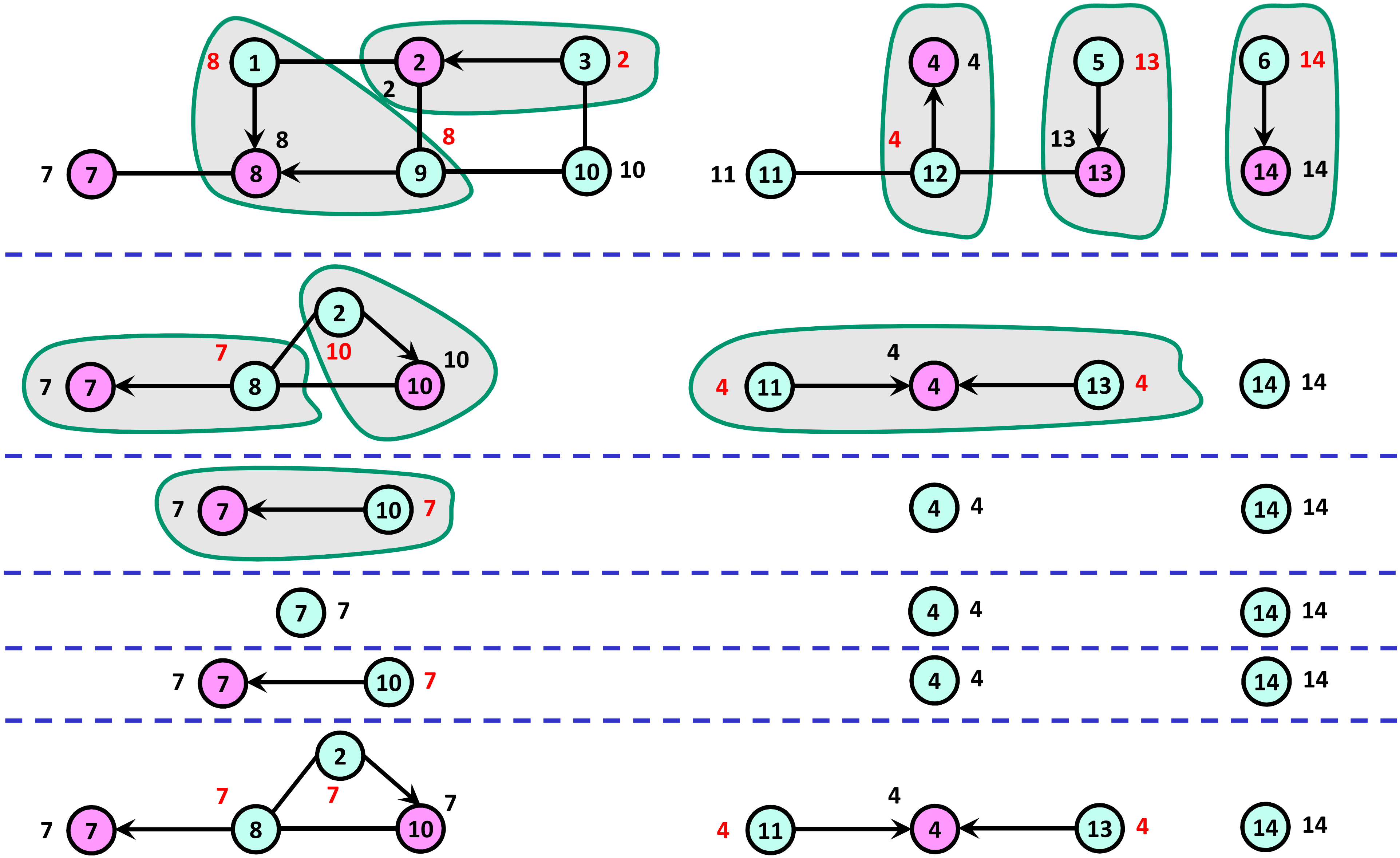
Randomized Parallel Connected Components



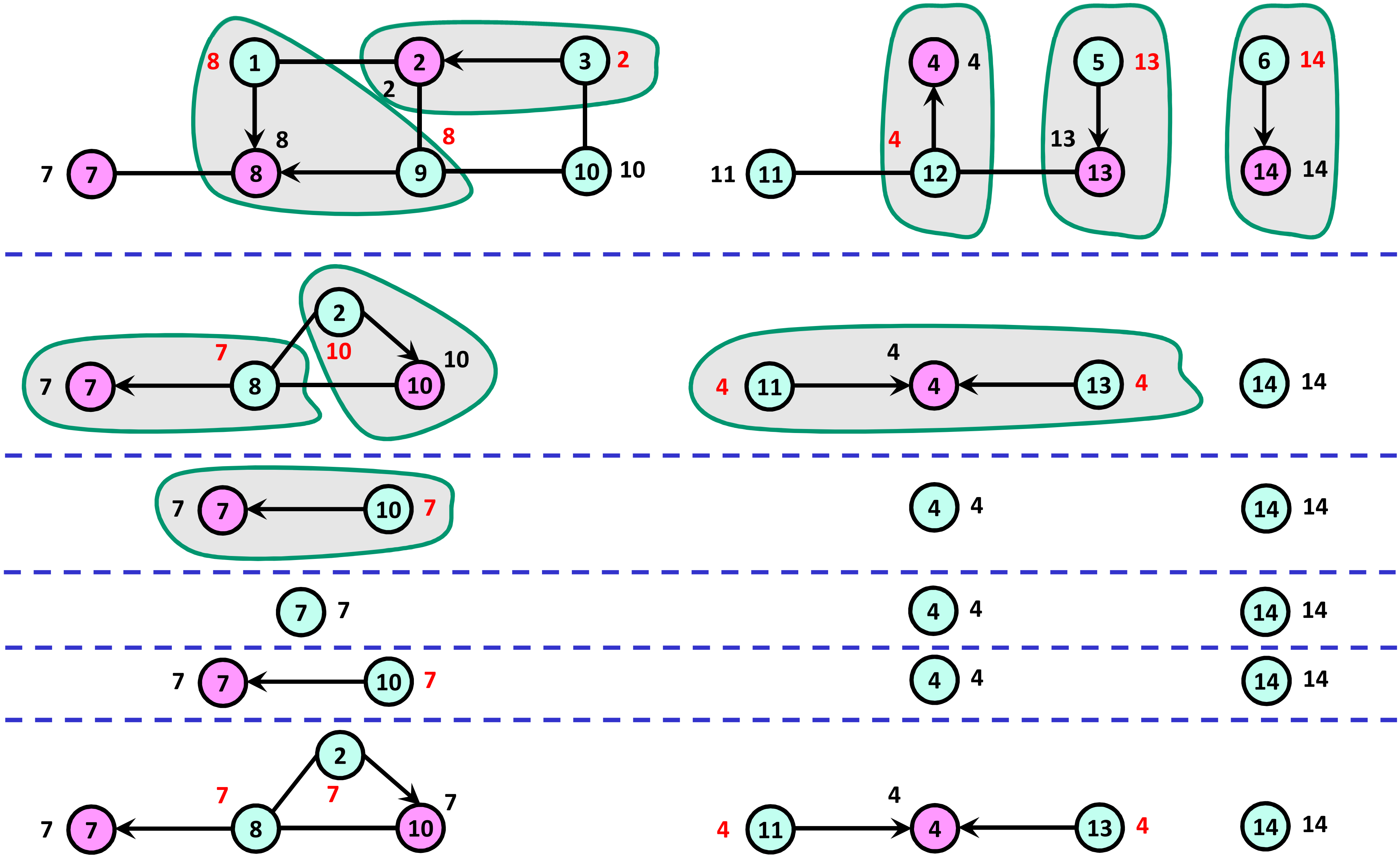
Randomized Parallel Connected Components



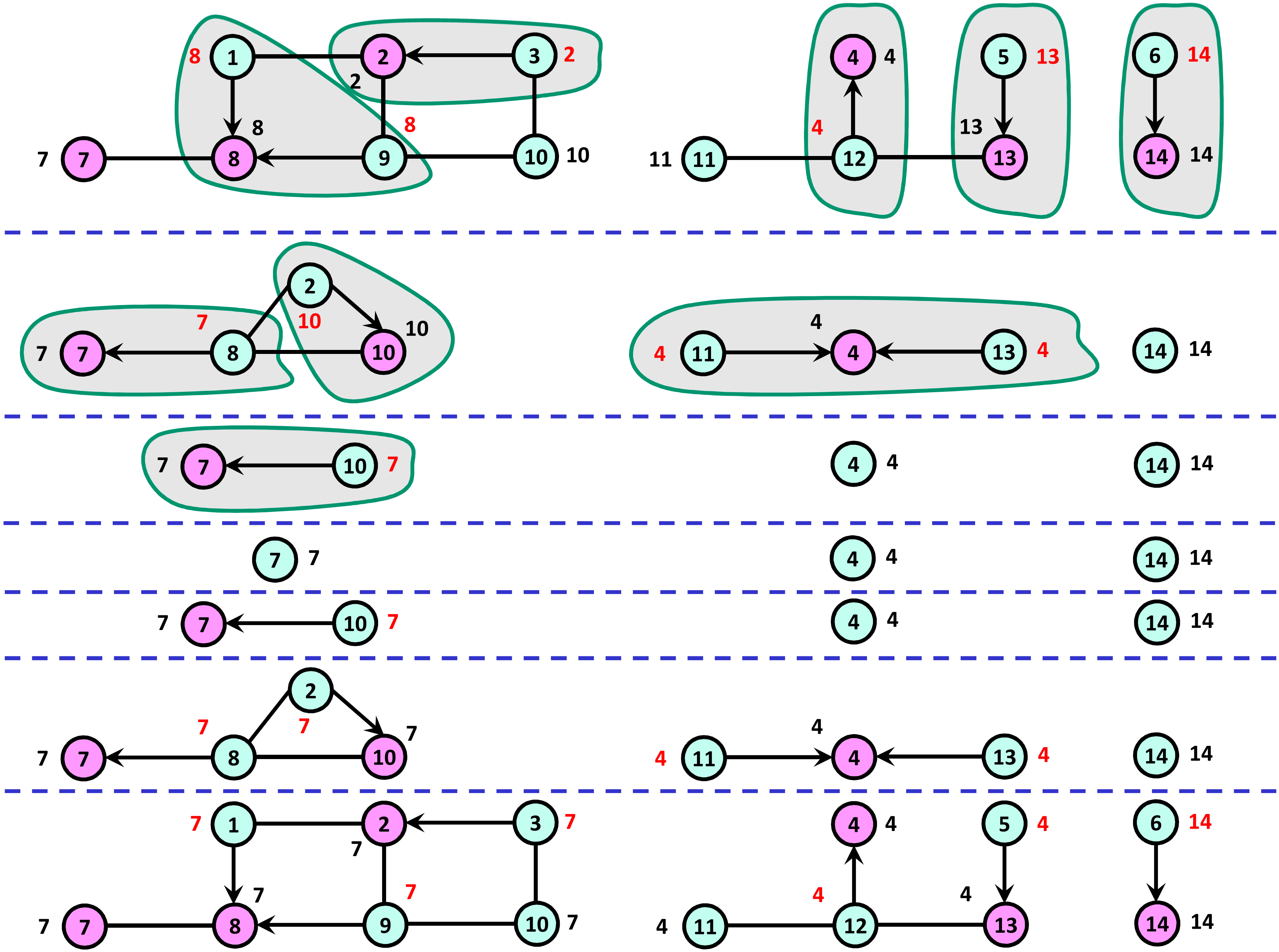
Randomized Parallel Connected Components



Randomized Parallel Connected Components



Randomized Parallel Connected Components



Randomized Parallel Connected Components (CC)

Input: *n* is the number of vertices in the graph numbered from 1 to *n*, *E* is the set of edges, and *L*[1 : *n*] are vertex labels with *L*[*v*] = *v* initially for all *v*.

Output: An array *M*[1: *n*] where for all *v*, *M*[*v*] is the unique id of the connected component containing *v*.

```
Par-Randomized-CC ( n, E, L )
if |E| = 0 then return L
array C[ 1 : n ], M[ 1 : n ], S[ 1 : |E| ]
parallel for v ← 1 to n do C[ v ] ← RANDOM{ Head, Tail }
parallel for each ( u, v ) ∈ E do
    if C[ u ] = Tail and C[ v ] = Head then L[ u ] ← L[ v ]
parallel for i ← 1 to |E| do
    if L[ E[ i ].u ] ≠ L[ E[ i ].v ] then S[ i ] ← 1 else S[ i ] ← 0
S ← Par-Prefix-Sum ( S, + )
array F[ 1 : S[ |E| ] ]
parallel for i ← 1 to |E| do
    if L[ E[ i ].u ] ≠ L[ E[ i ].v ] then
        F[ S[ i ] ] ← ( L[ E[ i ].u ], L[ E[ i ].v ] )
M ← Par-Randomized-CC ( n, F, L )
parallel for each ( u, v ) ∈ E do
    if v = L[ u ] then M[ u ] ← M[ v ]
return M
```

unbiased coin toss at each vertex

group: hook child to a parent (race!)

prepare to remove intra-group edges

find the rank of each inter-group edge among all such edges

copy the inter-group edges to *F*

find CC in the contracted graph

Map results back to the original graph

Randomized Parallel Connected Components (CC)

```
Par-Randomized-CC ( n, E, L )
   if |E| = 0 then return L
   array C[ 1 : n ], M[ 1 : n ], S[ 1 : |E| ]
   parallel for v ← 1 to n do
       C[ v ] ← RANDOM{ Head, Tail }
   parallel for each ( u, v ) ∈ E do
       if C[ u ] = Tail and C[ v ] = Head then L[ u ] ← L[ v ]
   parallel for i ← 1 to |E| do
       if L[ E[ i ].u ] ≠ L[ E[ i ].v ] then S[ i ] ← 1
       else S[ i ] ← 0
   S ← Par-Prefix-Sum ( S, + )
   array F[ 1 : S[ |E| ] ]
   parallel for i ← 1 to |E| do
       if L[ E[ i ].u ] ≠ L[ E[ i ].v ] then
           F[ S[ i ] ] ← ( L[ E[ i ].u ], L[ E[ i ].v ] )
   M ← Par-Randomized-CC ( n, F, L )
   parallel for each ( u, v ) ∈ E do
       if v = L[ u ] then M[ u ] ← M[ v ]
   return M
```

Suppose n is the number of vertices and m is the number of edges in the original graph.

Each contraction is expected to reduce #vertices of $+ve$ degree by a factor $\geq \frac{1}{4}$. [why?]

So, the expected number of contraction steps, $D = O(\log n)$. [show: the bound holds w.h.p.]

For each contraction step span is $\Theta(\log^2 n)$, and work is $\Theta(n + m)$. [why?]

Work: $T_1(n, m) = \Theta\big(D(n + m)\big)$

$= O\big((n + m)\log n\big)$ (w.h.p.)

Span: $T_\infty(n, m) = \Theta(D \log^2 n)$

$= O(\log^3 n)$ (w.h.p.)

Parallelism: $\frac{T_1(n,m)}{T_\infty(n,m)} = \Theta\left(\frac{n+m}{\log^2 n}\right)$

Pointer Jumping

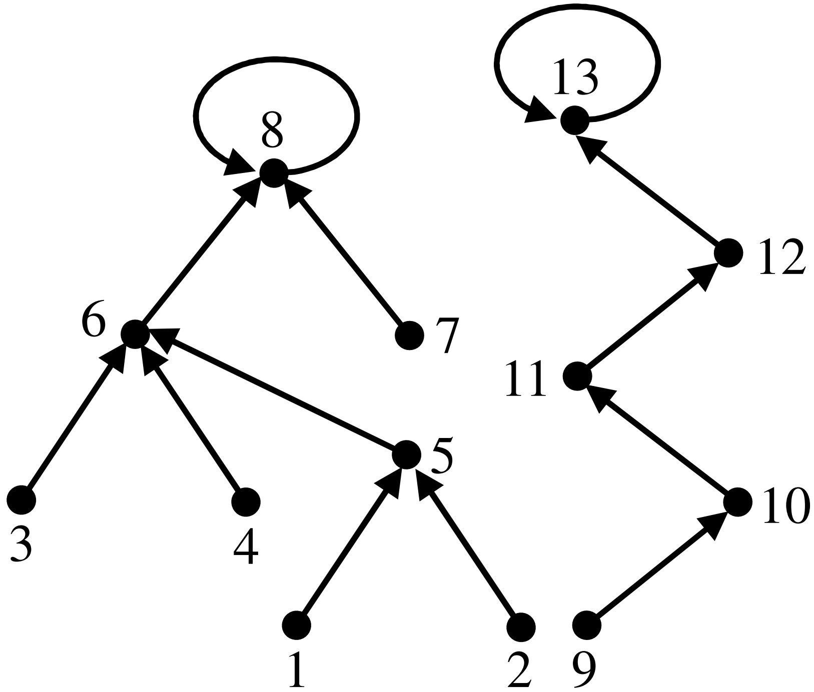
The *pointer jumping* (or *path doubling*) technique allows fast processing of data stored in the form of a set of rooted directed trees.

For every node v in the set pointer jumping involves replacing $v \rightarrow next$ with $v \rightarrow next \rightarrow next$ at every step.

Some Applications

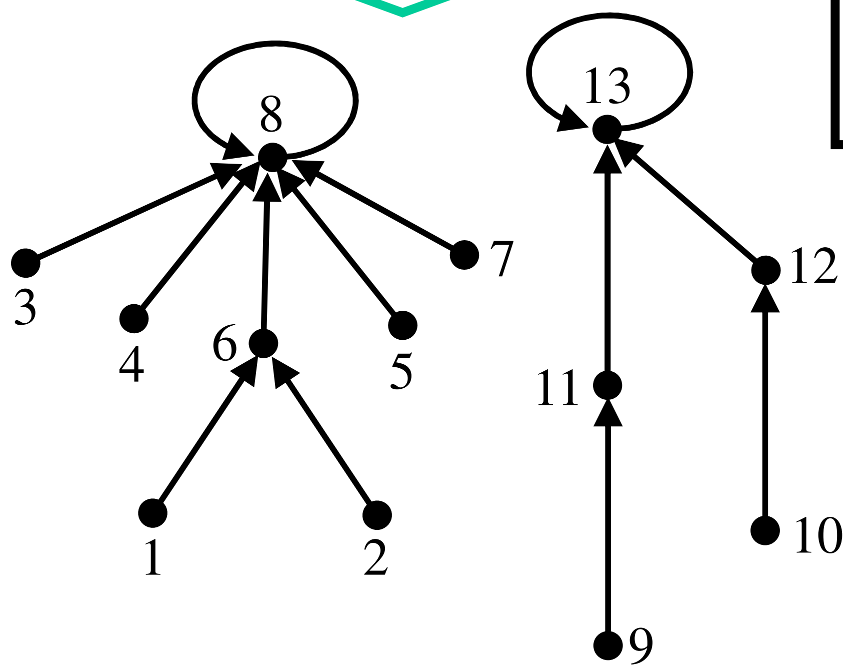
- Finding the roots of a forest of directed trees
- Parallel prefix on rooted directed trees
- List ranking

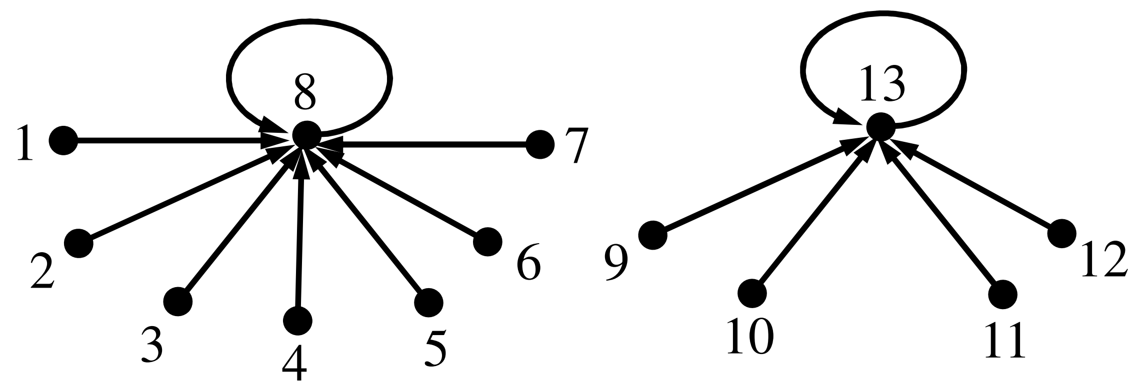
Pointer Jumping: Roots of a Forest of Directed Trees



Find-Roots (n, P, S) { **Input:** *A forest of rooted directed trees, each with a self-loop at its root, such that each edge is specified by* $\langle v, P(v) \rangle$ *for* $1 \le v \le n$. **Output:** *For each* v, *the root* $S(v)$ *of the tree containing* v. }

parallel for $v \leftarrow 1$ to n do
    $S(v) \leftarrow P(v)$
$flag \leftarrow true$
while $flag = true$ do
    $flag \leftarrow false$
    parallel for $v \leftarrow 1$ to n do
        $S(v) \leftarrow S(S(v))$
        if $S(v) \neq S(S(v))$ then $flag \leftarrow true$





Pointer Jumping: Roots of a Forest of Directed Trees

Let h be the maximum height of any tree in the forest.

Observe that the distance between v and $S(v)$ doubles after each iteration until $S\left(S(v)\right)$ is the root of the tree containing v.

```
Find-Roots ( n, P, S )   { Input: A forest of rooted directed trees, each
                           with a self-loop at its root, such that each
                           edge is specified by ⟨v, P(v)⟩ for 1 ≤ v ≤ n.
                           Output: For each v, the root S(v) of the tree
                           containing v. }

parallel for v ← 1 to n do
    S(v) ← P(v)
flag ← true
while flag = true do
    flag ← false
    parallel for v ← 1 to n do
        S(v) ← S(S(v))
        if S(v) ≠ S(S(v)) then flag ← true
```

Hence, the number of iterations is $\log h$. Thus (assuming that each parallel for loop takes $\Theta(1)$ time to execute),

Work: $T_1(n) = O(n \log h)$ and **Span:** $T_\infty(n) = \Theta(\log h)$

Parallelism: $\frac{T_1(n)}{T_\infty(n)} = O(n)$

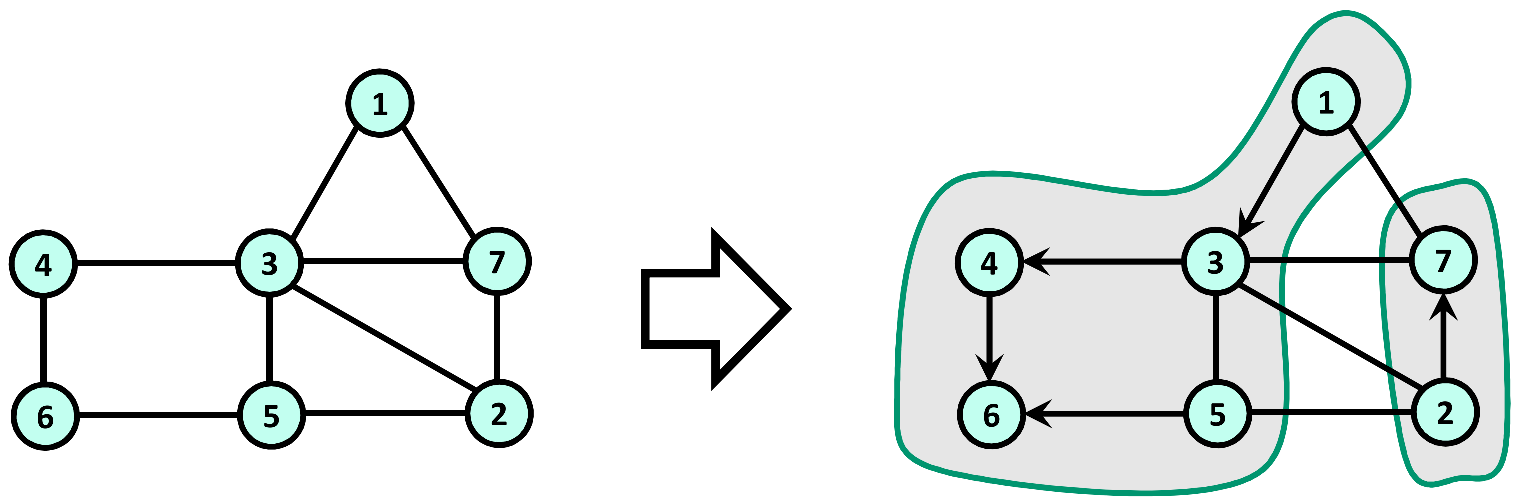
Deterministic Parallel Connected Components (CC)

Approach

- Form a set of disjoint subtrees
- Use pointer-jumping to reduce each subtree to a single vertex
- Recursively apply the same trick on the contracted graph

Forming Disjoint Subtrees

- Hook each vertex to a neighbor with larger label (if exists)
- Ensures that no cycles are formed

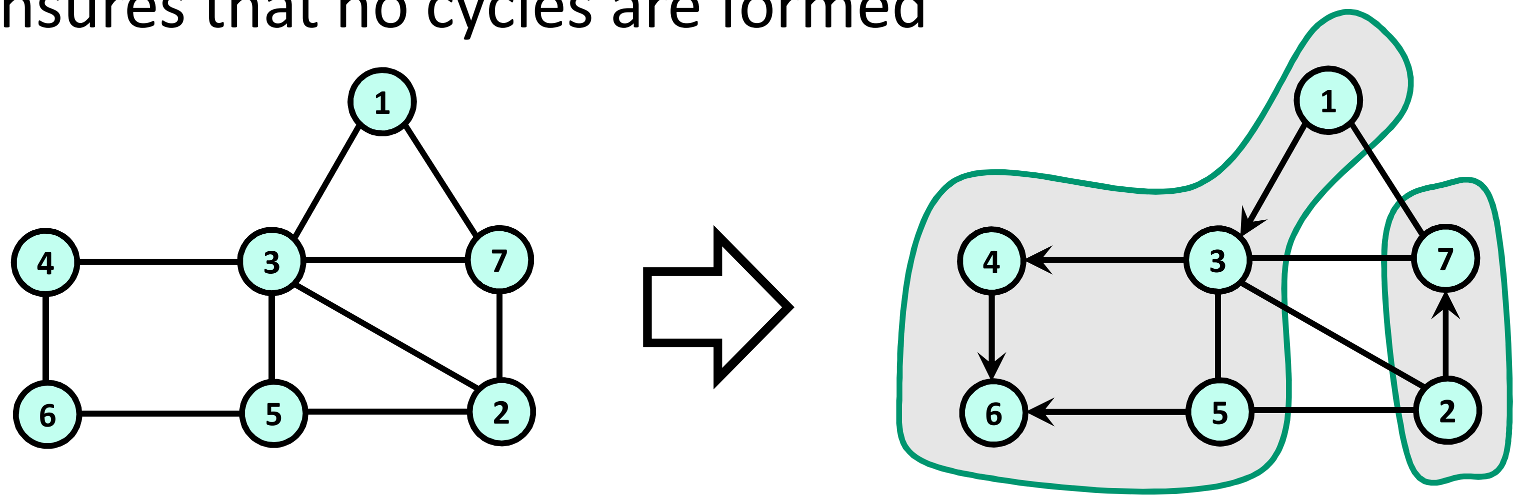


Deterministic Parallel Connected Components (CC)

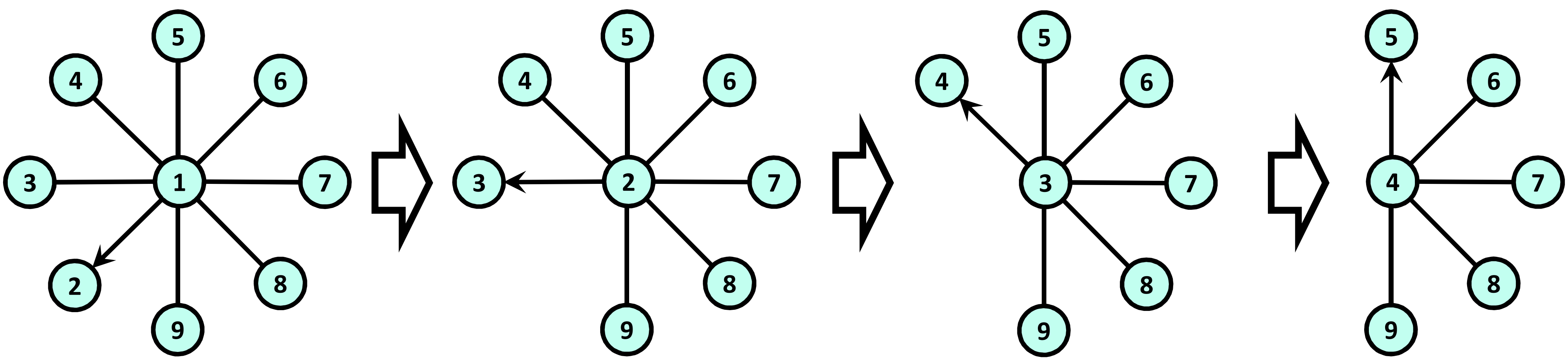
Forming Disjoint Subtrees

– Hook each vertex to a neighbor with larger label (if exists)

– Ensures that no cycles are formed



– But the number of contraction steps can be as large as $n - 1$!



Deterministic Parallel Connected Components (CC)

Observation:

Let $G = (V, E)$ be an undirected graph with n vertices in which each vertex has at least one neighbor. Then

$$\text{either } |\{u|(u,v) \in E \wedge (u < v)\}| \geq \frac{n}{2}$$
$$\text{or } |\{u|(u,v) \in E \wedge (u > v)\}| \geq \frac{n}{2}$$

Implication:

Between the two directions of hooking (i.e., smaller to larger label, and larger to smaller label) always choose the one that hooks more vertices.

Then in each contraction step the number of vertices will be reduced by a factor of at least $\frac{1}{2}$.

Deterministic Parallel Connected Components (CC)

Input: *n* is the number of vertices in the graph numbered from 1 to *n*, *E* is the set of edges, and *L*[1 : *n*] are vertex labels with *L*[*v*] = *v* initially for all *v*.

Output: Updated array *L*[1: *n*] where for all *v*, *L*[*v*] is the unique id of the connected component containing *v*.

```
Par-Deterministic-CC ( n, E, L )
if |E| = 0 then return L
array l2h[ 1 : n ], h2l[ 1 : n ], S[ 1 : |E| ]
parallel for v ← 1 to n do l2h[ v ] ← 0, h2l[ v ] ← 0
parallel for each ( u, v ) ∈ E do
    if u < v then l2h[ u ] ← 1 else h2l[ u ] ← 1
n_1 ← Par-Sum ( l2h, + ), n_2 ← Par-Sum ( h2l, + )
parallel for each ( u, v ) ∈ E do
    if n_1 ≥ n_2 and u < v then L[ u ] ← v
    else if n_1 < n_2 and u > v then L[ u ] ← v
Find-Roots ( n, L, L )
parallel for i ← 1 to |E| do S[ i ] ← ( L[ E[ i ].u ] ≠ L[ E[ i ].v ] ) ? 1 : 0
S ← Par-Prefix-Sum ( S, + )
array F[ 1 : S[ |E| ] ]
parallel for i ← 1 to |E| do
    if L[ E[ i ].u ] ≠ L[ E[ i ].v ] then F[ S[ i ] ] ← ( L[ E[ i ].u ], L[ E[ i ].v ] )
L ← Par-Deterministic-CC ( n, F, L )
return L
```

count hooks from smaller to larger indices, and vice versa

use pointer jumping to label each vertex with the id of its root

mark hooks from smaller to larger indices

mark hooks from larger to smaller indices

choose hook direction to maximize #hooks

similar to *Par-Randomized-CC*, except that relabeling is not needed after the recursive call

Deterministic Parallel Connected Components (CC)

```
Par-Deterministic-CC ( n, E, L )
 1. if |E| = 0 then return L
 2. array l2h[ 1 : n ], h2l[ 1 : n ], S[ 1 : |E| ]
 3. parallel for v ← 1 to n do
        l2h[ v ] ← 0, h2l[ v ] ← 0
 4. parallel for each ( u, v ) ∈ E do
 5.     if u < v then l2h[ u ] ← 1 else h2l[ u ] ← 1
 6. n1 ← Par-Sum ( l2h, + ), n2 ← Par-Sum ( h2l, + )
 7. parallel for each ( u, v ) ∈ E do
 8.     if n1 ≥ n2 and u < v then L[ u ] ← v
 9.     else if n1 < n2 and u > v then L[ u ] ← v
10. Find-Roots ( n, L, L )
11. parallel for i ← 1 to |E| do
        S[ i ] ← ( L[ E[ i ].u ] ≠ L[ E[ i ].v ] ) ? 1 : 0
12. S ← Par-Prefix-Sum ( S, + )
13. array F[ 1 : S[ |E| ] ]
14. parallel for i ← 1 to |E| do
15.     if L[ E[ i ].u ] ≠ L[ E[ i ].v ] then
            F[ S[ i ] ] ← ( L[ E[ i ].u ], L[ E[ i ].v ] )
16. L ← Par-Deterministic-CC ( n, F, L )
17. return L
```

Each contraction step reduces the number of vertices by a factor of at least $\frac{1}{2}$.

So, number of contraction steps, $D = O(\log n)$.

For contraction step $k \geq 0$ span is $O(\log^2 n)$, and work is $O(n \log n + m)$. [why?]

Work: $T_1(n, m) = O(\sum_{0 \leq i < D} (n \log n + m))$

$$= O((n \log n + m) D)$$

$$= O\big((n \log n + m) \log n\big)$$

Span: $T_\infty(n, m) = O(D \log^2 n)$

$$= O(\log^3 n)$$

How to get $T_1(n, m) = O\big((n + m) \log n\ \big)$?