# Part III

# PRAM Algorithms

# Prefix Sum

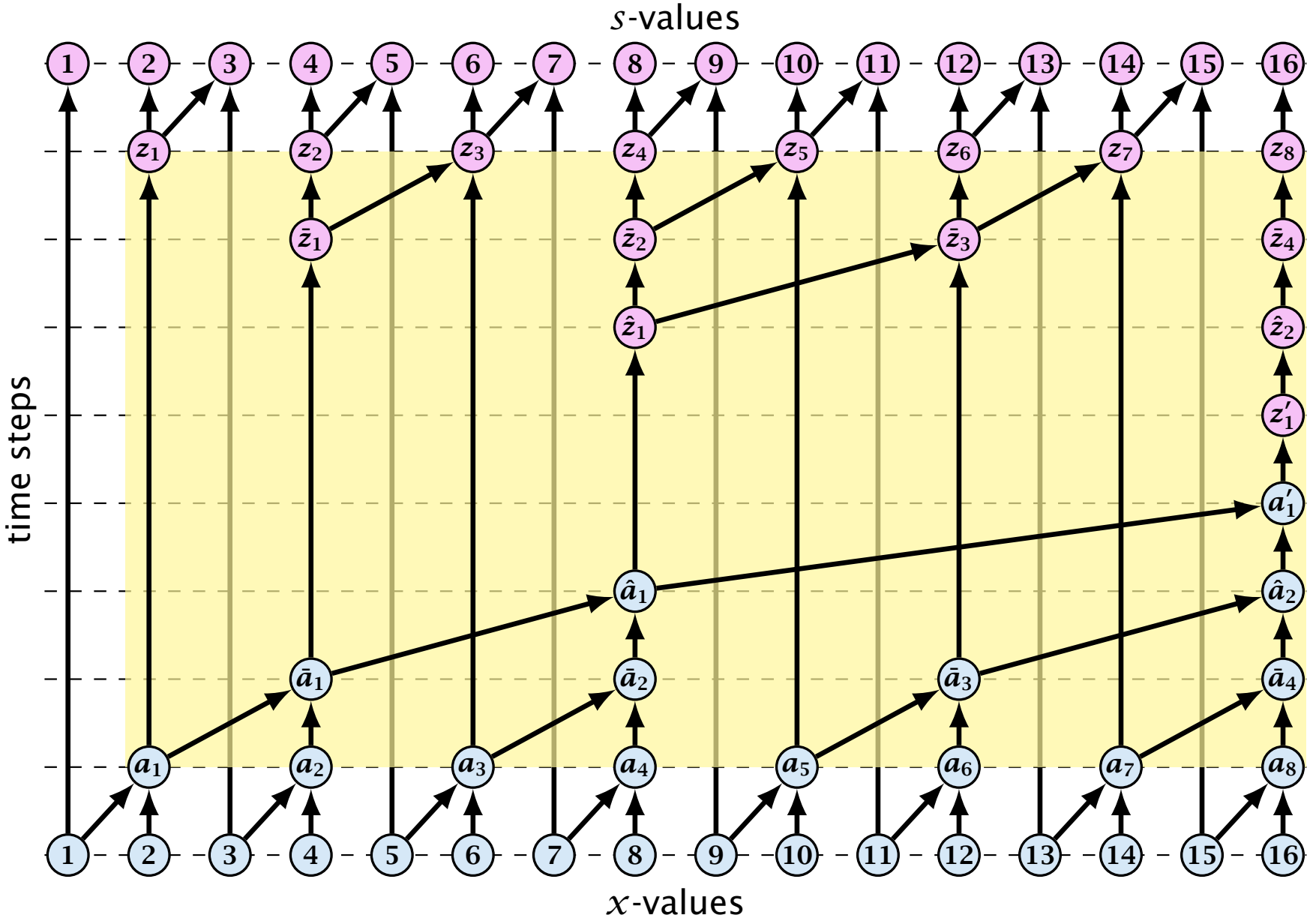
input: $x[1]\ldots x[n]$

output: $s[1]\ldots s[n]$ with $s[i] = \sum_{j=1}^{i} x[i]$ (w.r.t. operator $*$)

**Algorithm 6** PrefixSum$(n, x[1]\ldots x[n])$

```
1: // compute prefixsums; n = 2^k
2: if n = 1 then s[1] ← x[1]; return
3: for 1 ≤ i ≤ n/2 pardo
4:         a[i] ← x[2i − 1] ∗ x[2i]
5: z[1], ..., z[n/2] ← PrefixSum(n/2, a[1]...a[n/2])
6: for 1 ≤ i ≤ n pardo
7:         i even   : s[i] ← z[i/2]
8:         i = 1    : s[1] = x[1]
9:         i odd    : s[i] ← z[(i − 1)/2] ∗ x[i]
```

# Prefix Sum


s-values
time steps
x-values
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
$z_1$ $z_2$ $z_3$ $z_4$ $z_5$ $z_6$ $z_7$ $z_8$
$\bar{z}_1$ $\bar{z}_2$ $\bar{z}_3$ $\bar{z}_4$
$\hat{z}_1$ $\hat{z}_2$
$z'_1$
$a'_1$
$\hat{a}_1$ $\hat{a}_2$
$\bar{a}_1$ $\bar{a}_2$ $\bar{a}_3$ $\bar{a}_4$
$a_1$ $a_2$ $a_3$ $a_4$ $a_5$ $a_6$ $a_7$ $a_8$
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

# Prefix Sum

The algorithm uses work $\mathcal{O}(n)$ and time $\mathcal{O}(\log n)$ for solving Prefix Sum on an EREW-PRAM with $n$ processors.

It is clearly work-optimal.

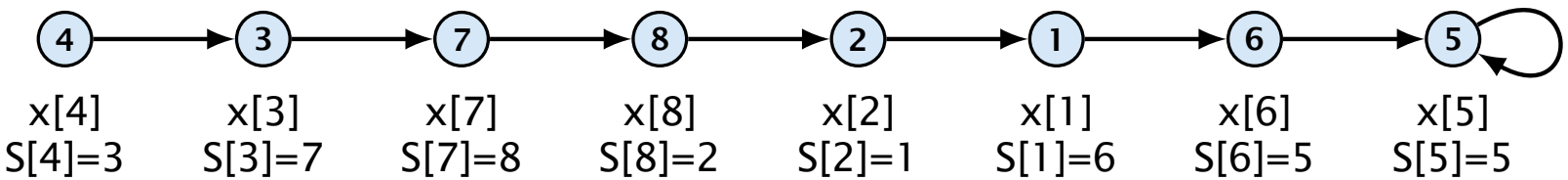
**Theorem 1**

*On a CREW PRAM a Prefix Sum requires running time $\Omega(\log n)$ regardless of the number of processors.*

# Parallel Prefix

**Input**: a linked list given by successor pointers; a value $x[i]$ for every list element; an operator $*$;

**Output**: for every list position $\ell$ the sum (w.r.t. $*$) of elements after $\ell$ in the list (including $\ell$)



# Parallel Prefix

**Algorithm 7** ParallelPrefix

```
1: for 1 ≤ i ≤ n pardo
2:         P[i] ← S[i]
3:         while S[i] ≠ S[S[i]] do
4:                 x[i] ← x[i] * x[S[i]]
5:                 S[i] ← S[S[i]]
6:         if P[i] ≠ i then S[i] ← x[S(i)]
```

The algorithm runs in time $\mathcal{O}(\log n)$.

It has work requirement $\mathcal{O}(n \log n)$. non-optimal

This technique is also known as pointer jumping

# 4.3 Divide & Conquer — Merging

Given two sorted sequences $A = (a_1, \ldots, a_n)$ and $B = (b_1, \ldots, b_n)$, compute the sorted squence $C = (c_1, \ldots, c_n)$.

**Definition 2**

Let $X = (x_1, \ldots, x_t)$ be a sequence. The rank $\mathrm{rank}(y : X)$ of $y$ in $X$ is

$$\mathrm{rank}(y : X) = |\{x \in X \mid x \leq y\}|$$

For a sequence $Y = (y_1, \ldots, y_s)$ we define $\mathrm{rank}(Y : X) := (r_1, \ldots, r_s)$ with $r_i = \mathrm{rank}(y_i : X)$.

# 4.3 Divide & Conquer — Merging

Given two sorted sequences $A = (a_1 \dots a_n)$ and $B = (b_1 \dots b_n)$, compute the sorted squence $C = (c_1 \dots c_n)$.

**Observation:**
We can assume wlog. that elements in $A$ and $B$ are different.

Then for $c_i \in C$ we have $i = \text{rank}(c_i : A \cup B)$.

**This means we just need to determine $\text{rank}(x : A \cup B)$ for all elements!**

Observe, that $\text{rank}(x : A \cup B) = \text{rank}(x : A) + \text{rank}(x : B)$.

Clearly, for $x \in A$ we already know $\text{rank}(x : A)$, and for $x \in B$ we know $\text{rank}(x : B)$.

# 4.3 Divide & Conquer — Merging

Compute $\mathrm{rank}(x : A)$ for all $x \in B$ and $\mathrm{rank}(x : B)$ for all $x \in A$.
can be done in $\mathcal{O}(\log n)$ time with $2n$ processors by binary search

**Lemma 3**
*On a CREW PRAM, Merging can be done in $\mathcal{O}(\log n)$ time and $\mathcal{O}(n \log n)$ work.*

**not optimal**

# 4.3 Divide & Conquer — Merging

$A = (a_1, \ldots, a_n)$; $B = (b_1, \ldots, b_n)$;
$\log n$ integral; $k := n/\log n$ integral;

**Algorithm 8** GenerateSubproblems

1: $j_0 \leftarrow 0$
2: $j_k \leftarrow n$
3: **for** $1 \leq i \leq k-1$ **pardo**
4: $\quad j_i \leftarrow \text{rank}(b_{i \log n} : A)$
5: **for** $0 \leq i \leq k-1$ **pardo**
6: $\quad B_i \leftarrow (b_{i \log n+1}, \ldots, b_{(i+1)\log n})$
7: $\quad A_i \leftarrow (a_{j_i+1}, \ldots, a_{j_{i+1}})$

If $C_i$ is the merging of $A_i$ and $B_i$ then the sequence $C_0 \ldots C_{k-1}$ is a sorted sequence.

# 4.3 Divide & Conquer — Merging

We can generate the subproblems in time $\mathcal{O}(\log n)$ and work $\mathcal{O}(n)$.

Note that in a sub-problem $B_i$ has length $\log n$.

If we run the algorithm again for every subproblem, (where $A_i$ takes the role of $B$) we can in time $\mathcal{O}(\log\log n)$ and work $\mathcal{O}(n)$ generate subproblems where $A_j$ and $B_j$ have both length at most $\log n$.

Such a subproblem can be solved by a single processor in time $\mathcal{O}(\log n)$ and work $\mathcal{O}(|A_i| + |B_i|)$.

Parallelizing the last step gives total work $\mathcal{O}(n)$ and time $\mathcal{O}(\log n)$.

**the resulting algorithm is work optimal**

# 4.4 Maximum Computation

**Lemma 4**

*On a CRCW PRAM the maximum of $n$ numbers can be computed in time $\mathcal{O}(1)$ with $n^2$ processors.*

proof on board...

# 4.4 Maximum Computation

**Lemma 5**

*On a CRCW PRAM the maximum of $n$ numbers can be computed in time $\mathcal{O}(\log\log n)$ with $n$ processors and work $\mathcal{O}(n \log\log n)$.*

proof on board...

# 4.4 Maximum Computation

**Lemma 6**

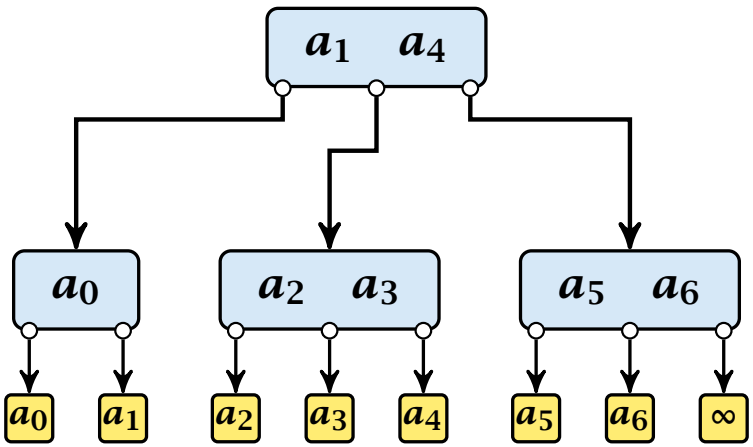
*On a CRCW PRAM the maximum of $n$ numbers can be computed in time $\mathcal{O}(\log\log n)$ with $n$ processors and work $\mathcal{O}(n)$.*

proof on board...

# 4.5 Inserting into a $(2,3)$-tree

Given a $(2,3)$-tree with $n$ elements, and a sequence $x_0 < x_1 < x_2 < \cdots < x_k$ of elements. We want to insert elements $x_1, \ldots, x_k$ into the tree ($k \ll n$).

**time: $\mathcal{O}(\log n)$; work: $\mathcal{O}(k \log n)$**

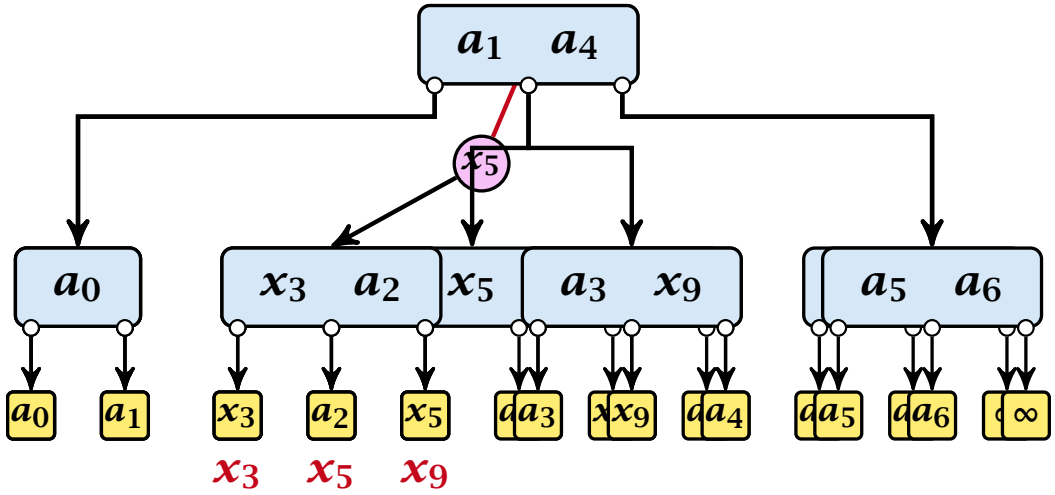


# 4.5 Inserting into a $(2,3)$-tree

1. determine for every $x_i$ the leaf element before which it has to be inserted
   time: $\mathcal{O}(\log n)$; work: $\mathcal{O}(k \log n)$; CREW PRAM

   all $x_i$'s that have to be inserted before the same element form a chain
2. determine the largest/smallest/middle element of every chain
   time: $\mathcal{O}(1)$; work: $\mathcal{O}(k)$;
3. insert the middle element of every chain
   compute new chains
   time: $\mathcal{O}(\log n)$; work: $\mathcal{O}(k_i \log n)$; $k_i$= #inserted elements
   (computing new chains is constant time)
4. repeat Step 3 for logarithmically many rounds
   time: $\mathcal{O}(\log n \log k)$; work: $\mathcal{O}(k \log n)$;

# Step 3




- each internal node is split into at most two parts
- each split operation promotes at most one element
- hence, on every level we want to insert at most one element per successor pointer
- we can use the same routine for every level

# 4.5 Inserting into a $(2,3)$-tree

- Step 3, works in phases; one phase for every level of the tree
- Step 4, works in rounds; in each round a different set of elements is inserted

**Observation**
We can start with phase $i$ of round $r$ as long as phase $i$ of round $r-1$ and (of course), phase $i-1$ of round $r$ has finished.

This is called Pipelining. Using this technique we can perform all rounds in Step 4 in just $\mathcal{O}(\log k + \log n)$ many parallel steps.

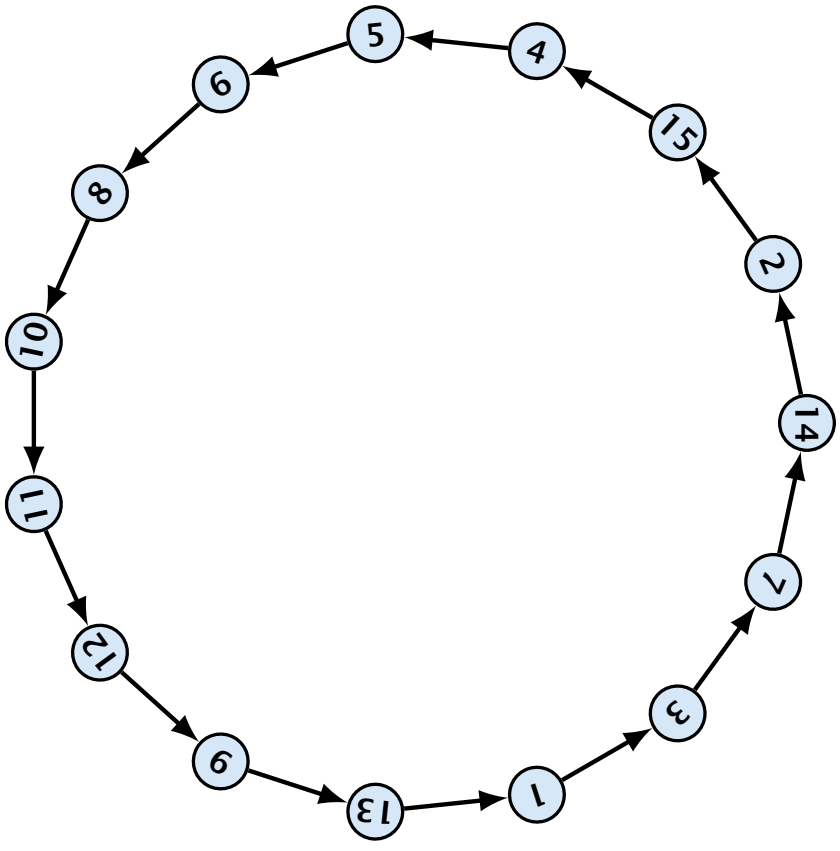
# 4.6 Symmetry Breaking

The following algorithm colors an $n$-node cycle with $\lceil \log n \rceil$ colors.

**Algorithm 9** BasicColoring

1: **for** $1 \leq i \leq n$ **pardo**
2: $\quad \mathrm{col}(i) \leftarrow i$
3: $\quad k_i \leftarrow$ smallest bitpos where $\mathrm{col}(i)$ and $\mathrm{col}(S(i))$ differ
4: $\quad \mathrm{col}'(i) \leftarrow 2k + \mathrm{col}(i)_k$

# 4.6 Symmetry Breaking




| $v$ | col | $k$ | col′ |
|---|---|---|---|
| 1 | 0001 | 1 | 2 |
| 3 | 0011 | 2 | 4 |
| 7 | 0111 | 0 | 1 |
| 14 | 1110 | 2 | 5 |
| 2 | 0010 | 0 | 0 |
| 15 | 1111 | 0 | 1 |
| 4 | 0100 | 0 | 0 |
| 5 | 0101 | 0 | 1 |
| 6 | 0110 | 1 | 3 |
| 8 | 1000 | 1 | 2 |
| 10 | 1010 | 0 | 0 |
| 11 | 1011 | 0 | 1 |
| 12 | 1100 | 0 | 0 |
| 9 | 1001 | 2 | 4 |
| 13 | 1101 | 2 | 5 |

# 4.6 Symmetry Breaking

Applying the algorithm to a coloring with bit-length $t$ generates a coloring with largest color at most

$$2(t-1)+1$$

and bit-length at most

$$\lceil \log_2(2(t-1)+1) \rceil \leq \lceil \log_2(t-1) \rceil + 1 \leq \lceil \log_2(t) \rceil + 1$$

Applying the algorithm repeatedly generates a constant number of colors after $\log^* n$ operations.

Note that the first inequality holds because $2(t-1)-1$ is odd.

# 4.6 Symmetry Breaking

As long as the bit-length $t \geq 4$ the bit-length decreases.

Applying the algorithm with bit-length 3 gives a coloring with colors in the range $0, \ldots, 5 = 2t - 1$.

We can improve to a 3-coloring by successively re-coloring nodes from a color-class:

**Algorithm 10** ReColor

```
1: for ℓ ← 5 to 3
2:     for 1 ≤ i ≤ n pardo
3:         if col(i) = ℓ then
4:             col(i) ← min{{0,1,2} \ {col(P[i]), col(S[i])}}
```

This requires time $\mathcal{O}(1)$ and work $\mathcal{O}(n)$.

# 4.6 Symmetry Breaking

**Lemma 7**

*We can color vertices in a ring with three colors in $\mathcal{O}(\log^* n)$ time and with $\mathcal{O}(n \log^* n)$ work.*

not work optimal

# 4.6 Symmetry Breaking

**Lemma 8**

*Given $n$ integers in the range $0, \ldots, \mathcal{O}(\log n)$, there is an algorithm that sorts these numbers in $\mathcal{O}(\log n)$ time using a linear number of operations.*

**Proof:** Exercise!

# 4.6 Symmetry Breaking

**Algorithm 11** OptColor

1: **for** $1 \leq i \leq n$ **pardo**
2: $\quad \mathrm{col}(i) \leftarrow i$
3: apply BasicColoring once
4: sort vertices by colors
5: **for** $\ell = 2\lceil \log n \rceil$ **to** $3$ **do**
6: $\quad$ **for** all vertices $i$ of color $\ell$ **pardo**
7: $\quad\quad \mathrm{col}(i) \leftarrow \min\{\{0,1,2\} \setminus \{\mathrm{col}(P[i]), \mathrm{col}(S[i])\}\}$

We can perform Lines 6 and 7 in time $\mathcal{O}(n_\ell)$ only because we sorted before. In general a statement like "**for** constraint **pardo**" should only contain a contraint on the id's of the processors but not something complicated (like the color) which has to be checked and, hence, induces work. Because of the sorting we can transform this complicated constraint into a constraint on just the processor id's.

**Lemma 9**

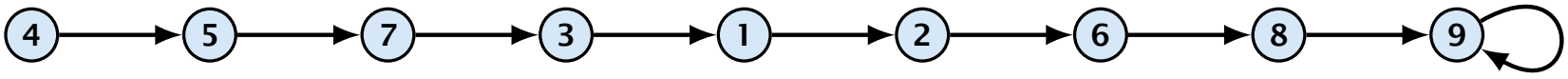
*A ring can be colored with $3$ colors in time $\mathcal{O}(\log n)$ and with work $\mathcal{O}(n)$.*

work optimal but not too fast

# List Ranking

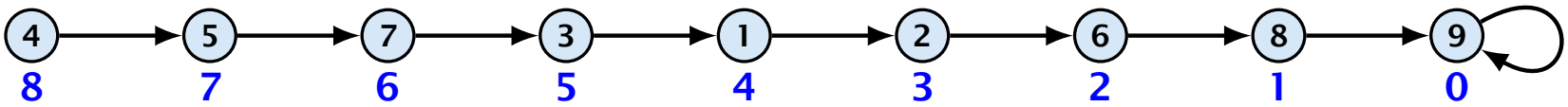
**Input:**

A list given by successor pointers;




**Output:**

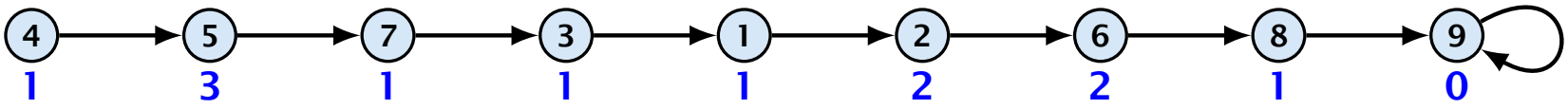
For every node number of hops to end of the list;




**Observation:**

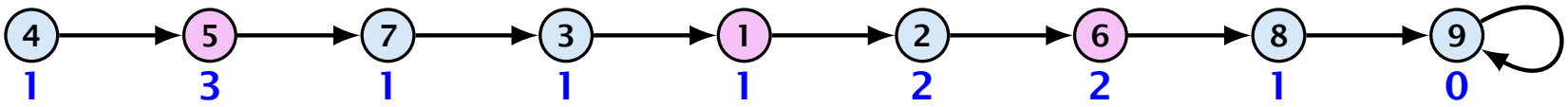
Special case of parallel prefix

# List Ranking




1. Given a list with values; perhaps from previous iterations.
   The list is given via predecessor pointers $P(i)$ and successor pointers $S(i)$.
   $S(4) = 5$, $S(2) = 6$, $P(3) = 7$, etc.

# List Ranking

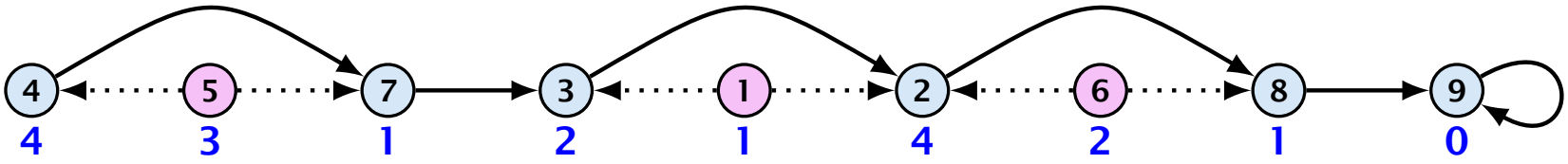



**2.** Find an independent set; time: $\mathcal{O}(\log n)$; work: $\mathcal{O}(n)$.

The independent set should contain a constant fraction of the vertices.

Color vertices; take local minima

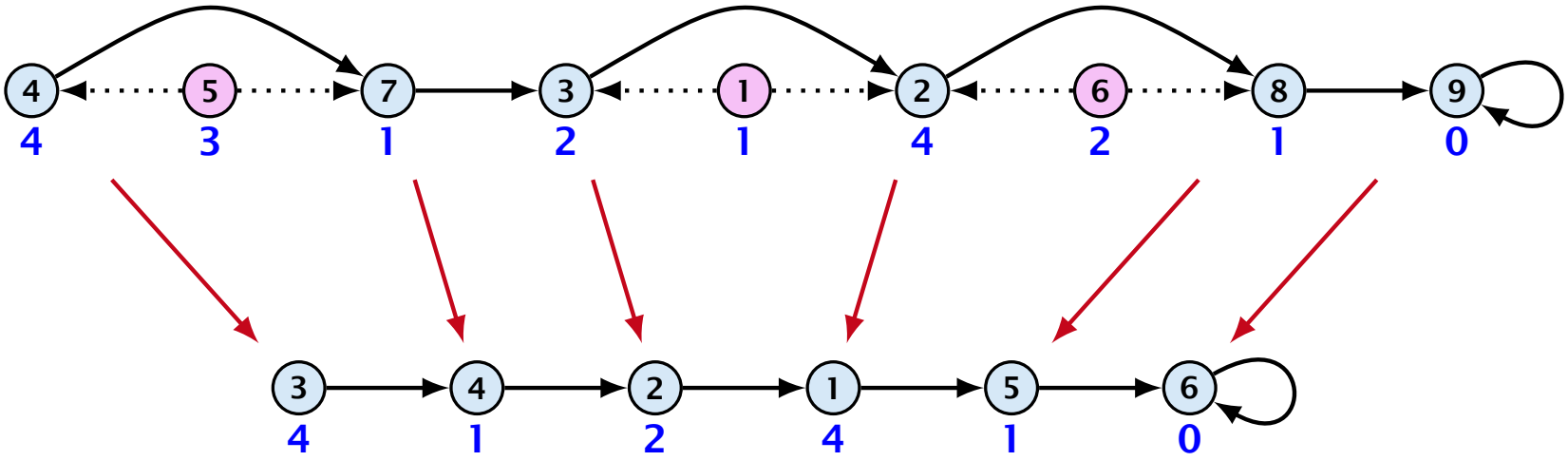
# List Ranking




3. Splice the independent set out of the list;

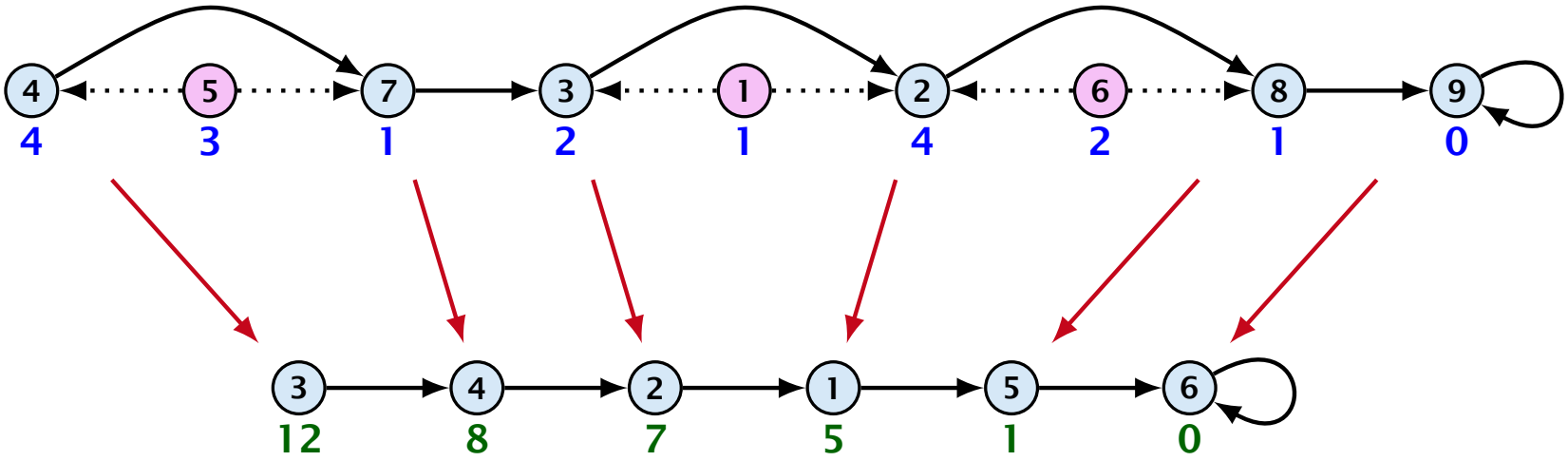
   At the independent set vertices the array still contains old values for $P(i)$ and $S(i)$;

# List Ranking



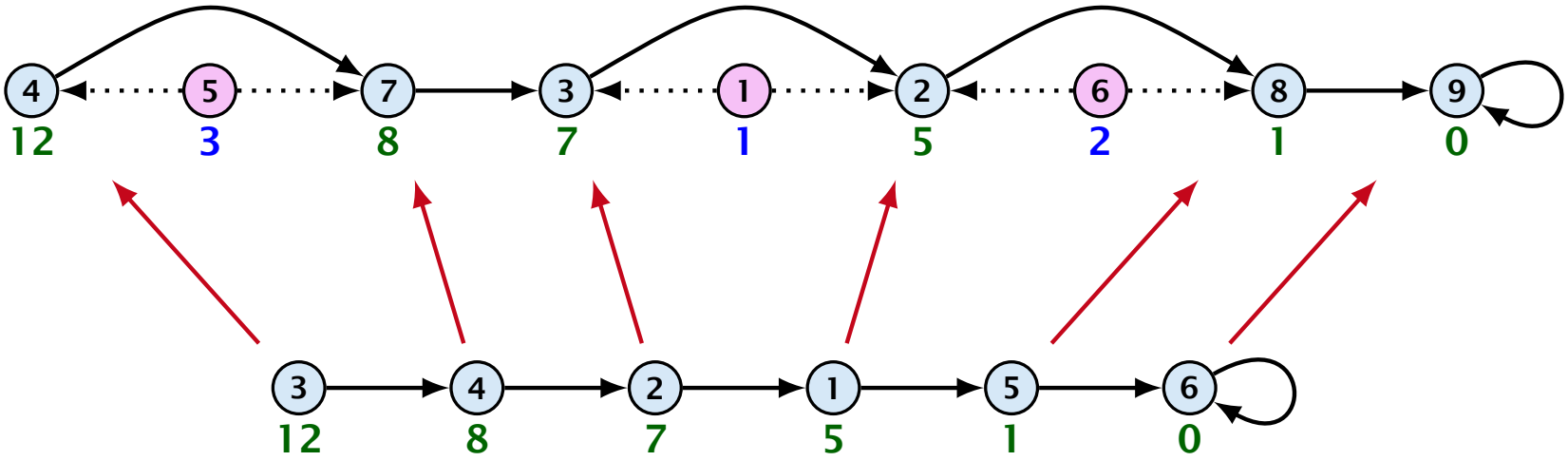

4. Compress remaining $n'$ nodes into a new array of $n'$ entries.
   The index positions can be computed by a prefix sum in time $\mathcal{O}(\log n)$ and work $\mathcal{O}(n)$
   Pointers can then be adjusted in time $\mathcal{O}(1)$.

# List Ranking



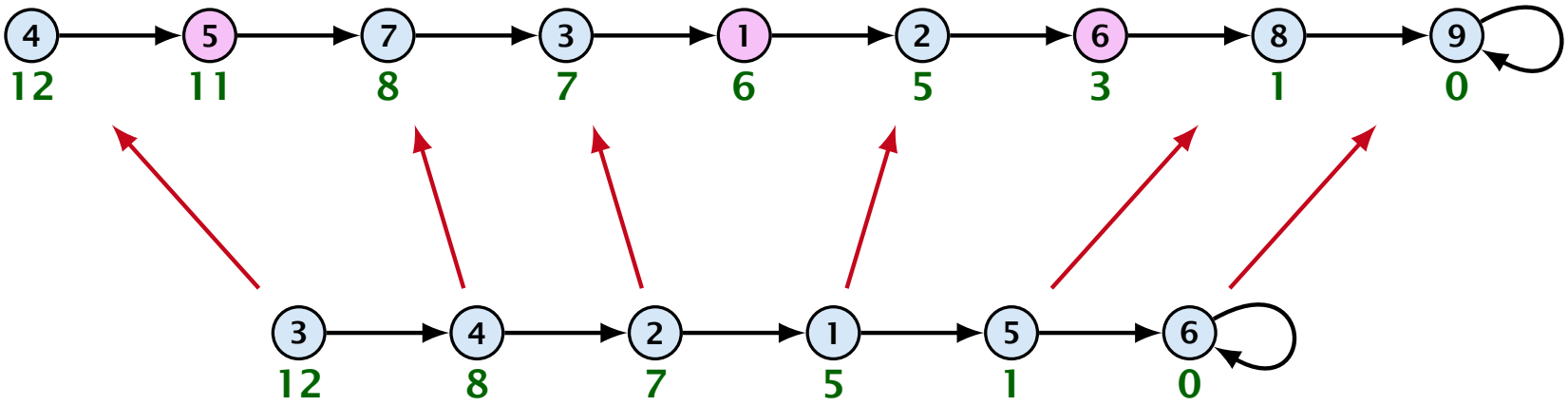

5. Solve the problem on the remaining list.
   If current size is less than $n/\log n$ do pointer jumping: time $\mathcal{O}(\log n)$; work $\mathcal{O}(n)$.
   Otherwise continue shrinking the list by finding an independent set

# List Ranking




6. Map the values back into the larger list. Time: $\mathcal{O}(1)$; Work: $\mathcal{O}(n)$

# List Ranking




7. Compute values for independent set nodes. Time: $\mathcal{O}(1)$; Work: $\mathcal{O}(1)$.
8. Splice nodes back into list. Time: $\mathcal{O}(1)$; Work: $\mathcal{O}(1)$.

We need $\mathcal{O}(\log\log n)$ shrinking iterations until the size of the remaining list reaches $\mathcal{O}(n/\log n)$.

Each shrinking iteration takes time $\mathcal{O}(\log n)$.

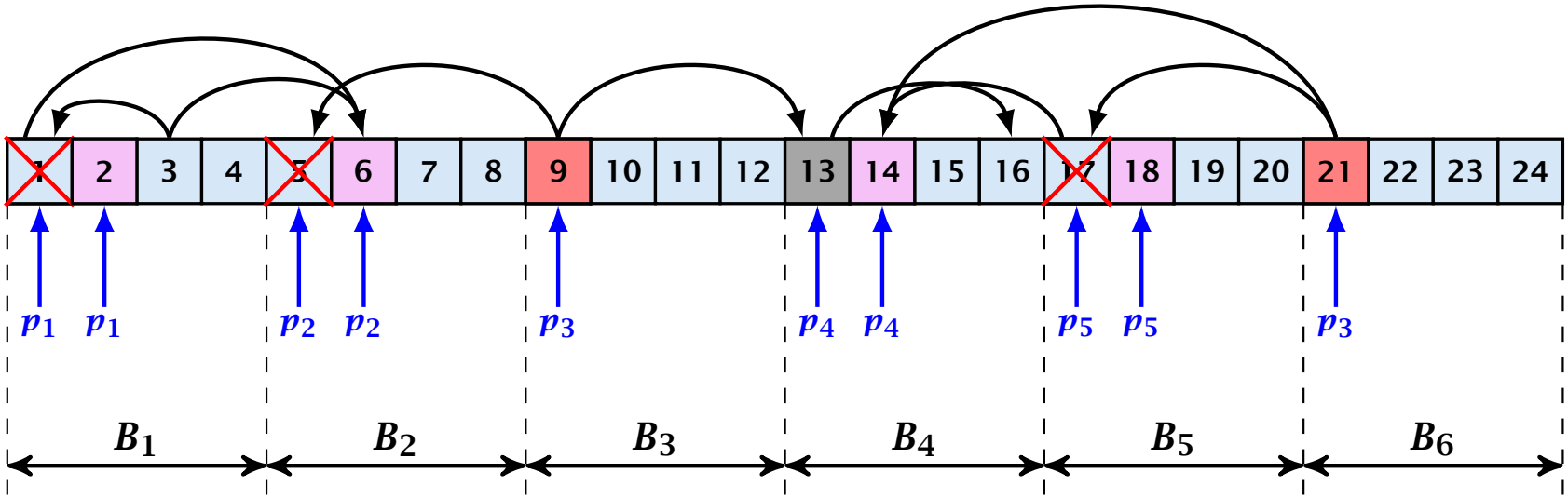
The work for all shrinking operations is just $\mathcal{O}(n)$, as the size of the list goes down by a constant factor in each round.

List Ranking can be solved in time $\mathcal{O}(\log n \log\log n)$ and work $\mathcal{O}(n)$ on an EREW-PRAM.

# Optimal List Ranking

In order to reduce the work we have to improve the shrinking of the list to $\mathcal{O}(n/\log n)$ nodes.

After this we apply pointer jumping




- some nodes are active;
- active nodes without neighbouring active nodes are isolated;
- the others form sublists;

1. delete isolated nodes from the list;
2. color each sublist with $\mathcal{O}(\log\log n)$ colors; time: $\mathcal{O}(1)$; work: $\mathcal{O}(n)$;

label local minima w.r.t. color as ruler; others as subject

# Optimal List Ranking

Each iteration requires constant time and work $\mathcal{O}(n/\log n)$, because we just work on one node in every block.

We need to prove that we just require $\mathcal{O}(\log n)$ iterations to reduce the size of the list to $\mathcal{O}(n/\log n)$.

**Observations/Remarks:**

- If the $p$-pointer of a block cannot be advanced without leaving the block, the processor responsible for this block simply stops working; all other blocks continue.
- The $p$-node of a block (the node $p_i$ is pointing to) at the beginning of a round is either a ruler with a living subject or the node will become active during the round.
- The subject nodes always lie to the left of the $p$-node of the respective block (if it exists).

**Measure of Progress**:

- a ruler will delete a subject
- an active node either
  - becomes a ruler (with a subject)
  - becomes a subject
  - is isolated and therefore gets deleted

# Analysis

For the analysis we assign a weight to every node in every block as follows.

**Definition 10**

The weight of the $i$-th node in a block is

$$(1-q)^i$$

with $q = \frac{1}{\log\log n}$, where the node-numbering starts from $0$.
Hence, a block has nodes $\{0,\ldots,\log n - 1\}$.

# Definition of Rulers

**Properties:**

- A ruler should have at most $\log \log n$ subjects.
- The weight of a ruler should be at most the weight of any of its subjects.
- Each ruler must have at least one subject.
- We must be able to remove the next subject in constant time.
- We need to make the ruler/subject decision in constant time.

Given a sublist of active nodes.

Color the sublist with $\mathcal{O}(\log\log n)$ colors. Take the local minima w.r.t. this coloring.

If the first node is not a ruler

- if the second node is a ruler switch ruler status between first and second
- otw. just make the first node into a ruler

**This partitions the sub-list into chains of length at most $\log\log n$ each starting with a ruler**

Now we change the ruler definition.

Consider some chain.

We make all local minima w.r.t. the weight function into a ruler; ties are broken according to block-id (so that comparing weights gives a strict inequality).

A ruler gets as subjects the nodes left of it until the next local maximum (or the start of the chain) (including the local maximum) and the nodes right of it until the next local maximum (or the end of the chain) (excluding the local maximum).

In case the first node is a ruler the above definition could leave it without a subject. We use constant time to fix this in some arbitrary manner

Set $q = \frac{1}{\log\log n}$.

The $i$-th node in a block is assigned a weight of $(1-q)^i$,
$0 \le i < \log n$

The total weight of a block is at most $1/q$ and the total weight of all items is at most $\frac{n}{q \log n}$.

**to show:**
After $\mathcal{O}(\log n)$ iterations the weight is at most
$(n/\log n)(1-q)^{\log n}$

This means at most $n/\log n$ nodes remain because the smallest weight a node can have is $(1-q)^{\log n - 1}$.

In every iteration the weight drops by a factor of

$$(1 - q/4) \ .$$

We consider subject nodes to just have half their weight.

We can view the step of becoming a subject as a precursor to deletion.

Hence, a node looses half its weight when becoming a subject and the remaining half when deleted.

Note that subject nodes will be deleted after just an additional $\mathcal{O}(\log\log n)$ iterations.

The weight is reduced because

- An isolated node is removed.
- A node is labelled as ruler, and the corresponding subjects reduce their weight by a factor of 1/2.
- A node is a ruler and deletes one of its subjects.

Hence, the weight reduction comes from $p$-nodes (ruler/active).

Each $p$-node is responsible for some other nodes; it has to generate a weight reduction large enough so that the weight of all nodes it is responsible for decreases by the desired factor.

An active node is responsible for all nodes that come after it in its block.

A ruler is responsible for all nodes that come after it in its block **and** for all its subjects.

Note that by this definition every node remaining in the list is covered.

# Case 1: Isolated Node

Suppose we delete an isolated node $v$ that is the $i$-th node in its block.

The weight of all node that $v$ is responsible for is

$$\sum_{i \le j < \log n} (1-q)^j$$

This weight reduces to

$$\sum_{i < j < \log n} (1-q)^j \le (1-q) \sum_{i \le j < \log n} (1-q)^j$$

Hence, weight reduces by a factor $(1-q) \le (1-q/4)$.

# Case 2: Creating Subjects

Suppose we generate a ruler with at least one subject.

Weight of ruler: $(1-q)^{i_1}$.
Weight of subjects: $(1-q)^{i_j}$, $2 \le j \le k$.

Initial weight:

$$Q = \sum_{j=1}^{k} \sum_{i_j \le \ell < \log n} (1-q)^{\ell} \le \frac{1}{q} \sum_{j=1}^{k} (1-q)^{i_j} \le \frac{2}{q} \sum_{j=2}^{k} (1-q)^{i_j}$$

New weight:

$$Q' = Q - \frac{1}{2} \sum_{j=2}^{k} (1-q)^{i_j} \le (1 - \frac{q}{4})Q$$

# Case 3: Removing Subjects

weight of ruler: $(1-q)^{i_1}$; weight of subjects: $(1-q)^{i_j}$, $2 \le j \le k$

Assume ruler removes subject with largest weight say $i_2$ (why?).

Initial weight:

$$\begin{aligned} Q &= \sum_{i_1 \le \ell < \log n} (1-q)^{\ell} + \frac{1}{2}\sum_{j=2}^{k}(1-q)^{i_j} \\ &\le \frac{1}{q}(1-q)^{i_1} + \frac{k}{2}(1-q)^{i_2} \\ &\le \frac{1}{q}(1-q)^{i_2} + \frac{1}{2q}(1-q)^{i_2} \end{aligned}$$

New weight:

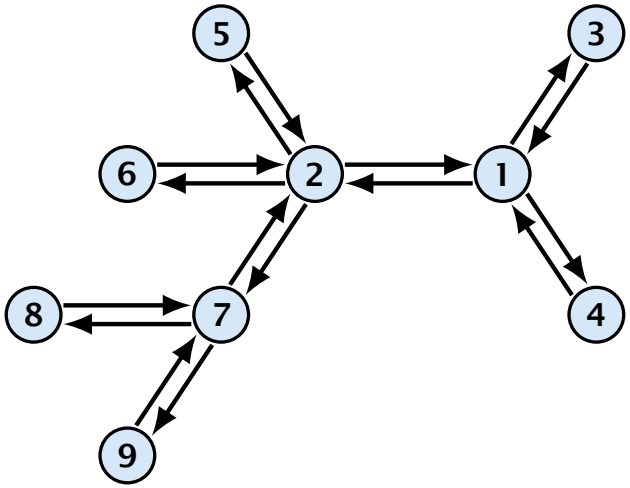
$$Q' = Q - \frac{1}{2}(1-q)^{i_2} \le (1-\frac{q}{3})Q$$

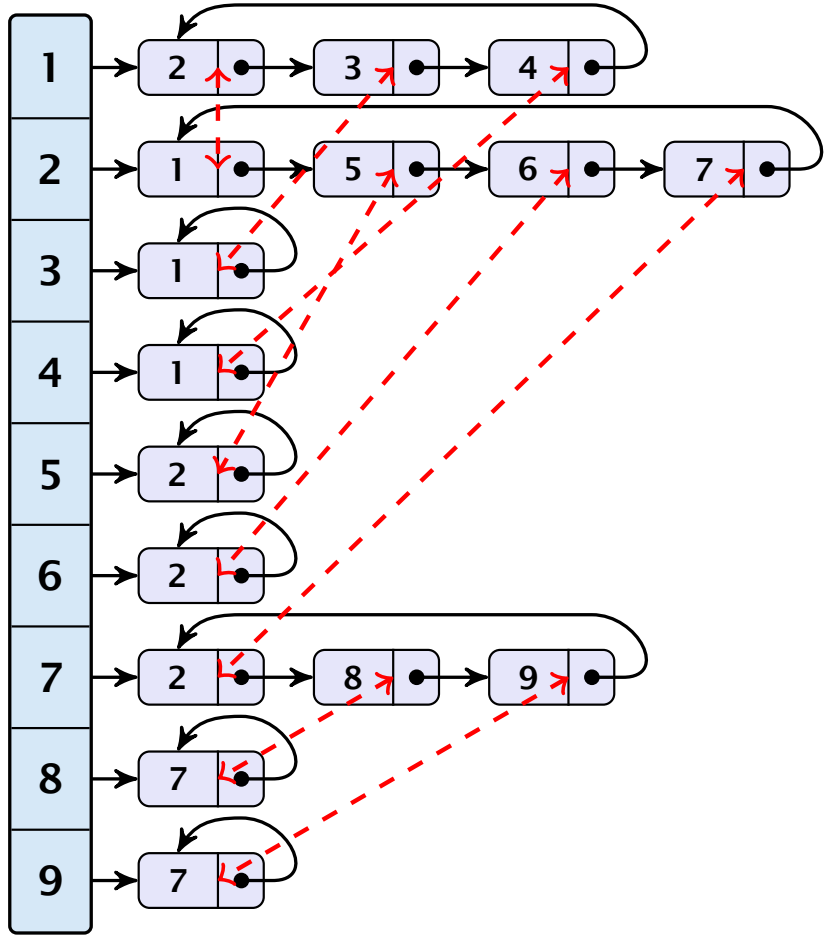
After $s$ iterations the weight is at most

$$\frac{n}{q\log n}\left(1-\frac{q}{4}\right)^s \stackrel{!}{\leq} \frac{n}{\log n}(1-q)^{\log n}$$

Choosing $i = 5\log n$ the inequality holds for sufficiently large $n$.

# Tree Algorithms



# Euler Circuits

Every node $v$ fixes an arbitrary ordering among its adjacent nodes:

$$u_0, u_1, \ldots, u_{d-1}$$

We obtain an Euler tour by setting

$$\mathrm{succ}((u_i, v)) = (v, u_{(i+1) \bmod d})$$

# Euler Circuits

**Lemma 11**

*An Euler circuit can be computed in constant time $\mathcal{O}(1)$ with $\mathcal{O}(n)$ operations.*

# Euler Circuits — Applications

**Rooting a tree**

- split the Euler tour at node $r$
- this gives a list on the set of directed edges (Euler path)
- assign $x[e] = 1$ for every edge;
- perform parallel prefix; let $s[\cdot]$ be the result array
- if $s[(u,v)] < s[(v,u)]$ then $u$ is parent of $v$;

# Euler Circuits — Applications

**Postorder Numbering**

- split the Euler tour at node $r$
- this gives a list on the set of directed edges (Euler path)
- assign $x[e] = 1$ for every edge $(v, \text{parent}(v))$
- assign $x[e] = 0$ for every edge $(\text{parent}(v), v)$
- perform parallel prefix
- $\text{post}(v) = s[(v, \text{parent}(v))]$; $\text{post}(r) = n$

# Euler Circuits — Applications

**Level of nodes**

- split the Euler tour at node $r$
- this gives a list on the set of directed edges (Euler path)
- assign $x[e] = -1$ for every edge $(v, \text{parent}(v))$
- assign $x[e] = 1$ for every edge $(\text{parent}(v), v)$
- perform parallel prefix
- $\text{level}(v) = s[(\text{parent}(v), v)]$; $\text{level}(r) = 0$

# Euler Circuits — Applications

**Number of descendants**

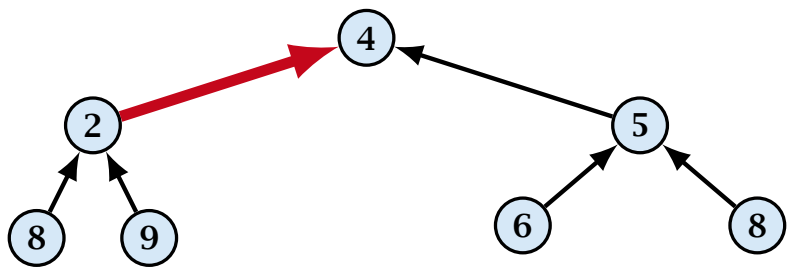
- split the Euler tour at node $r$
- this gives a list on the set of directed edges (Euler path)
- assign $x[e] = 0$ for every edge $(\text{parent}(v), v)$
- assign $x[e] = 1$ for every edge $(v, \text{parent}(v))$, $v \neq r$
- perform parallel prefix
- $\text{size}(v) = s[(v, \text{parent}(v))] - s[(\text{parent}(v), v)]$

# Rake Operation

Given a binary tree $T$.

Given a leaf $u \in T$ with $p(u) \neq r$ the rake-operation does the following

- remove $u$ and $p(u)$
- attach sibling of $u$ to $p(p(u))$



We want to apply rake operations to a binary tree $T$ until $T$ just consists of the root with two children.

**Possible Problems:**

1. we could concurrently apply the rake-operation to two siblings
2. we could concurrently apply the rake-operation to two leaves $u$ and $v$ such that $p(u)$ and $p(v)$ are connected

By choosing leaves carefully we ensure that none of the above cases occurs

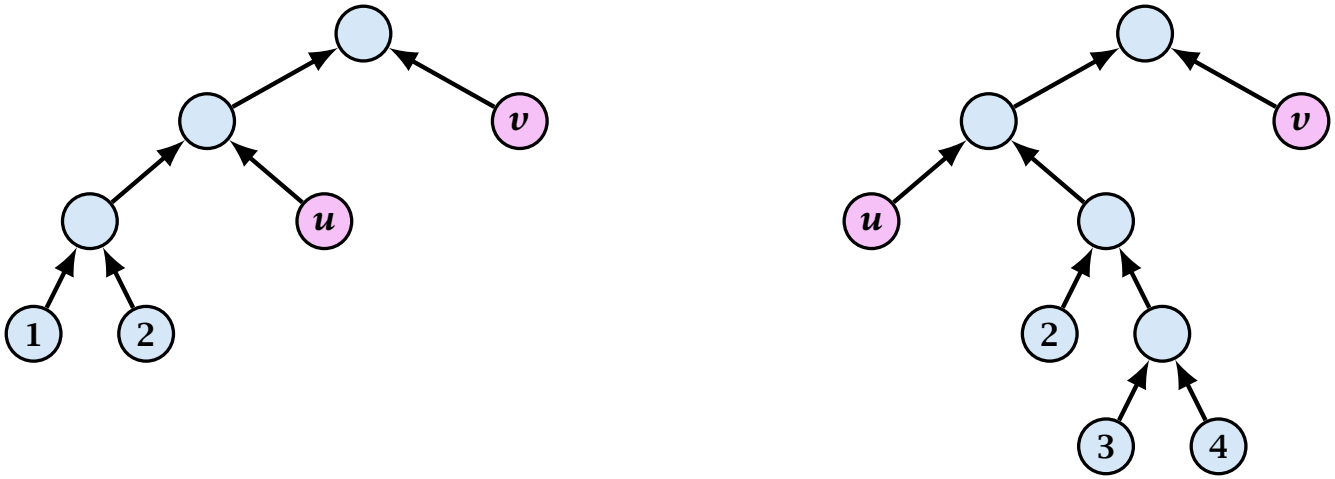
**Algorithm:**

- label leaves consecutively from left to right (excluding left-most and right-most leaf), and store them in an array $A$
- for $\lceil \log(n+1) \rceil$ iterations
  - apply rake to all odd leaves that are left children
  - apply rake operation to remaining odd leaves (odd at start of round!!!)
  - A=even leaves

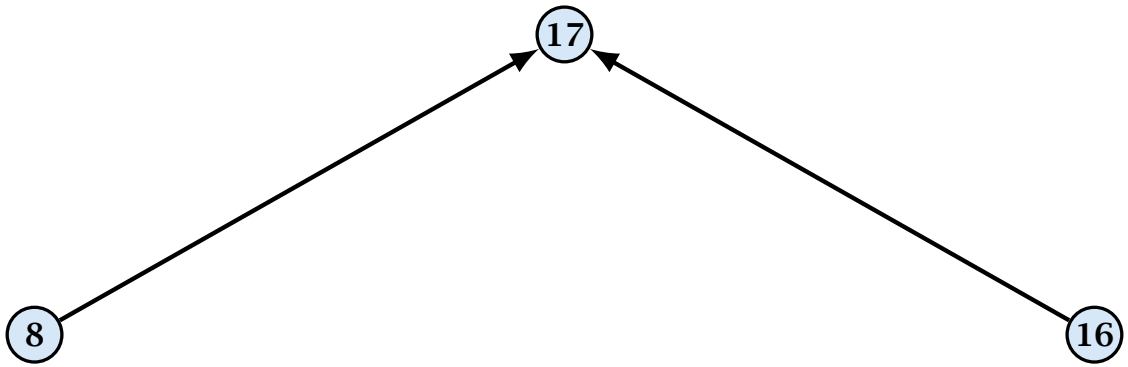
## Observations

- the rake operation does not change the order of leaves
- two leaves that are siblings do not perform a rake operation in the same round because one is even and one odd at the start of the round
- two leaves that have adjacent parents either have different parity (even/odd) or they differ in the type of child (left/right)

Cases, when the left edge btw. $p(u)$ and $p(v)$ is a left-child edge.



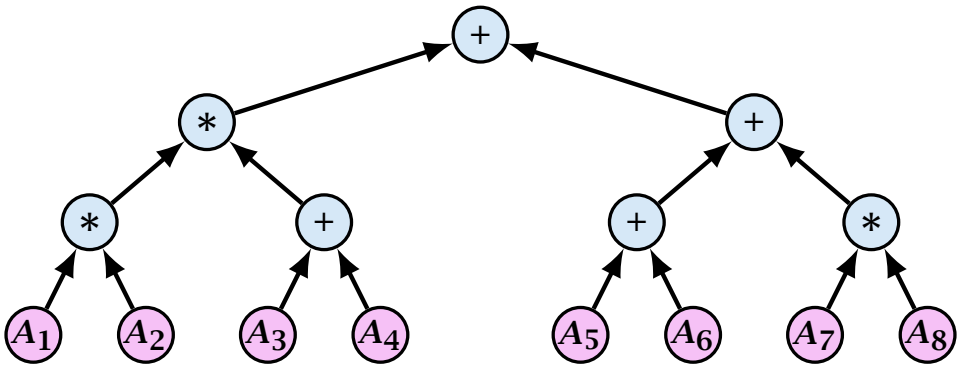
# Example

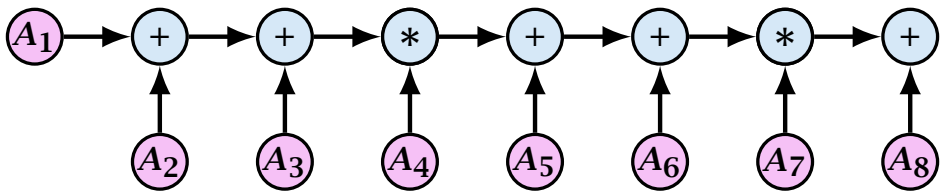


- one iteration can be performed in constant time with $\mathcal{O}(|A|)$ processors, where $A$ is the array of leaves;
- hence, **all** iterations can be performed in $\mathcal{O}(\log n)$ time and $\mathcal{O}(n)$ work;
- the intial parallel prefix also requires time $\mathcal{O}(\log n)$ and work $\mathcal{O}(n)$

# Evaluating Expressions

Suppose that we want to evaluate an expression tree, containing additions and multiplications.




If the tree is not balanced this may be time-consuming.

We can use the rake-operation to do this quickly.

Applying the rake-operation changes the tree.

In order to maintain the value we introduce parameters $a_v$ and $b_v$ for every node that still allows to compute the value of a node based on the value of its children.

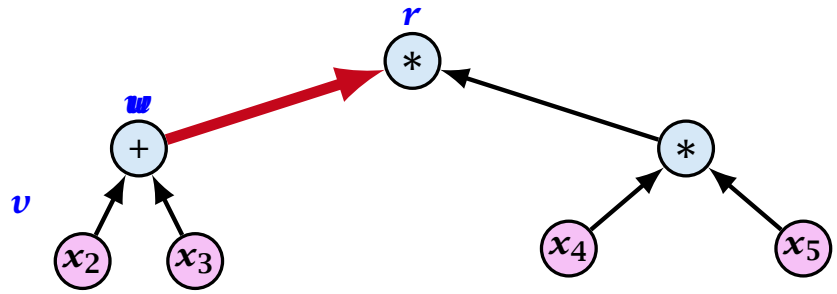
**Invariant:**
Let $u$ be internal node with children $v$ and $w$. Then

$$\mathrm{val}(u) = (a_v \cdot \mathrm{val}(v) + b_v) \otimes (a_w \cdot \mathrm{val}(w) + b_w)$$

where $\otimes \in \{*, +\}$ is the operation at node $u$.

Initially, we can choose $a_v = 1$ and $b_v = 0$ for every node.

# Rake Operation




Currently the value at $u$ is

$$\begin{aligned}\mathrm{val}(u) &= (a_v \cdot \mathrm{val}(v) + b_v) + (a_w \cdot \mathrm{val}(w) + b_w) \\ &= x_1 + (a_w \cdot \mathrm{val}(w) + b_w)\end{aligned}$$

In the expression for $r$ this goes in as

$$\begin{aligned}&a_u \cdot [x_1 + (a_w \cdot \mathrm{val}(w) + b_w)] + b_u \\ &= \underbrace{a_u a_w}_{a'_w} \cdot \mathrm{val}(w) + \underbrace{a_u x_1 + a_u b_w + b_u}_{b'_w}\end{aligned}$$

If we change the $a$ and $b$-values during a rake-operation according to the previous slide we can calculate the value of the root in the end.

**Lemma 12**

*We can evaluate an arithmetic expression tree in time $\mathcal{O}(\log n)$ and work $\mathcal{O}(n)$ regardless of the height or depth of the tree.*

By performing the rake-operation in the reverse order we can also compute the value at each node in the tree.

**Lemma 13**

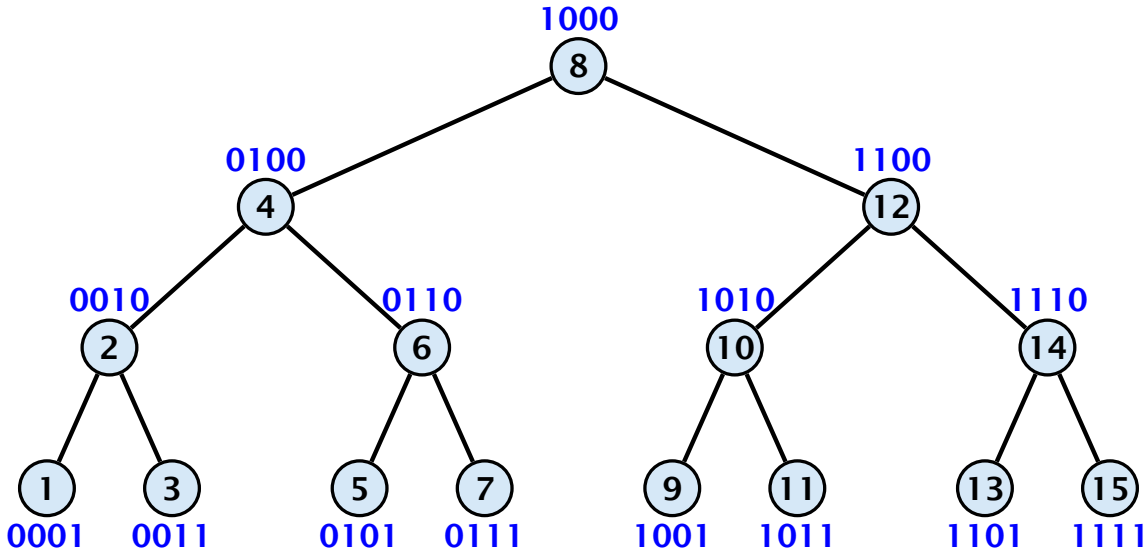
*We compute tree functions for arbitrary trees in time $\mathcal{O}(\log n)$ and a linear number of operations.*

proof on board...

In the LCA (least common ancestor) problem we are given a tree and the goal is to design a data-structure that answers LCA-queries in constant time.

# Least Common Ancestor

LCAs on complete binary trees (inorder numbering):

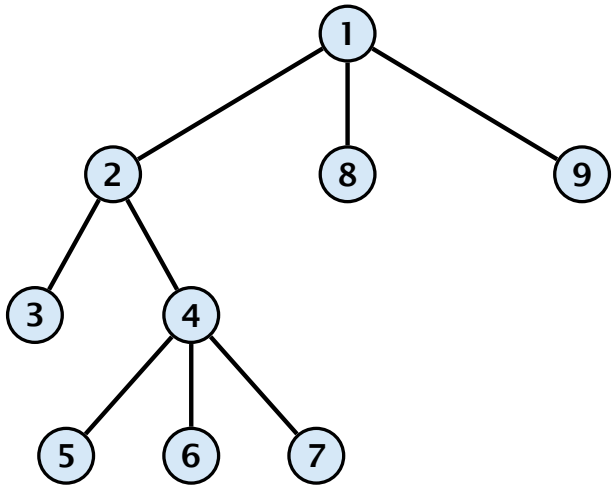



The least common ancestor of $u$ and $v$ is

$$z_1 z_2 \ldots z_i 1 0 \ldots 0$$

where $z_{i+1}$ is the first bit-position in which $u$ and $v$ differ.

# Least Common Ancestor




| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| nodes | 1 | 2 | 3 | 2 | 4 | 5 | 4 | 6 | 4 | 7 | 4 | 2 | 1 | 8 | 1 | 9 | 1 |
| levels | 0 | 1 | 2 | 1 | 2 | 3 | 2 | 3 | 2 | 3 | 2 | 1 | 0 | 1 | 0 | 1 | 0 |

$\ell(v)$ is index of first appearance of $v$ in node-sequence.

$r(v)$ is index of last appearance of $v$ in node-squence.

$\ell(v)$ and $r(v)$ can be computed in constant time, given the node- and level-sequence.

# Least Common Ancestor

**Lemma 14**

1. $u$ *is ancestor of* $v$ *iff* $\ell(u) < \ell(v) < r(u)$
2. $u$ *and* $v$ *are not related iff either* $r(u) < \ell(v)$ *or* $\ell(u) < r(v)$
3. *suppose* $r(u) < \ell(v)$ *then* $\mathrm{LCA}(u,v)$ *is vertex with minimum level over interval* $[r(u), \ell(v)]$.

# Range Minima Problem

Given an array $A[1 \ldots n]$, a range minimum query $(\ell, r)$ consists of a left index $\ell \in \{1, \ldots, n\}$ and a right index $r \in \{1, \ldots, n\}$.

The answer has to return the index of the minimum element in the subsequence $A[\ell \ldots r]$.

The goal in the range minima problem is to preprocess the array such that range minima queries can be answered quickly (constant time).

**Observation**

Given an algorithm for solving the range minima problem in time $T(n)$ and work $W(n)$ we can obtain an algorithm that solves the LCA-problem in time $\mathcal{O}(T(n) + \log n)$ and work $\mathcal{O}(n + W(n))$.

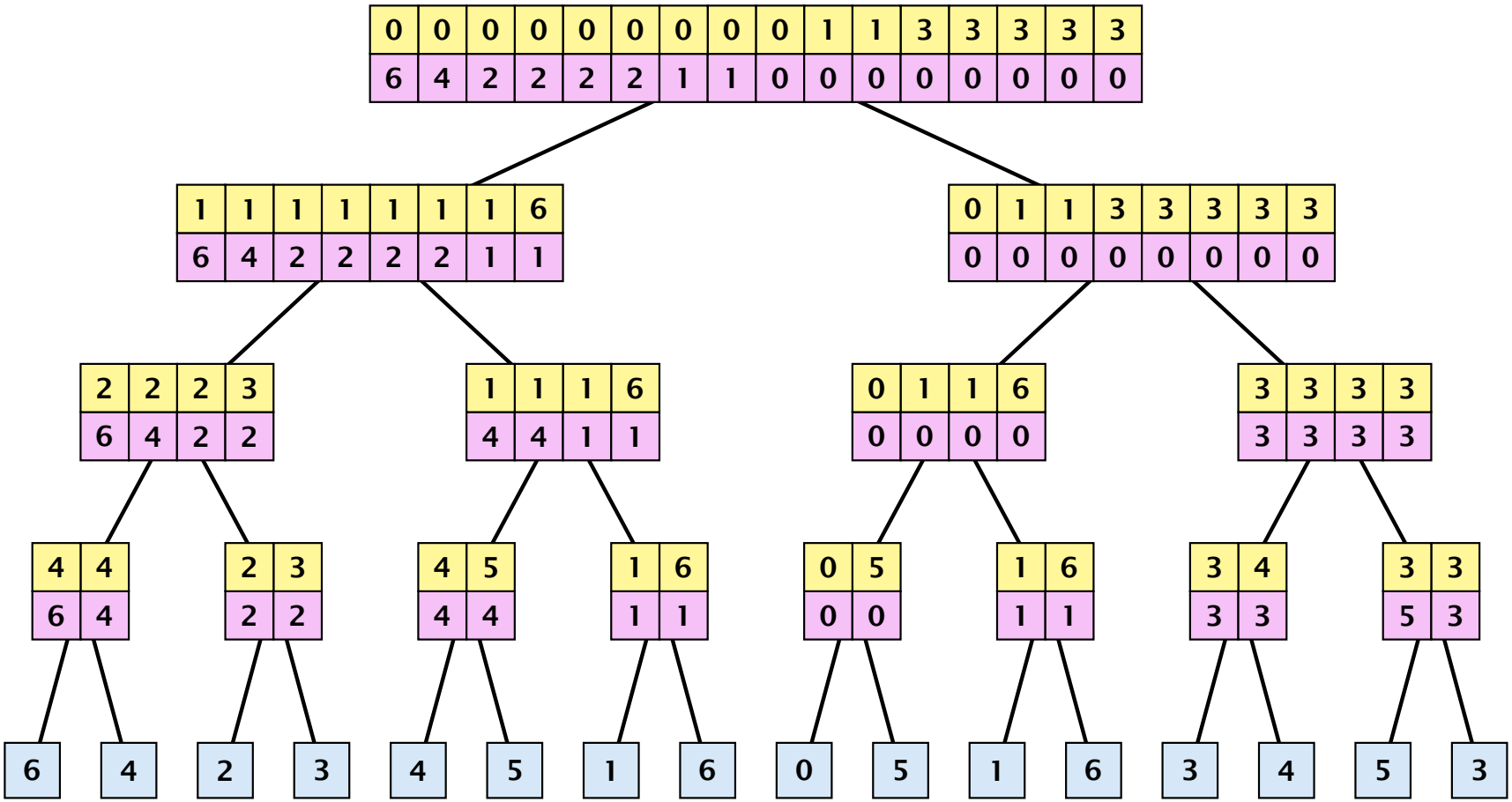
**Remark**

In the sequential setting the LCA-problem and the range minima problem are equivalent. This is not necessarily true in the parallel setting.

For solving the LCA-problem it is sufficient to solve the restricted range minima problem where two successive elements in the array just differ by $+1$ or $-1$.

# Prefix and Suffix Minima

Tree with prefix-minima and suffix-minima:



- Suppose we have an array $A$ of length $n = 2^k$
- We compute a complete binary tree $T$ with $n$ leaves.
- Each internal node corresponds to a subsequence of $A$. It contains an array with the prefix and suffix minima of this subsequence.

Given the tree $T$ we can answer a range minimum query $(\ell, r)$ in constant time.

- we can determine the LCA $x$ of $\ell$ and $r$ in constant time since $T$ is a complete binary tree
- Then we consider the suffix minimum of $\ell$ in the left child of $x$ and the prefix minimum of $r$ in the right child of $x$.
- The minimum of these two values is the result.

**Lemma 15**

*We can solve the range minima problem in time $\mathcal{O}(\log n)$ and work $\mathcal{O}(n \log n)$.*

# Reducing the Work

Partition $A$ into blocks $B_i$ of length $\log n$

Preprocess each $B_i$ block separately by a sequential algorithm so that range-minima queries within the block can be answered in constant time. (**how?**)

For each block $B_i$ compute the minimum $x_i$ and its prefix and suffix minima.

Use the previous algorithm on the array $(x_1, \ldots, x_{n/\log n})$.

**Answering a query $(\ell, r)$:**

- if $\ell$ and $r$ are from the same block the data-structure for this block gives us the result in constant time
- if $\ell$ and $r$ are from different blocks the result is a minimum of three elements:
  - the suffix minmum of entry $\ell$ in $\ell$'s block
  - the minimum among $x_{\ell+1}, \ldots, x_{r-1}$
  - the prefix minimum of entry $r$ in $r$'s block

# Searching

An extension of binary search with $p$ processors gives that one can find the rank of an element in

$$\log_{p+1}(n) = \frac{\log n}{\log(p+1)}$$

many parallel steps with $p$ processors. (not work-optimal)

This requires a CREW PRAM model. For the EREW model searching cannot be done faster than $\mathcal{O}(\log n - \log p)$ with $p$ processors even if there are $p$ copies of the search key.

# Merging

Given two sorted sequences $A = (a_1, \ldots, a_n)$ and $B = (b_1, \ldots, b_n)$, compute the sorted squence $C = (c_1, \ldots, c_n)$.

**Definition 16**

Let $X = (x_1, \ldots, x_t)$ be a sequence. The rank $\mathrm{rank}(y : X)$ of $y$ in $X$ is

$$\mathrm{rank}(y : X) = |\{x \in X \mid x \leq y\}|$$

For a sequence $Y = (y_1, \ldots, y_s)$ we define $\mathrm{rank}(Y : X) := (r_1, \ldots, r_s)$ with $r_i = \mathrm{rank}(y_i : X)$.

# Merging

We have already seen a merging-algorithm that runs in time $\mathcal{O}(\log n)$ and work $\mathcal{O}(n)$.

Using the fast search algorithm we can improve this to a running time of $\mathcal{O}(\log\log n)$ and work $\mathcal{O}(n \log\log n)$.

# Merging

Input: $A = a_1, \ldots, a_n$; $B = b_1, \ldots, b_m$; $m \leq n$

1. if $m < 4$ then rank elements of $B$, using the parallel search algorithm with $p$ processors. Time: $\mathcal{O}(1)$. Work: $\mathcal{O}(n)$.
2. Concurrently rank elements $b_{\sqrt{m}}, b_{2\sqrt{m}}, \ldots, b_m$ in $A$ using the parallel search algorithm with $p = \sqrt{n}$. Time: $\mathcal{O}(1)$. Work: $\mathcal{O}(\sqrt{m} \cdot \sqrt{n}) = \mathcal{O}(n)$

   $j(i) := \text{rank}(b_{i\sqrt{m}} : A)$
3. Let $B_i = (b_{i\sqrt{m}+1}, \ldots, b_{(i+1)\sqrt{m}-1})$; and $A_i = (a_{j(i)+1}, \ldots, a_{j(i+1)})$.

   Recursively compute $\text{rank}(B_i : A_i)$.
4. Let $k$ be index not a multiple of $\sqrt{m}$. $i = \lceil \frac{k}{\sqrt{m}} \rceil$. Then $\text{rank}(b_k : A) = j(i) + \text{rank}(b_k : A_i)$.

The algorithm can be made work-optimal by standard techniques.

proof on board...

# Mergesort

**Lemma 17**

*A straightforward parallelization of Mergesort can be implemented in time $\mathcal{O}(\log n \log\log n)$ and with work $\mathcal{O}(n \log n)$.*

*This assumes the CREW-PRAM model.*

# Mergesort

Let $L[v]$ denote the (sorted) sublist of elements stored at the leaf nodes rooted at $v$.

We can view Mergesort as computing $L[v]$ for a complete binary tree where the leaf nodes correspond to nodes in the given array.

Since the merge-operations on one level of the complete binary tree can be performed in parallel we obtain time $\mathcal{O}(h \log\log n)$ and work $\mathcal{O}(hn)$, where $h = \mathcal{O}(\log n)$ is the height of the tree.

# Pipelined Mergesort

We again compute $L[v]$ for every node in the complete binary tree.

After round $s$, $L_s[v]$ is an **approximation** of $L[v]$ that will be improved in future rounds.

For $s \geq 3\,\mathrm{height}(v)$, $L_s[v] = L[v]$.

# Pipelined Mergesort

In every round, a node $v$ sends $\text{sample}(L_s[v])$ (an approximation of its current list) upwards, and receives approximations of the lists of its children.

It then computes a new approximation of its list.

A node is called active in round $s$ if $s \leq 3\,\text{height}(v)$ (this means its list is not yet complete at the start of the round, i.e., $L_{s-1}[v] \neq L[v]$).

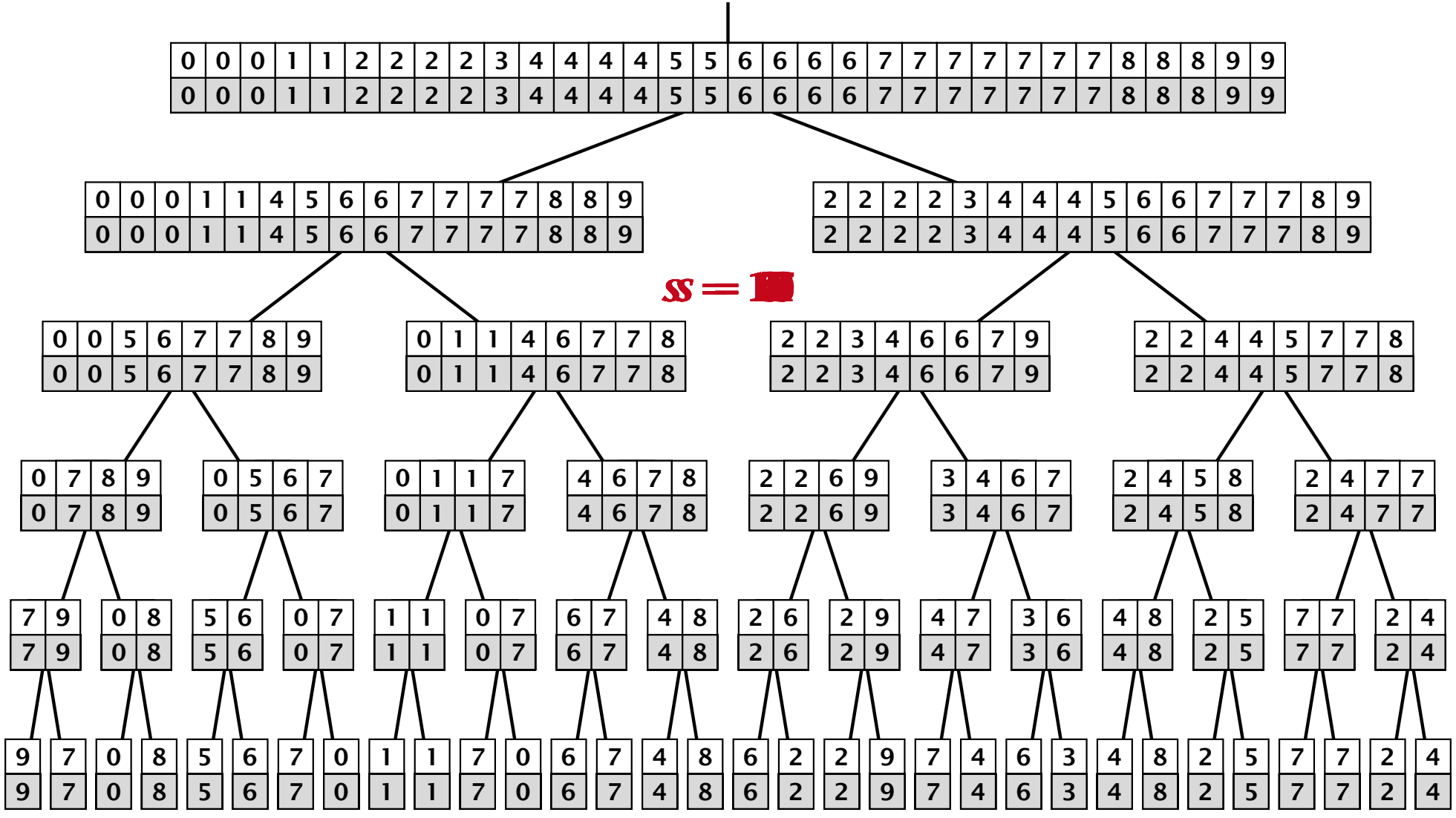
# Pipelined Mergesort

**Algorithm 11** ColeSort()

1: initialize $L_0[v] = A_v$ for leaf nodes; $L_0[v] = \emptyset$ otw.
2: **for** $s \leftarrow 1$ **to** $3 \cdot \text{height}(T)$ **do**
3: **for** all active nodes $v$ **do**
4: // $u$ and $w$ children of $v$
5: $L'_s[u] \leftarrow \text{sample}(L_{s-1}[u])$
6: $L'_s[w] \leftarrow \text{sample}(L_{s-1}[w])$
7: $L_s[v] \leftarrow \text{merge}(L'_s[u], L'_s[u])$

$$\text{sample}(L_s[v]) = \begin{cases} \text{sample}_4(L_s[v]) & s \leq 3\,\text{height}(v) \\ \text{sample}_2(L_s[v]) & s = 3\,\text{height}(v) + 1 \\ \text{sample}_1(L_s[v]) & s = 3\,\text{height}(v) + 2 \end{cases}$$

# Colesort


0 0 0 1 1 2 2 2 2 3 4 4 4 4 5 5 6 6 6 6 7 7 7 7 7 7 7 8 8 8 9 9
0 0 0 1 1 2 2 2 2 3 4 4 4 4 5 5 6 6 6 6 7 7 7 7 7 7 7 8 8 8 9 9
0 0 0 1 1 4 5 6 6 7 7 7 7 8 8 9
0 0 0 1 1 4 5 6 6 7 7 7 7 8 8 9
2 2 2 2 3 4 4 4 5 6 6 7 7 7 8 9
2 2 2 2 3 4 4 4 5 6 6 7 7 7 8 9
0 0 5 6 7 7 8 9
0 0 5 6 7 7 8 9
0 1 1 4 6 7 7 8
0 1 1 4 6 7 7 8
2 2 3 4 6 6 7 9
2 2 3 4 6 6 7 9
2 2 4 4 5 7 7 8
2 2 4 4 5 7 7 8
0 7 8 9
0 7 8 9
0 5 6 7
0 5 6 7
0 1 1 7
0 1 1 7
4 6 7 8
4 6 7 8
2 2 6 9
2 2 6 9
3 4 6 7
3 4 6 7
2 4 5 8
2 4 5 8
2 4 7 7
2 4 7 7
7 9
7 9
0 8
0 8
5 6
5 6
0 7
0 7
1 1
1 1
0 7
0 7
6 7
6 7
4 8
4 8
2 6
2 6
2 9
2 9
4 7
4 7
3 6
3 6
4 8
4 8
2 5
2 5
7 7
7 7
2 4
2 4
9 7 0 8 5 6 7 0 1 1 7 0 6 7 4 8 6 2 2 9 7 4 6 3 4 8 2 5 7 7 2 4
9 7 0 8 5 6 7 0 1 1 7 0 6 7 4 8 6 2 2 9 7 4 6 3 4 8 2 5 7 7 2 4

# Pipelined Mergesort

## Lemma 18

*After round $s = 3\,\mathrm{height}(v)$, the list $L_s[v]$ is complete.*

**Proof:**

- clearly true for leaf nodes
- suppose it is true for all nodes up to height $h$;
- fix a node $v$ on level $h+1$ with children $u$ and $w$
- $L_{3h}[u]$ and $L_{3h}[w]$ are complete by induction hypothesis
- further $\mathrm{sample}(L_{3h+2}[u]) = L[u]$ and $\mathrm{sample}(L_{3h+2}[v]) = L[v]$
- hence in round $3h+3$ node $v$ will merge the complete list of its children; after the round $L[v]$ will be complete

# Pipelined Mergesort

**Lemma 19**

*The number of elements in lists $L_s[v]$ for active nodes $v$ is at most $\mathcal{O}(n)$.*

proof on board...

**Definition 20**

A sequence $X$ is a $c$-cover of a sequence $Y$ if for any two consecutive elements $\alpha, \beta$ from $(-\infty, X, \infty)$ the set $|\{y_i \mid \alpha \leq y_i \leq \beta\}| \leq c$.

# Pipelined Mergesort

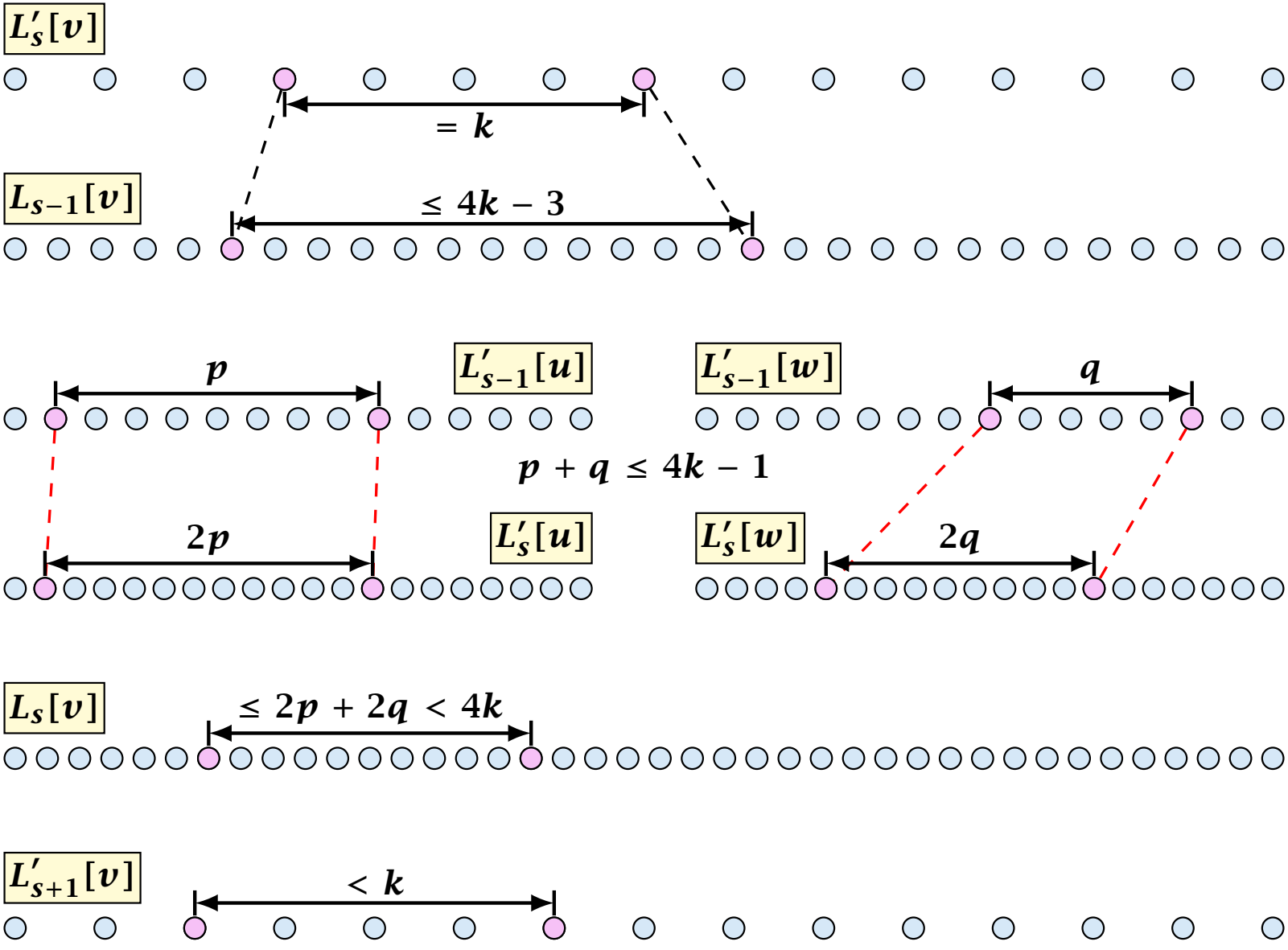
**Lemma 21**

*$L'_s[v]$ is a 4-cover of $L'_{s+1}[v]$.*

If $[a,b]$ with $a,b \in L'_s[v] \cup \{-\infty,\infty\}$ fulfills $|[a,b] \cap (L'_s[v] \cup \{-\infty,\infty\})| = k$ we say $[a,b]$ intersects $(-\infty, L'_s[v], +\infty)$ in $k$ items.

**Lemma 22**

*If $[a,b]$ intersects $(-\infty, L'_s[v], \infty)$ in $k \geq 2$ items, then $[a,b]$ intersects $(-\infty, L'_{s+1}, \infty)$ in at most $2k$ items.*




Note that the last step holds as long $L'_{s+1}[v] = \text{sample}_4(L_s[v])$. Otw. $L_{s-1}[v]$ has already been full, and hence, $L'_s[v], L'_{s+1}[v], L'_{s+2}[v]$ are 4-covers of the complete list $L[v]$, and also 4-covers of each other.

# Merging with a Cover

**Lemma 23**

*Given two sorted sequences $A$ and $B$. Let $X$ be a $c$-cover of $A$ and $B$ for constant $c$, and let $\mathrm{rank}(X:A)$ and $\mathrm{rank}(X:B)$ be known.*

*We can merge $A$ and $B$ in time $\mathcal{O}(1)$ using $\mathcal{O}(|X|)$ operations.*

# Merging with a Cover

**Lemma 24**

*Given two sorted sequences $A$ and $B$. Let $X$ be a $c$-cover of $A$ for constant $c$, and let* $\mathrm{rank}(X:A)$ *and* $\mathrm{rank}(X:B)$ *be known.*

*We can merge $A$ and $B$ in time $\mathcal{O}(1)$ using $\mathcal{O}(|X|+|B|)$ operations; this means we can compute* $\mathrm{rank}(A:B)$ *and* $\mathrm{rank}(B:A)$.

In order to do the merge in iteration $s+1$ in constant time we need to know

$$\text{rank}(L_s[v] : L'_{s+1}[u]) \text{ and } \text{rank}(L_s[v] : L'_{s+1}[v])$$

and we need to know that $L_s[v]$ is a 4-cover of $L'_{s+1}[u]$ and $L'_{s+1}[v]$.

**Lemma 25**

*$L_s[v]$ is a 4-cover of $L'_{s+1}[u]$ and $L'_{s+1}[v]$.*

- $L_s[v] \supseteq L'_s[u], L'_s[u]$
- $L'_s[u]$ is 4-cover of $L'_{s+1}[u]$
- Hence, $L_s[v]$ is 4-cover of $L'_{s+1}[u]$ as adding more elements cannot destroy the cover-property.

# Analysis

## Lemma 26

*Suppose we know for every internal node $v$ with children $u$ and $w$*

- $\text{rank}(L'_s[v] : L'_{s+1}[v])$
- $\text{rank}(L'_s[u] : L'_s[w])$
- $\text{rank}(L'_s[w] : L'_s[u])$

*We can compute*

- $\text{rank}(L'_{s+1}[v] : L'_{s+2}[v])$
- $\text{rank}(L'_{s+1}[u] : L'_{s+1}[w])$
- $\text{rank}(L'_{s+1}[w] : L'_{s+1}[u])$

*in constant time and $\mathcal{O}(|L_{s+1}[v]|)$ operations, where $v$ is the parent of $u$ and $w$.*

Given

- $\text{rank}(L'_s[u] : L'_{s+1}[u])$ **(4-cover)**
- $\text{rank}(L'_s[u] : L'_s[w])$
- $\text{rank}(L'_s[w] : L'_s[u])$
- $\text{rank}(L'_s[w] : L'_{s+1}[w])$ **(4-cover)**

Compute

- $\text{rank}(L'_s[w] : L'_{s+1}[u])$
- $\text{rank}(L'_s[u] : L'_{s+1}[w])$

Compute

- $\text{rank}(L'_{s+1}[w] : L'_{s+1}[u])$
- $\text{rank}(L'_{s+1}[u] : L'_{s+1}[w])$

**ranks between siblings can be computed easily**

Given

- $\text{rank}(L'_s[u] : L'_{s+1}[u])$ **(4-cover)**
- $\text{rank}(L'_s[u] : L'_{s+1}[w])$
- $\text{rank}(L'_s[w] : L'_{s+1}[u])$
- $\text{rank}(L'_s[w] : L'_{s+1}[w])$ **(4-cover)**

Compute (recall that $L_s[v] = \text{merge}(L'_s[u], L'_s[w])$)

- $\text{rank}(L_s[v] : L'_{s+1}[u])$
- $\text{rank}(L_s[v] : L'_{s+1}[w])$

Compute

- $\text{rank}(L_s[v] : L_{s+1}[v])$ (by adding)
- $\text{rank}(L'_{s+1}[v] : L'_{s+2}[v])$ (by sampling)

**Definition 27**

A $0$-$1$ sequence $S$ is bitonic if it can be written as the concatenation of subsequences $S_1$ and $S_2$ such that either

- $S_1$ is monotonically increasing and $S_2$ monotonically decreasing, or
- $S_1$ is monotonically decreasing and $S_2$ monotonically increasing.

Note, that this just defines bitonic $0$-$1$ sequences. Bitonic sequences are defined differently.

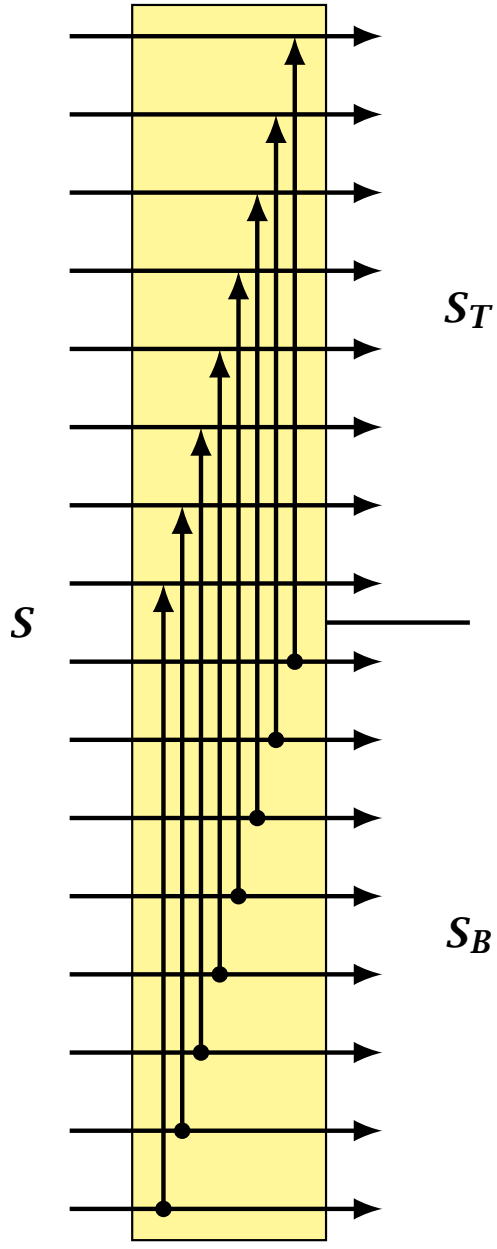
# Bitonic Merger

If we feed a bitonic $0$-$1$ sequence $S$ into the network on the right we obtain two bitonic sequences $S_T$ and $S_B$ s.t.

1. $S_B \leq S_T$ (element-wise)
2. $S_B$ and $S_T$ are bitonic

**Proof:**

- assume wlog. $S$ more $1$'s than $0$'s.
- assume for contradiction two $0$s at same comparator $(i, j = i + 2^d)$
  - everything $0$ btw $i$ and $j$ means we have more than 50% zeros (↯).
  - all $1$s btw. $i$ and $j$ means we have less than 50% ones (↯).
  - $1$ btw. $i$ and $j$ and elsewhere means $S$ is not bitonic (↯).



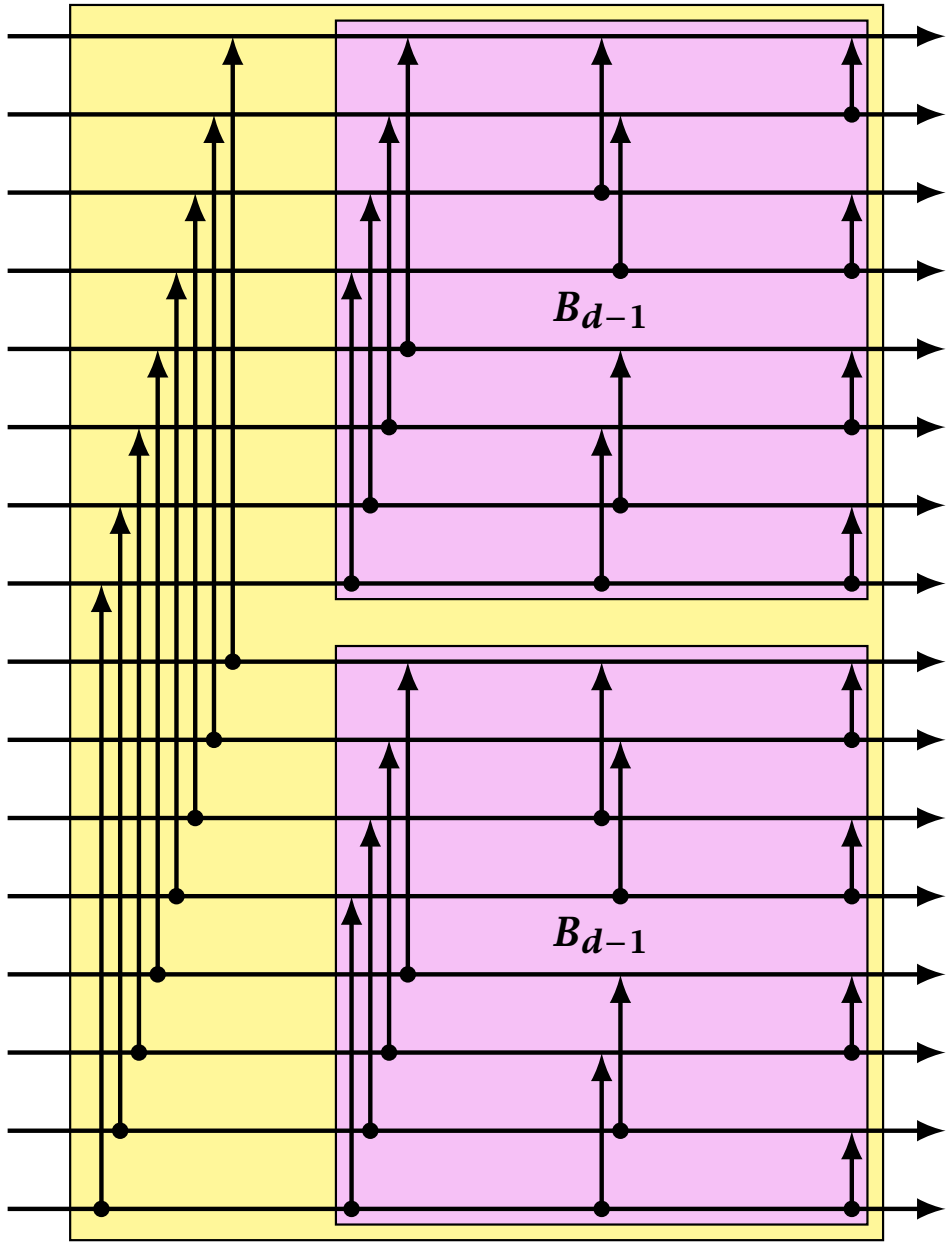
# Bitonic Merger

**Bitonic Merger $B_d$**

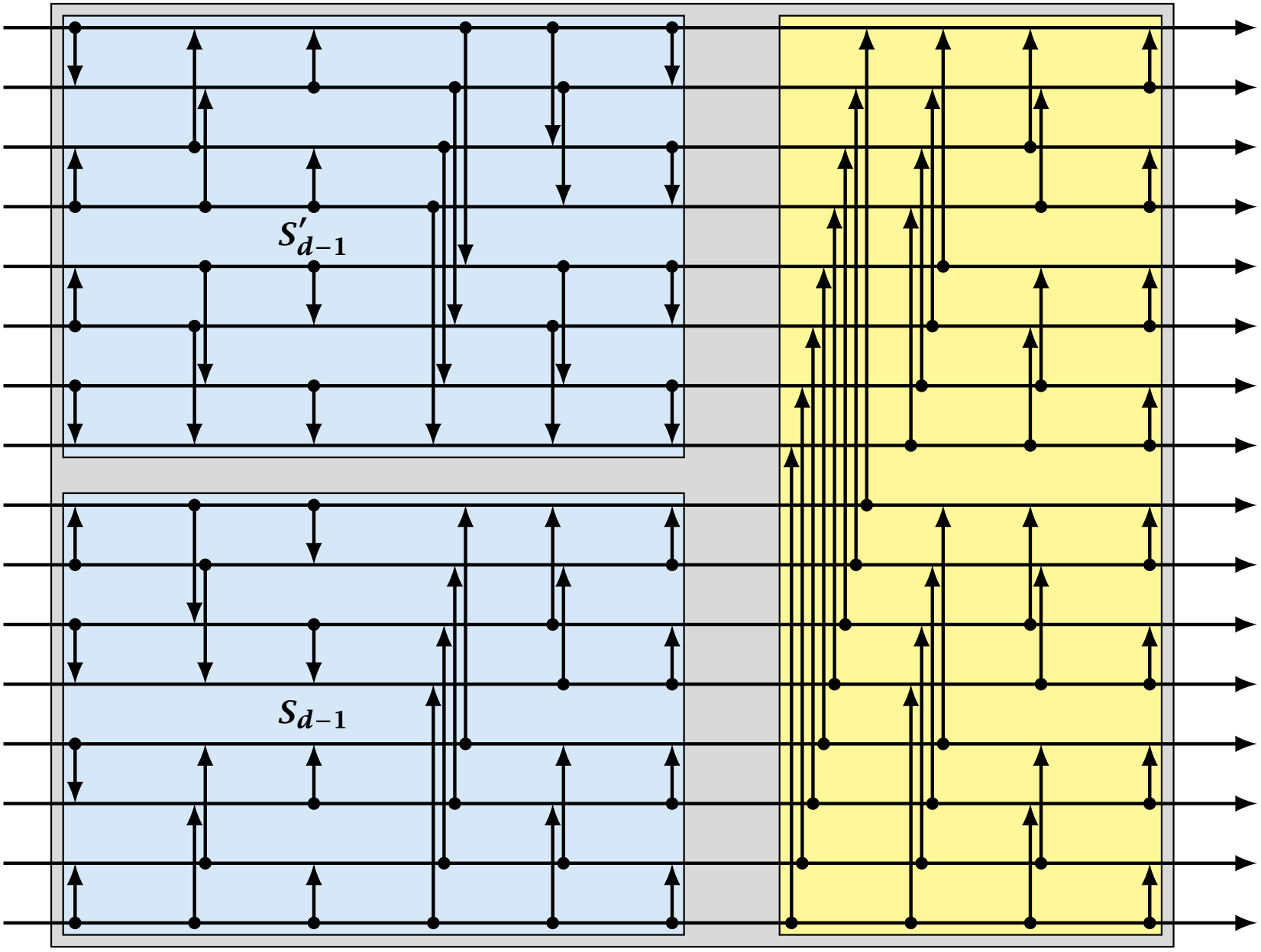
The bitonic merger $B_d$ of dimension $d$ is constructed by combining two bitonic mergers of dimension $d-1$.

If we feed a bitonic $0$-$1$ sequence into this, the sequence will be sorted.

(actually, any bitonic sequence will be sorted, but we do not prove this)



# Bitonic Sorter $S_d$



**Bitonic Merger: ($n = 2^d$)**

- comparators: $C(n) = 2C(n/2) + n/2 \Rightarrow C(n) = \mathcal{O}(n \log n)$.
- depth: $D(n) = D(n/2) + 1 \Rightarrow D(d) = \mathcal{O}(\log n)$.

**Bitonic Sorter: ($n = 2^d$)**

- comparators: $C(n) = 2C(n/2) + \mathcal{O}(n \log n) \Rightarrow$ $C(n) = \mathcal{O}(n \log^2 n)$.
- depth: $D(n) = D(n/2) + \log n \Rightarrow D(n) = \Theta(\log^2 n)$.

# Odd-Even Merge

How to merge two sorted sequences?

$A = (a_1, a_2, \ldots, a_n)$, $B = (b_1, b_2, \ldots, b_n)$, $n$ even.

Split into odd and even sequences:

$A_{\text{odd}} = (a_1, a_3, a_5, \ldots, a_{n-1})$, $A_{\text{even}} = (a_2, a_4, a_6, \ldots a_n)$

$B_{\text{odd}} = (b_1, b_3, b_5, \ldots, b_{n-1})$, $B_{\text{even}} = (b_2, b_4, b_6, \ldots, b_n)$

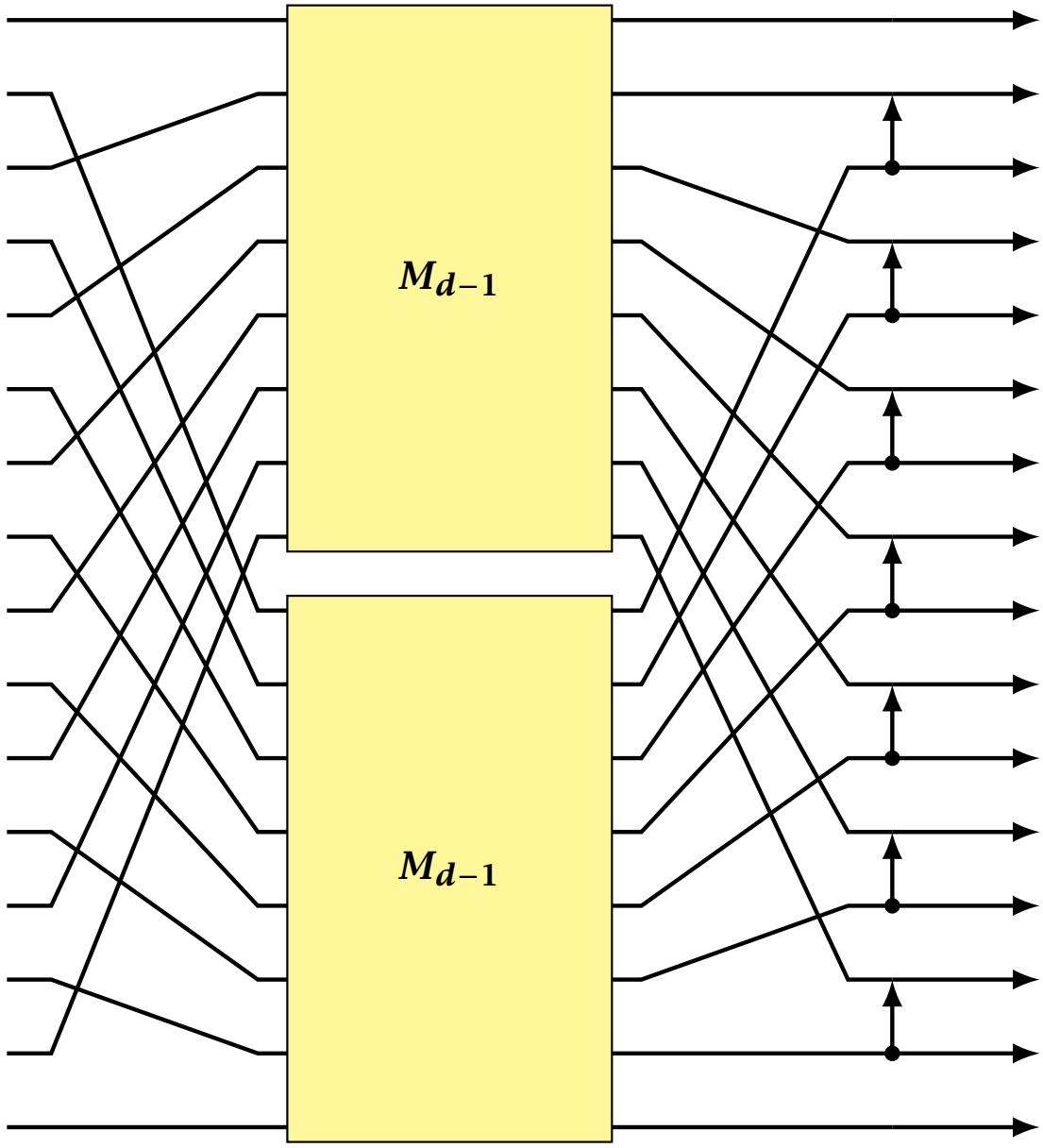
Let

$$X = \text{merge}(A_{\text{odd}}, B_{\text{odd}}) \text{ and } Y = \text{merge}(A_{\text{even}}, B_{\text{even}})$$

Then

$$S = (x_1, \min\{x_2, y_1\}, \max\{x_2, y_1\}, \min\{x_3, y_2\}, \ldots, y_n)$$

# Odd-Even Merge



**Theorem 28**

*There exists a sorting network with depth $\mathcal{O}(\log n)$ and $\mathcal{O}(n \log n)$ comparators.*

# Parallel Comparison Tree Model

A parallel comparison tree (with parallelism $p$) is a $3^p$-ary tree.

- each internal node represents a set of $p$ comparisons btw. $p$ pairs (not necessarily distinct)
- a leaf $v$ corresponds to a unique permutation that is valid for all the comparisons on the path from the root to $v$
- the number of parallel steps is the height of the tree

# Comparison PRAM

A comparison PRAM is a PRAM where we can only compare the input elements;

- we cannot view them as strings
- we cannot do calculations on them

A lower bound for the comparison tree with parallelism $p$ directly carries over to the comparison PRAM with $p$ processors.

# A Lower Bound for Searching

**Theorem 29**

*Given a sorted table $X$ of $n$ elements and an element $y$. Searching for $y$ in $X$ requires $\Omega(\frac{\log n}{\log(p+1)})$ steps in the parallel comparsion tree with parallelism $p < n$.*

# A Lower Bound for Maximum

**Theorem 30**

*A graph $G$ with $m$ edges and $n$ vertices has an independent set on at least $\frac{n^2}{2m+n}$ vertices.*

**base case ($n = 1$)**

- The only graph with one vertex has $m = 0$, and an independent set of size 1.

**induction step ($1, \ldots, n \to n+1$)**

- Let $G$ be a graph with $n+1$ vertices, and $v$ a node with minimum degree ($d$).
- Let $G'$ be the graph after deleting $v$ and its adjacent vertices in $G$.
- $n' = n - (d+1)$
- $m' \leq m - \frac{d}{2}(d+1)$ as we remove $d+1$ vertices, each with degree at least $d$
- In $G'$ there is an independent set of size $((n')^2/(2m'+n'))$.
- By adding $v$ we obtain an indepent set of size

$$1 + \frac{(n')^2}{2m'+n'} \geq \frac{n^2}{2m+n}$$

# A Lower Bound for Maximum

**Theorem 31**

*Computing the maximum of $n$ elements in the comparison tree requires $\Omega(\log\log n)$ steps whenever the degree of parallelism is $p \leq n$.*

**Theorem 32**

*Computing the maximum of $n$ elements requires $\Omega(\log\log n)$ steps on the comparison PRAM with $n$ processors.*

An adversary can specify the input such that at the end of the $(i+1)$-st step the maximum lies in a set $C_{i+1}$ of size $s_{i+1}$ such that

- no two elements of $C_{i+1}$ have been compared
- $s_{i+1} \geq \frac{s_i^2}{2p+c_i}$

**Theorem 33**

*The selection problem requires* $\Omega(\log n / \log\log n)$ *steps on a comparison PRAM.*

not proven yet

# A Lower Bound for Merging

The $(k, s)$-merging problem, asks to merge $k$ pairs of subsequences $A^1, \ldots, A^k$ and $B^1, \ldots, B^k$ where we know that all elements in $A^i \cup B^i$ are smaller than elements in $A^j \cup B^j$ for $(i < j)$.

# A Lower Bound for Merging

**Lemma 34**

*Suppose we are given a parallel comparison tree with parallelism $p$ to solve the $(k,s)$ merging problem. After the first step an adversary can specify the input such that an arbitrary $(k',s')$ merging problem has to be solved, where*

$$k' = \frac{3}{4}\sqrt{pk}$$

$$s' = \frac{s}{4}\sqrt{\frac{k}{p}}$$

# A Lower Bound for Merging

Partition $A^i s$ and $B^i s$ into blocks of length roughly $s/\ell$; hence $\ell$ blocks.

Define an $\ell \times \ell$ binary matrix $M^i$, where $M^i_{xy}$ is $0$ iff the parallel step **did not** compare an element from $A^i_x$ with an element from $B^i_y$.

The matrix has $2\ell - 1$ diagonals.

Choose for every $i$ the diagonal of $M^i$ that has most zeros.

Pair all $A^i_{j+d_i}, B^i_j$, (where $d_i \in \{-(\ell-1), \ldots, \ell-1\}$ specifies the chosen diagonal) for which the entry in $M^i$ is zero.

We can choose value s.t. elements for the $j$-th pair along the diagonal are **all** smaller than for the $(j+1)$-th pair.

Hence, we get a $(k', s')$ problem.

## How many pairs do we have?

- there are $k\ell$ blocks in total
- there are $k \cdot \ell^2$ matrix entries in total
- there are at least $k \cdot \ell^2 - p$ zeros.
- choosing a random diagonal (same for every matrix $M^i$) hits at least

$$\frac{k\ell^2 - p}{2\ell - 1} \geq \frac{k\ell}{2} - \frac{p}{2\ell}$$

  zeroes.
- Choosing $\ell = 2\sqrt{\frac{p}{k}}$ gives

$$k' \geq \frac{3}{4}\sqrt{pk} \text{ and } s' = \lfloor \frac{s}{\ell} \rfloor \geq \frac{s}{2\ell} = \frac{s}{4}\sqrt{\frac{k}{p}}$$

  where we assume $\frac{s}{\ell} \geq 2$.

**Lemma 35**

*Let $T(k,s,p)$ be the number of parallel steps required on a comparison tree to solve the $(k,s)$ merging problem. Then*

$$T(k,p,s) \geq \frac{1}{4}\log\frac{\log\frac{p}{k}}{\log\frac{p}{ks}}$$

*provided that $p \geq 2ks$ and $p \leq ks^2/4$*

**Induction Step:**

Assume that

$$
\begin{aligned}
T(k',s',p) &\geq \frac{1}{4}\log\frac{\log\frac{p}{k'}}{\log\frac{p}{k's'}} \\
&\geq \frac{1}{4}\log\frac{\log\frac{4}{3}\sqrt{\frac{p}{k}}}{\log\frac{16}{3}\frac{p}{ks}} \\
&\geq \frac{1}{4}\log\frac{\frac{1}{2}\log\frac{p}{k}}{7\log\frac{p}{ks}} \\
&\geq \frac{1}{4}\log\frac{\log\frac{p}{k}}{\log\frac{p}{ks}} - 1
\end{aligned}
$$

This gives the induction step.

**Theorem 36**

*Merging requires at least $\Omega(\log\log n)$ time on a CRCW PRAM with $n$ processors.*

# Simulations between PRAMs

**Theorem 37**

*We can simulate a $p$-processor priority CRCW PRAM on a $p$-processor EREW PRAM with slowdown $\mathcal{O}(\log p)$.*

# Simulations between PRAMs

**Theorem 38**

*We can simulate a $p$-processor priority CRCW PRAM on a $p \log p$-processor common CRCW PRAM with slowdown $\mathcal{O}(1)$.*

# Simulations between PRAMs

**Theorem 39**

*We can simulate a $p$-processor priority CRCW PRAM on a $p$-processor common CRCW PRAM with slowdown $\mathcal{O}(\frac{\log p}{\log\log p})$.*

# Simulations between PRAMs

**Theorem 40**

*We can simulate a $p$-processor priority CRCW PRAM on a $p$-processor arbitrary CRCW PRAM with slowdown $\mathcal{O}(\log\log p)$.*

# Lower Bounds for the CREW PRAM

**Ideal PRAM:**

- every processor has unbounded local memory
- in each step a processor reads a global variable
- then it does some (unbounded) computation on its local memory
- then it writes a global variable

# Lower Bounds for the CREW PRAM

**Definition 41**

An input index $i$ affects a memory location $M$ at time $t$ on some input $I$ if the content of $M$ at time $t$ differs between inputs $I$ and $I(i)$ ($i$-th bit flipped).

$L(M,t,I) = \{i \mid i \text{ affects } M \text{ at time } t \text{ on input } I\}$

# Lower Bounds for the CREW PRAM

**Definition 42**

An input index $i$ affects a processor $P$ at time $t$ on some input $I$ if the state of $P$ at time $t$ differs between inputs $I$ and $I(i)$ ($i$-th bit flipped).

$K(P,t,I) = \{i \mid i \text{ affects } P \text{ at time } t \text{ on input } I\}$

# Lower Bounds for the CREW PRAM

**Lemma 43**

*If $i \in K(P,t,I)$ with $t > 1$ then either*

- $i \in K(P,t-1,I)$, *or*
- $P$ *reads a global memory location* $M$ *on input* $I$ *at time* $t$, *and* $i \in L(M,t-1,I)$.

# Lower Bounds for the CREW PRAM

**Lemma 44**

*If $i \in L(M,t,I)$ with $t > 1$ then either*

- *A processor writes into $M$ at time $t$ on input $I$ and $i \in K(P,t,I)$, or*
- *No processor writes into $M$ at time $t$ on input $I$ and*
  - *either $i \in L(M,t-1,I)$*
  - *or a processor $P$ writes into $M$ at time $t$ on input $I(i)$.*

Let $k_0 = 0, \ell_0 = 1$ and define

$$k_{t+1} = k_t + \ell_t \text{ and } \ell_{t+1} = 3k_t + 4\ell_t$$

**Lemma 45**

*$|K(P,t,I)| \leq k_t$ and $|L(M,t,I)| \leq \ell_t$ for any $t \geq 0$*

**base case ($t = 0$):**

- No index can influence the local memory/state of a processor before the first step (hence $|K(P, 0, I)| = k_0 = 0$).
- Initially every index in the input affects exactly one memory location. Hence $|L(M, 0, I)| = 1 = \ell_0$.

**induction step ($t \rightarrow t+1$):**

$K(P,t+1,I) \subseteq K(P,t,I) \cup L(M,t,I)$, where $M$ is the location read by $P$ in step $t+1$.

Hence,

$$\begin{aligned} |K(P,t+1,I)| &\leq |K(P,t,I)| + |L(M,t,I)| \\ &\leq k_t + \ell_t \end{aligned}$$

**induction step ($t \rightarrow t+1$):**

For the bound on $|L(M, t+1, I)|$ we have two cases.

**Case 1:**
A processor $P$ writes into location $M$ at time $t+1$ on input $I$.

Then,

$$\begin{aligned} |L(M, t+1, I)| &\leq |K(P, t+1, I)| \\ &\leq k_t + \ell_t \\ &\leq 3k_t + \ell_t = \ell_{t+1} \end{aligned}$$

**Case 2:**
No processor $P$ writes into location $M$ at time $t+1$ on input $I$.

An index $i$ affects $M$ at time $t+1$ iff $i$ affects $M$ at time $t$ or some processor $P$ writes into $M$ at $t+1$ on $I(i)$.

$$L(M,t+1,I) \subseteq L(M,t,I) \cup Y(M,t+1,I)$$

$Y(M,t+1,I)$ is the set of indices $u_j$ that cause some processor $P_{w_j}$ to write into $M$ at time $t+1$ on input $I$.

$Y(M, t+1, I)$ is the set of indices $u_j$ that cause some processor $P_{w_j}$ to write into $M$ at time $t+1$ on input $I$.

**Fact:**
For all pairs $u_s$, $u_t$ with $P_{w_s} \neq P_{w_t}$ either
$u_s \in K(P_{w_t}, t+1, I(u_t))$ or $u_t \in K(P_{w_s}, t+1, I(u_s))$.

Otherwise, $P_{w_t}$ and $P_{w_s}$ would both write into $M$ at the same time on input $I(u_s)(u_t)$.

Let $U = \{u_1, \ldots, u_r\}$ denote all indices that cause some processor to write into $M$.

Let $V = \{(I(u_1), P_{w_1}), \ldots\}$.

We set up a bipartite graph between $U$ and $V$, such that $(u_i, (I(u_j), P_{w_j})) \in E$ if $u_i$ affects $P_{w_j}$ at time $t+1$ on input $I(u_j)$.

Each vertex $(I(u_j), P_{w_j})$ has degree at most $k_{t+1}$ as this is an upper bound on indices that can influence a processor $P_{w_j}$.

Hence, $|E| \leq r \cdot k_{t+1}$.

For an index $u_j$ there can be at most $k_{t+1}$ indices $u_i$ with $P_{w_i} = P_{w_j}$.

Hence, there must be at least $\frac{1}{2}r(r-k_{t+1})$ pairs $u_i, u_j$ with $P_{w_i} \neq P_{w_j}$.

Each pair introduces at least one edge.

Hence,

$$|E| \geq \frac{1}{2}r(r-k_{t+1})$$

This gives $r \leq 3k_{t+1} \leq 3k_t + 3\ell_t$

Recall that $L(M, t+1, i) \subseteq L(M, t, i) \cup Y(M, t+1, I)$

$|L(M, t+1, i)| \leq 3k_t + 4\ell_t$

$$\begin{pmatrix} k_{t+1} \\ \ell_{t+1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} k_t \\ \ell_t \end{pmatrix} \qquad \begin{pmatrix} k_0 \\ \ell_0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

Eigenvalues:

$$\lambda_1 = \frac{1}{2}(5 + \sqrt{21}) \text{ and } \lambda_2 = \frac{1}{2}(5 - \sqrt{21})$$

Eigenvectors:

$$v_1 = \begin{pmatrix} 1 \\ -(1 - \lambda_1) \end{pmatrix} \text{ and } v_2 = \begin{pmatrix} 1 \\ -(1 - \lambda_2) \end{pmatrix}$$

$$v_1 = \begin{pmatrix} 1 \\ \frac{3}{2} + \frac{1}{2}\sqrt{21}) \end{pmatrix} \text{ and } v_2 = \begin{pmatrix} 1 \\ \frac{3}{2} - \frac{1}{2}\sqrt{21} \end{pmatrix}$$

$$v_1 = \begin{pmatrix} 1 \\ \frac{3}{2} + \frac{1}{2}\sqrt{21}) \end{pmatrix} \text{ and } v_2 = \begin{pmatrix} 1 \\ \frac{3}{2} - \frac{1}{2}\sqrt{21} \end{pmatrix}$$

$$\begin{pmatrix} k_0 \\ \ell_0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \frac{1}{\sqrt{21}}(v_1 - v_2)$$

$$\begin{pmatrix} k_t \\ \ell_t \end{pmatrix} = \frac{1}{\sqrt{21}}\left(\lambda_1^t v_1 - \lambda_2^t v_2\right)$$

Solving the recurrence gives

$$k_t = \frac{\lambda_1^t}{\sqrt{21}} - \frac{\lambda_2^t}{\sqrt{21}}$$

$$\ell_t = \frac{3+\sqrt{21}}{2\sqrt{21}}\lambda_1^t + \frac{-3+\sqrt{21}}{2\sqrt{21}}\lambda_2^t$$

with $\lambda_1 = \frac{1}{2}(5+\sqrt{21})$ and $\lambda_2 = \frac{1}{2}(5-\sqrt{21})$.

**Theorem 46**

*The following problems require logarithmic time on a CREW PRAM.*

- *Sorting a sequence of $x_1, \ldots, x_n$ with $x_i \in \{0, 1\}$*
- *Computing the maximum of $n$ inputs*
- *Computing the sum $x_1 + \cdots + x_n$ with $x_i \in \{0, 1\}$*

# A Lower Bound for the EREW PRAM

**Definition 47 (Zero Counting Problem)**

Given a monotone binary sequence $x_1, x_2, \ldots, x_n$ determine the index $i$ such that $x_i = 0$ and $x_{i+1} = 1$.

We show that this problem requires $\Omega(\log n - \log p)$ steps on a $p$-processor EREW PRAM.

Let $I_i$ be the input with $i$ zeros folled by $n-i$ ones.

Index $i$ affects processor $P$ at time $t$ if the state in step $t$ is differs between $I_{i-1}$ and $I_i$.

Index $i$ affects location $M$ at time $t$ if the content of $M$ after step $t$ differs between inputs $I_{i-1}$ and $I_i$.

**Lemma 48**

*If $i \in K(P,t)$ then either*

- $i \in K(P,t-1)$, *or*
- $P$ *reads some location* $M$ *on input* $I_i$ *(and, hence, also on* $I_{i-1}$*) at step* $t$ *and* $i \in L(M,t-1)$

**Lemma 49**

*If $i \in L(M,t)$ then either*

- $i \in L(M,t-1)$, *or*
- *Some processor $P$ writes $M$ at step $t$ on input $I_i$ and $i \in K(P,t)$.*
- *Some processor $P$ writes $M$ at step $t$ on input $I_{i-1}$ and $i \in K(P,t)$.*

Define

$$C(t) = \sum_P |K(P,t)| + \sum_M \max\{0, |L(M,t)| - 1\}$$

$C(T) \geq n$, $C(0) = 0$

**Claim:**
$C(t) \leq 6C(t-1) + 3|P|$

This gives $C(T) \leq \frac{6^T - 1}{5} 3|P|$ and hence $T = \Omega(\log n - \log |P|)$.

For an index $i$ to newly appear in $L(M,t)$ some processor must write into $M$ on either input $I_i$ or $I_{i-1}$.

Hence, any index in $K(P,t)$ can at most generate two new indices in $L(M,t)$.

This means that the number of new indices in any set $L(M,t)$ (over all $M$) is at most

$$2\sum_{P}|K(P,t)|$$

Hence,

$$\sum_M |L(M,t)| \leq \sum_M |L(M,t-1)| + 2\sum_P |K(P,t)|$$

We can assume wlog. that $L(M,t-1) \subseteq L(M,t)$. Then

$$\sum_M \max\{0, |L(M,t)| - 1\} \leq \sum_M \max\{0, |L(M,t-1)| - 1\} + 2\sum_P |K(P,t)|$$

For an index $i$ to newly appear in $K(P,t)$, $P$ must read a memory location $M$ with $i \in L(M,t)$ on input $I_i$ (and also on input $I_{i-1}$).

Since we are in the EREW model at most one processor can do so in every step.

Let $J(i,t)$ be memory locations read in step $t$ on input $I_i$, and let $J_t = \bigcup_i J(i,t)$.

$$\sum_P |K(P,t)| \leq \sum_P |K(P,t-1)| + \sum_{M \in J_t} |L(M,t-1)|$$

Over all inputs $I_i$ a processor can read at most $|K(P,t-1)| + 1$ different memory locations (why?).

Hence,

$$\begin{aligned}
\sum_P |K(P,t)| &\le \sum_P |K(P,t-1)| + \sum_{M\in J_t} |L(M,t-1)| \\
&\le \sum_P |K(P,t-1)| + \sum_{M\in J_t} (|L(M,t-1)|-1) + J_t \\
&\le 2\sum_P |K(P,t-1)| + \sum_{M\in J_t} (|L(M,t-1)|-1) + |P| \\
&\le 2\sum_P |K(P,t-1)| + \sum_M \max\{0,|L(M,t-1)|-1\} + |P|
\end{aligned}$$

Recall

$$\sum_M \max\{0,|L(M,t)|-1\} \le \sum_M \max\{0,|L(M,t-1)|-1\} + 2\sum_P |K(P,t)|$$

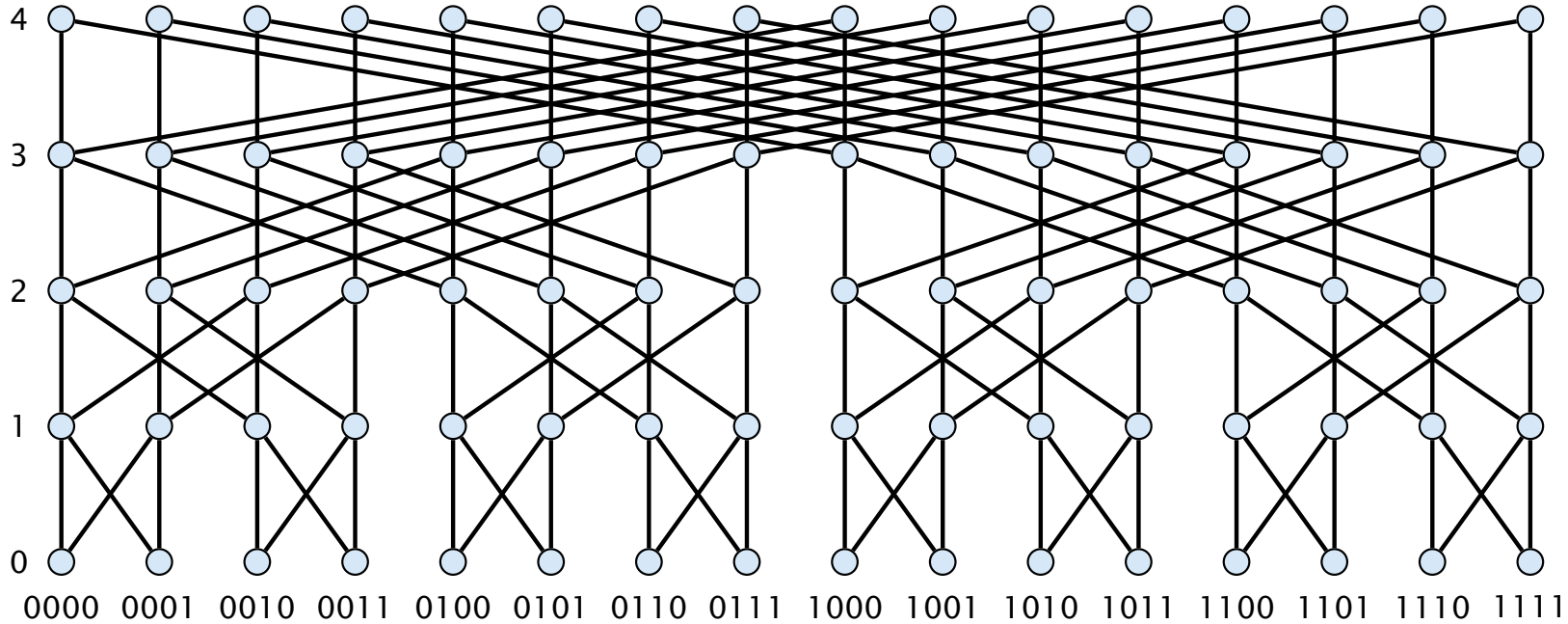
This gives

$$\sum_P K(P,t) + \sum_M \max\{0, |L(M,t)| - 1\}$$
$$\leq 4 \sum_M \max\{0, |L(M,t-1)| - 1\} + 6 \sum_P |K(P,t-1)| + 3|P|$$

Hence,

$$C(t) \leq 6C(t-1) + 3|P|$$

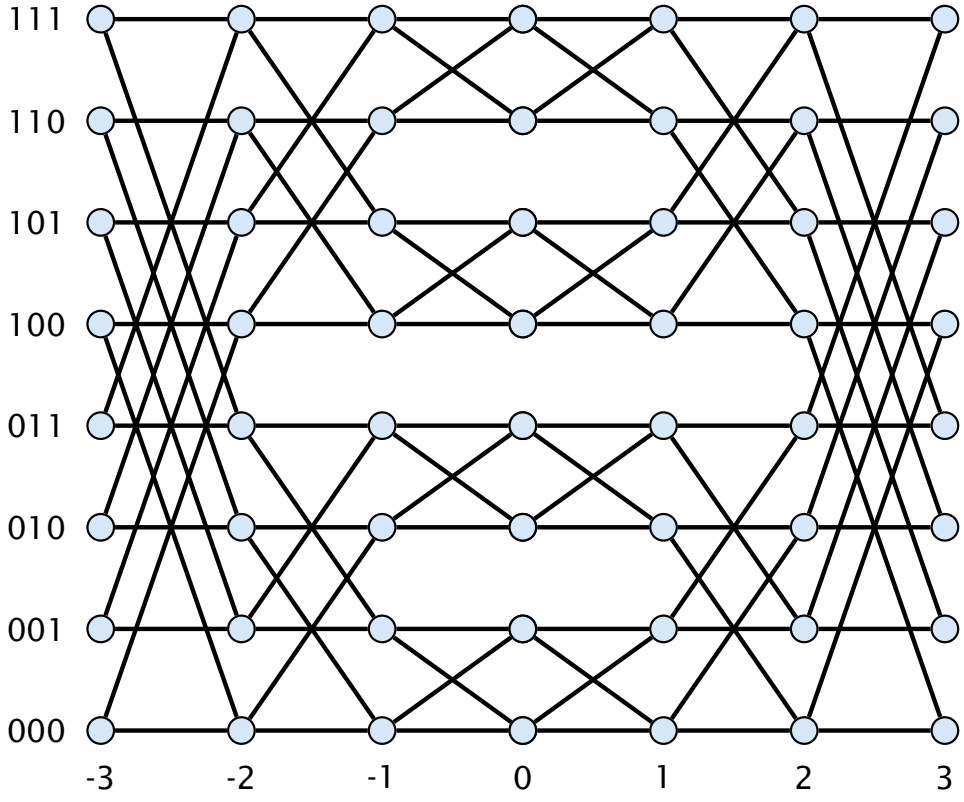
# Bufferfly Network $\mathrm{BF}(d)$




- node set $V = \{(\ell, \bar{x}) \mid \bar{x} \in [2]^d, \ell \in [d+1]\}$, where $\bar{x} = x_0\, x_1 \ldots x_{d-1}$ is a bit-string of length $d$
- edge set $E = \{\{(\ell, \bar{x}), (\ell+1, \bar{x}')\} \mid \ell \in [d], \bar{x} \in [2]^d, x'_i = x_i \text{ for } i \neq \ell\}$

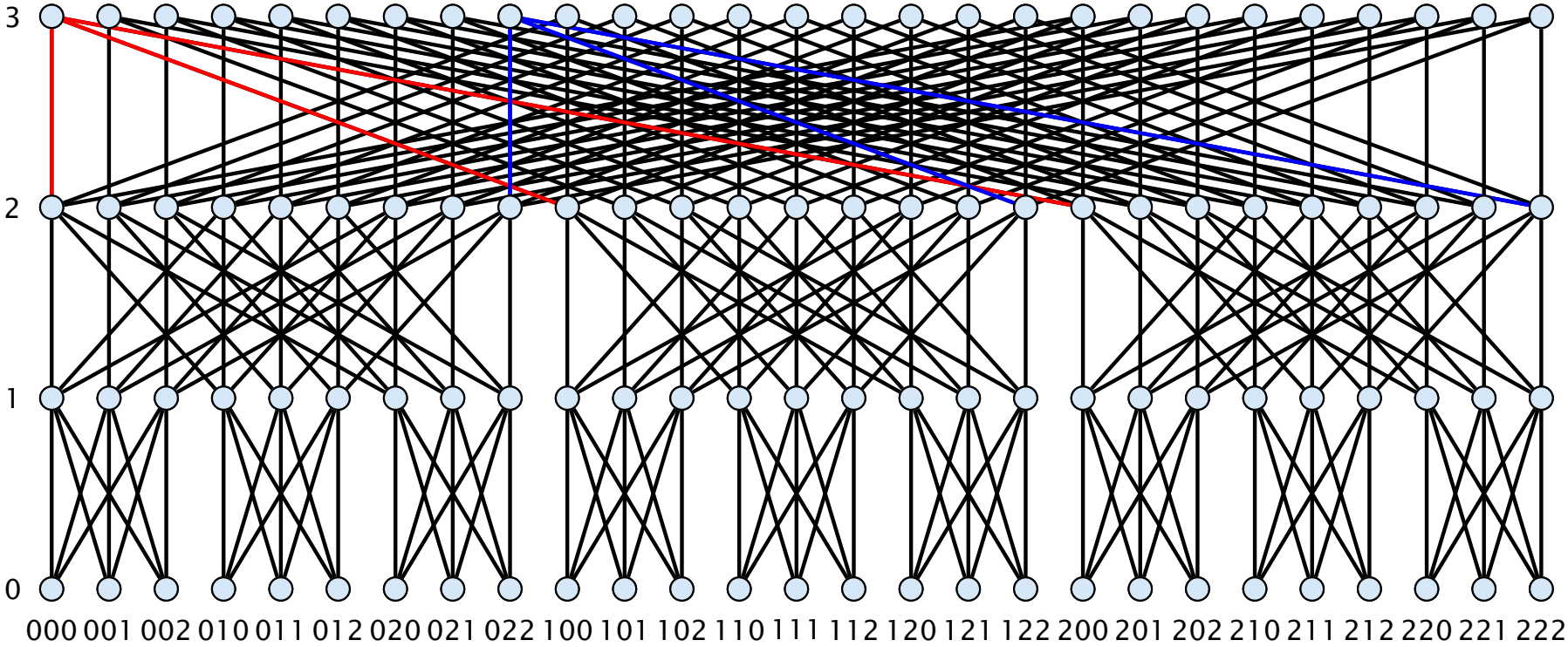
Sometimes the first and last level are identified.

# Beneš Network



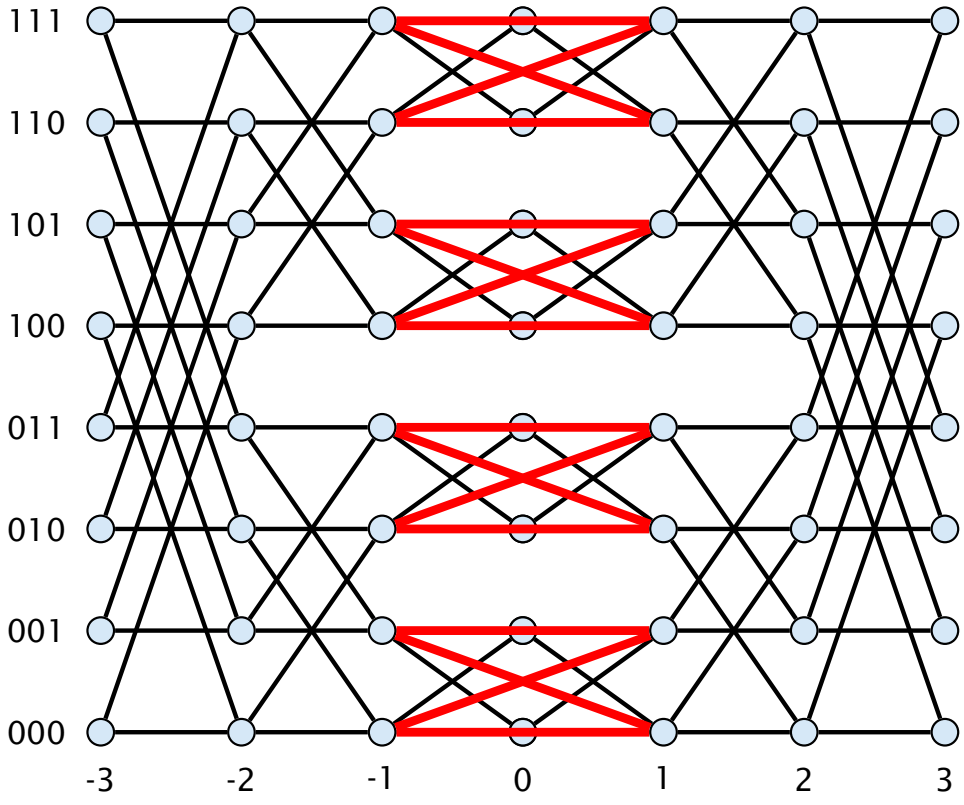

- node set $V = \{(\ell, \bar{x}) \mid \bar{x} \in [2]^d, \ell \in \{-d, \ldots, d\}\}$
- edge set
  $E = \{\{(\ell, \bar{x}), (\ell + 1, \bar{x}')\} \mid \ell \in [d], \bar{x} \in [2]^d, x'_i = x_i \text{ for } i \neq \ell\}$
  $\cup \{\{(-\ell, \bar{x}), (\ell - 1, \bar{x}')\} \mid \ell \in [d], \bar{x} \in [2]^d, x'_i = x_i \text{ for } i \neq \ell\}$

# $n$-ary Bufferfly Network $\mathrm{BF}(n, d)$



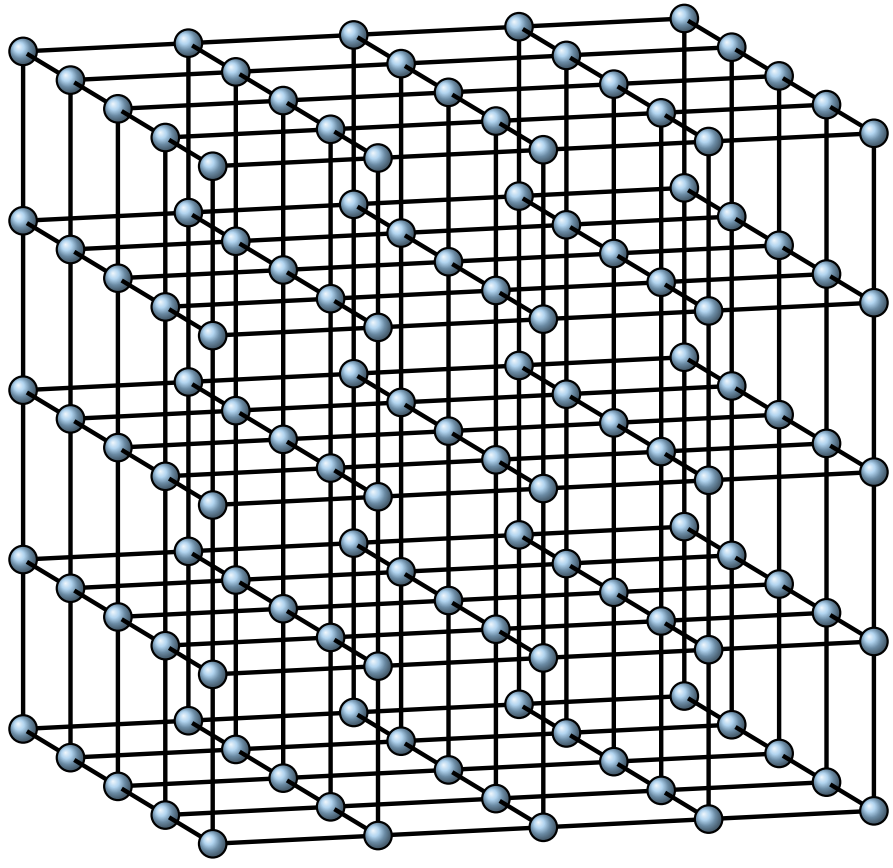

- node set $V = \{(\ell, \bar{x}) \mid \bar{x} \in [n]^d, \ell \in [d+1]\}$, where $\bar{x} = x_0\, x_1\, \ldots x_{d-1}$ is a bit-string of length $d$
- edge set $E = \{\{(\ell, \bar{x}), (\ell+1, \bar{x}')\} \mid \ell \in [d], \bar{x} \in [n]^d, x'_i = x_i \text{ for } i \neq \ell\}$

# Permutation Network $\mathrm{PN}(n, d)$




- There is an $n$-ary version of the Benes network ($2$ $n$-ary butterflies glued at level $0$).
- identifying levels $0$ and $1$ (or $0$ and $-1$) gives $\mathrm{PN}(n, d)$.

# The $d$-dimensional mesh $M(n, d)$



- node set $V = [n]^d$
- edge set $E = \{\{(x_0, \ldots, x_i, \ldots, x_{d-1}), (x_0, \ldots, x_i + 1, \ldots, x_{d-1})\} \mid x_s \in [n] \text{ for } s \in [d] \setminus \{i\}, x_i \in [n-1]\}$

# Remarks

$M(2,d)$ is also called $d$-dimensional hypercube.

$M(n,1)$ is also called linear array of length $n$.

# Permutation Routing

**Lemma 50**

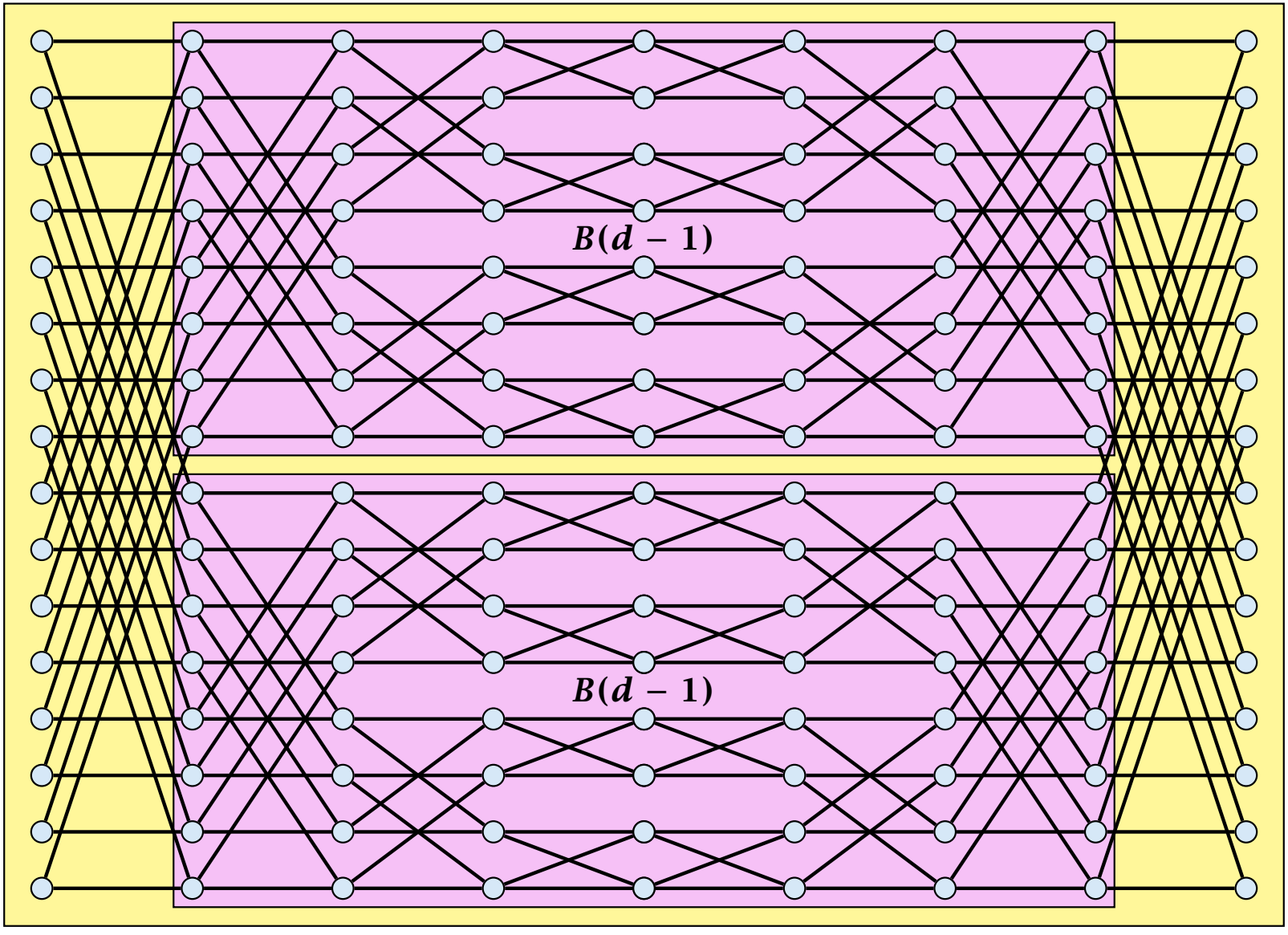
*On the linear array $M(n,1)$ any permutation can be routed online in $2n$ steps with buffersize 3.*

# Permutation Routing

**Lemma 51**

*On the Beneš network any permutation can be routed offline in $2d$ steps between the sources level $(+d)$ and target level $(-d)$.*

# Recursive Beneš Network



# Permutation Routing

**base case $d = 0$**

trivial

**induction step $d \to d + 1$**

- The packets that start at $(\bar{a}, d)$ and $(\bar{a}(d), d)$ have to be sent into different sub-networks.
- The packets that end at $(\bar{a}, -d)$ and $(\bar{a}(d), -d)$ have to come out of different sub-networks.

We can generate a graph on the set of packets.

- Every packet has an incident source edge (connecting it to the conflicting start packet)
- Every packet has an incident target edge (connecting it to the conflicting packet at its target)
- This clearly gives a bipartite graph; Coloring this graph tells us which packet to send into which sub-network.

# Permutation Routing on the $n$-ary Beneš Network

Instead of two we have $n$ sub-networks $B(n, d-1)$.

All packets starting at positions $\{(x_0, \ldots, x_i, \ldots, x_{d-1}, d) \mid x_i \in [n]\}$ have to be send to different sub-networks.

All packets ending at positions $\{(x_0, \ldots, x_i, \ldots, x_{d-1}, d) \mid x_i \in [n]\}$ have to come from different sub-networks.

The conflict graph is a $n$-uniform $2$-regular hypergraph.

We can color such a graph with $n$ colors such that no two nodes in a hyperedge share a color.

This gives the routing.

**Lemma 52**

*On a $d$-dimensional mesh with sidelength $n$ we can route any permutation (offline) in $4dn$ steps.*

We can simulate the algorithm for the $n$-ary Beneš Network.

Each step can be simulated by routing on disjoint linear arrays.
This takes at most $2n$ steps.

We simulate the behaviour of the Beneš network on the $n$-dimensional mesh.

In round $r \in \{-d, \ldots, -1, 0, 1, \ldots, d-1\}$ we simulate the step of sending from level $r$ of the Beneš network to level $r+1$.

Each node $\bar{x} \in [n]^d$ of the mesh simulates the node $(r, \bar{x})$.

Hence, if in the Beneš network we send from $(r, \bar{x})$ to $(r+1, \bar{x}')$ we have to send from $\bar{x}$ to $\bar{x}'$ in the mesh.

All communication is performed along linear arrays. In round $r < 0$ the linear arrays along dimension $-r-1$ (recall that dimensions are numbered from $0$ to $d-1$) are used

$$\bar{x}_{d-1} \ldots \bar{x}_{-r} \alpha \bar{x}_{-r-2} \ldots \bar{x}_0$$

In rounds $r \geq 0$ linear arrays along dimension $r$ are used.

Hence, we can perform a round in $\mathcal{O}(n)$ steps.

**Lemma 53**

*We can route any permutation on the Beneš network in $\mathcal{O}(d)$ steps with constant buffer size.*

The same is true for the butterfly network.

The nodes are of the form $(\ell, \bar{x})$, $\bar{x} \in [n]^d, \ell \in -d, \ldots, d$.

We can view nodes with same first coordinate forming columns and nodes with the same second coordinate as forming rows. This gives rows of length $2d+1$ and columns of length $n^d$.

We route in 3 phases:

1. Permute packets along the rows such that afterwards no column contains packets that have the same target row. $\mathcal{O}(d)$ steps.
2. We can use pipeling to permute **every** column, so that afterwards every packet is in its target row. $\mathcal{O}(2d+2d)$ steps.
3. Every packet is in its target row. Permute packets to their right destinations. $\mathcal{O}(d)$ steps.

**Lemma 54**

*We can do offline permutation routing of (partial) permutations in $2d$ steps on the hypercube.*

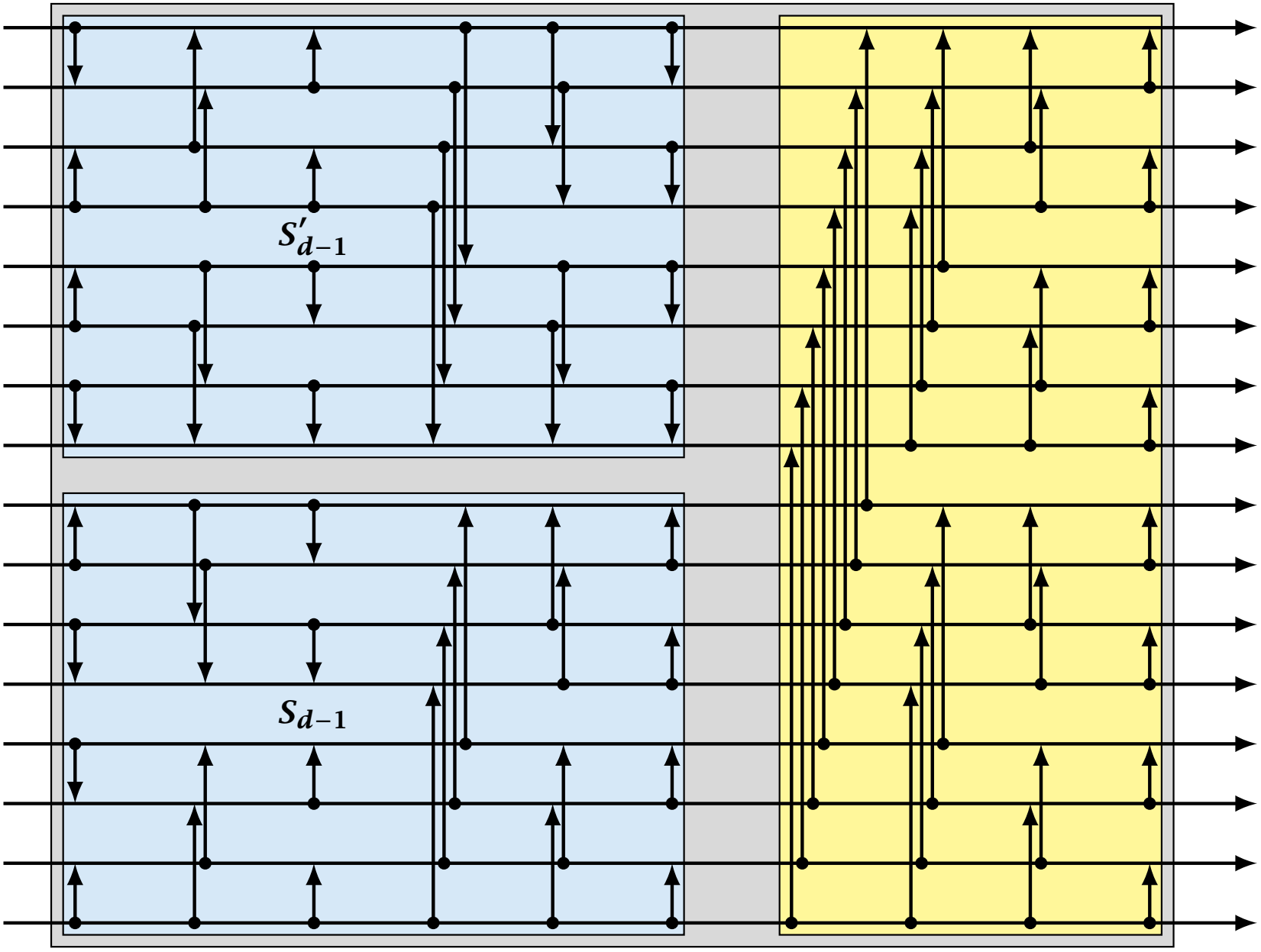
**Lemma 55**

*We can sort on the hypercube $M(2,d)$ in $\mathcal{O}(d^2)$ steps.*

**Lemma 56**

*We can do online permutation routing of permutations in $\mathcal{O}(d^2)$ steps on the hypercube.*

# Bitonic Sorter $S_d$



# ASCEND/DESCEND Programs

**Algorithm 11** ASCEND(procedure $oper$)

1: **for** $dim = 0$ **to** $d-1$
2: **for all** $\bar{a} \in [2]^d$ **pardo**
3: oper$(\bar{a}, \bar{a}(dim), dim)$

**Algorithm 11** DESCEND(procedure $oper$)

1: **for** $dim = d-1$ **to** $0$
2: **for all** $\bar{a} \in [2]^d$ **pardo**
3: oper$(\bar{a}, \bar{a}(dim), dim)$

oper should only depend on the dimension and on values stored in the respective processor pair $(\bar{a}, \bar{a}(dim), V[\bar{a}], V[\bar{a}(dim)])$.

oper should take constant time.

**Algorithm 11** oper$(a, a', dim, T_a, T_{a'})$

1: **if** $a_{dim}, \ldots, a_0 = 0^{dim+1}$ **then**
2: $\quad T_a = \min\{T_a, T_{a'}\}$

Performing an ASCEND run with this operation computes the minimum in processor $0$.

We can sort on $M(2, d)$ by using $d$ DESCEND runs.

We can do offline permutation routing by using a DESCEND run followed by an ASCEND run.

We can perform an ASCEND/DESCEND run on a linear array $M(2^d, 1)$ in $\mathcal{O}(2^d)$ steps.

The CCC network is obtained from a hypercube by replacing every node by a cycle of degree $d$.

- nodes $\{(\ell, \bar{x}) \mid \bar{x} \in [2]^d, \ell \in [d]\}$
- edges $\{\{(\ell, \bar{x}), (\ell, \bar{x}(\ell)\} \mid x \in [2]^d, \ell \in [d]\}$

**constand degree**

**Lemma 57**

*Let $d = 2^k$. An ASCEND run of a hypercube $M(2, d + k)$ can be simulated on CCC(d) in $\mathcal{O}(d)$ steps.*

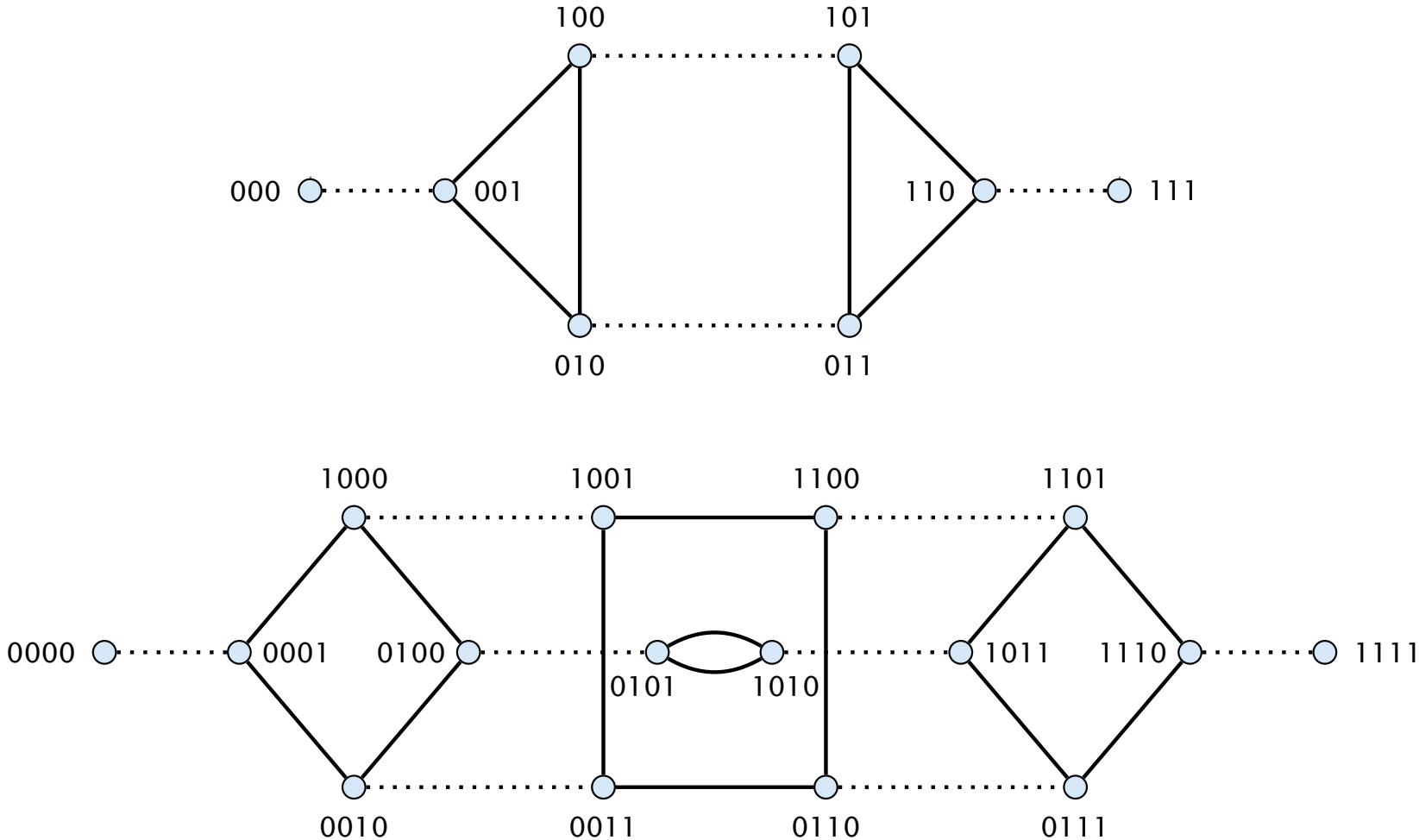
The shuffle exchange network SE($d$) is defined as follows

- nodes: $V = [2]^d$
- edges:
  $E = \left\{\{x\bar{\alpha}, \bar{\alpha}x\} \mid x \in [2], \bar{\alpha} \in [2]^{d-1}\right\} \cup \left\{\{\bar{\alpha}0, \bar{\alpha}1\} \mid \bar{\alpha} \in [2]^{d-1}\right\}$

**constand degree**

Edges of the first type are called shuffle edges. Edges of the second type are called exchange edges

# Shuffle Exchange Networks



**Lemma 58**

*We can perform an ASCEND run of $M(2,d)$ on SE$(d)$ in $\mathcal{O}(d)$ steps.*

# Simulations between Networks

For the following observations we need to make the definition of parallel computer networks more precise.

Each node of a given network corresponds to a processor/RAM.

In addition each processor has a read register and a write register.

In one (synchronous) step each neighbour of a processor $P_i$ can write into $P_i$'s write register or can read from $P_i$'s read register.

Usually we assume that proper care has to be taken to avoid concurrent reads and concurrent writes from/to the same register.

# Simulations between Networks

**Definition 59**

A configuration $C_i$ of processor $P_i$ is the complete description of the state of $P_i$ including local memory, program counter, read-register, write-register, etc.

Suppose a machine $M$ is in configuration $(C_0, \ldots, C_{p-1})$, performs $t$ synchronous steps, and is then in configuration $C = (C'_0, \ldots, C'_{p-1})$.

$C'_i$ is called the $t$-th successor configuration of $C$ for processor $i$.

# Simulations between Networks

**Definition 60**

Let $C = (C_0, \ldots, C_{p-1})$ a configuration of $M$. A machine $M'$ with $q \geq p$ processors weakly simulates $t$ steps of $M$ with slowdown $k$ if

- in the beginning there are $p$ non-empty processors sets $A_0, \ldots, A_{p-1} \subseteq M'$ so that all processors in $A_i$ know $C_i$;
- after at most $k \cdot t$ steps of $M'$ there is a processor $Q^{(i)}$ that knows the $t$-th successors configuration of $C$ for processor $P_i$.

# Simulations between Networks

**Definition 61**

$M'$ simulates $M$ with slowdown $k$ if

- $M'$ weakly simulates machine $M$ with slowdown $k$
- and **every** processor in $A_i$ knows the $t$-th successor configuration of $C$ for processor $P_i$.

We have seen how to simulate an ASCEND/DESCEND run of the hypercube $M(2, d+k)$ on CCC$(d)$ with $d = 2^k$ in $\mathcal{O}(d)$ steps.

Hence, we can simulate $d+k$ steps (one ASCEND run) of the hypercube in $\mathcal{O}(d)$ steps. This means slowdown $\mathcal{O}(1)$.

**Lemma 62**

*Suppose a network $S$ with $n$ processors can route any permutation in time $\mathcal{O}(t(n))$. Then $S$ can simulate any* ***constant degree*** *network $M$ with at most $n$ vertices with slowdown $\mathcal{O}(t(n))$.*

Map the vertices of $M$ to vertices of $S$ in an arbitrary way.

Color the edges of $M$ with $\Delta + 1$ colors, where $\Delta = \mathcal{O}(1)$ denotes the maximum degree.

Each color gives rise to a permutation.

We can route this permutation in $S$ in $t(n)$ steps.

Hence, we can perform the required communication for one step of $M$ by routing $\Delta + 1$ permutations in $S$. This takes time $t(n)$.

A processor of $M$ is simulated by the same processor of $S$ throughout the simulation.

**Lemma 63**

*Suppose a network $S$ with $n$ processors can sort $n$ numbers in time $\mathcal{O}(t(n))$. Then $S$ can simulate any network $M$ with at most $n$ vertices with slowdown $\mathcal{O}(t(n))$.*

**Lemma 64**

*There is a constant degree network on $\mathcal{O}(n^{1+\epsilon})$ nodes that can simulate any constant degree network with slowdown $\mathcal{O}(1)$.*

Suppose we allow concurrent reads, this means in every step all neighbours of a processor $P_i$ can read $P_i$'s read register.

**Lemma 65**

*A constant degree network $M$ that can simulate any $n$-node network has slowdown $\mathcal{O}(\log n)$ (independent of the size of $M$).*

We show the lemma for the following type of simulation.

- There are representative sets $A_i^t$ for every step $t$ that specify which processors of $M$ simulate processor $P_i$ in step $t$ (know the configuration of $P_i$ after the $t$-th step).
- The representative sets for different processors are disjoint.
- for all $i \in \{1, \ldots, n\}$ and steps $t$, $A_i^t \neq \emptyset$.

This is a step-by-step simulation.

Suppose processor $P_i$ reads from processor $P_{j_i}$ in step $t$.

Every processor $Q \in M$ with $Q \in A_i^{t+1}$ must have a path to a processor $Q' \in A_i^t$ and to $Q'' \in A_{j_i}^t$.

Let $k_t$ be the largest distance (maximized over all $i$, $j_i$).

Then the simulation of step $t$ takes time at least $k_t$.

The slowdown is at least

$$k = \frac{1}{\ell} \sum_{t=1}^{\ell} k_t$$

We show

- The simulation of a step takes at least time $\gamma \log n$, or
- the size of the representative sets shrinks by a lot

$$\sum_i |A_i^{t+1}| \leq \frac{1}{n^\epsilon} \sum_i |A_i^t|$$

Suppose there is no pair $(i,j)$ such that $i$ reading from $j$ requires time $\gamma \log n$.

- For every $i$ the set $\Gamma_{2k}(A_i)$ contains a node from $A_j$.
- Hence, there must exist a $j_i$ such that $\Gamma_{2k}(A_i)$ contains at most

$$|C_{j_i}| := \frac{|A_i| \cdot c^{2k}}{n-1} \leq \frac{|A_i| \cdot c^{3k}}{n} \ .$$

processors from $|A_{j_i}|$

If we choose that $i$ reads from $j_i$ we get

$$\begin{aligned} |A_i'| &\leq |C_{j_i}| \cdot c^k \\ &\leq c^k \cdot \frac{|A_i| \cdot c^{3k}}{n} \\ &= \frac{1}{n}|A_i| \cdot c^{4k} \end{aligned}$$

Choosing $k = \Theta(\log n)$ gives that this is at most $|A_i|/n^{\epsilon}$.

Let $\ell$ be the total number of steps and $s$ be the number of short steps when $k_t < \gamma \log n$.

In a step of time $k_t$ a representative set can at most increase by $c^{k_t+1}$.

Let $h_\ell$ denote the number of representatives after step $\ell$.

$$n \leq h_\ell \leq h_0 \Big(\frac{1}{n^\epsilon}\Big)^s \prod_{t \in \text{long}} c^{k_t+1} \leq \frac{n}{n^{\epsilon s}} \cdot c^{\ell + \sum_t k_t}$$

If $\sum_t k_t \geq \ell(\frac{\epsilon}{2}\log_c n - 1)$, we are done. Otw.

$$n \leq n^{1-\epsilon s + \ell\frac{\epsilon}{2}}$$

This gives $s \leq \ell/2$ .

Hence, at most 50% of the steps are short.

# Deterministic Online Routing

**Lemma 66**

*A permutation on an $n \times n$-mesh can be routed online in $\mathcal{O}(n)$ steps.*

# Deterministic Online Routing

**Definition 67 (Oblivious Routing)**

Specify a path-system $\mathcal{W}$ with a path $P_{u,v}$ between $u$ and $v$ for every pair $\{u,v\} \in V \times V$.

A packet with source $u$ and destination $v$ moves along path $P_{u,v}$.

# Deterministic Online Routing

### Definition 68 (Oblivious Routing)

Specify a path-system $\mathcal{W}$ with a path $P_{u,v}$ between $u$ and $v$ for every pair $\{u,v\} \in V \times V$.

### Definition 69 (node congestion)

For a given path-system the node congestion is the maximum number of path that go through any node $v \in V$.

### Definition 70 (edge congestion)

For a given path-system the edge congestion is the maximum number of path that go through any edge $e \in E$.

# Deterministic Online Routing

**Definition 71 (dilation)**

For a given path system the dilation is the maximum length of a path.

**Lemma 72**

*Any oblivious routing protocol requires at least* $\max\{C_f, D_f\}$ *steps, where* $C_f$ *and* $D_f$*, are the congestion and dilation, respectively, of the path-system used. (node congestion or edge congestion depending on the communication model)*

**Lemma 73**

*Any reasonable oblivious routing protocol requires at most* $\mathcal{O}(D_f \cdot C_f)$ *steps (unbounded buffers).*

**Theorem 74 (Borodin, Hopcroft)**

*For any path system $\mathcal{W}$ there exists a permutation $\pi : V \to V$ and an edge $e \in E$ such that at least $\Omega(\sqrt{n}/\Delta)$ of the paths go through $e$.*

Let $\mathcal{W}_v = \{P_{v,u} \mid u \in V\}$.

We say that an edge $e$ is $z$-popular for $v$ if at least $z$ paths from $\mathcal{W}_v$ contain $e$.

For any node $v$ there are many edges that are are quite popular for $v$.

$|V| \times |E|$-matrix $A(z)$:

$$A_{v,e}(z) = \begin{cases} 1 & e \text{ is } z\text{-popular for } v \\ 0 & \text{otherwise} \end{cases}$$

Define

- $$A_v(z) = \sum_e A_{v,e}(z)$$
- $$A_e(z) = \sum_v A_{v,e}(z)$$

**Lemma 75**

*Let $z \leq \frac{n-1}{\Delta}$.*

*For every node $v \in V$ there exist at least $\frac{n}{2\Delta z}$ edges that are $z$ popular for $v$. This means*

$$A_v(z) \geq \frac{n}{2\Delta z}$$

**Lemma 76**

*There exists an edge $e'$ that is $z$-popular for at least $z$ nodes with $z = \Omega(\sqrt{n}\Delta)$.*

$$\sum_e A_e(z) = \sum_v A_v(z) \geq \frac{n^2}{2\Delta z}$$

There must exist an edge $e'$

$$A_{e'}(z) \geq \left\lceil \frac{n^2}{|E| \cdot 2\Delta z} \right\rceil \geq \left\lceil \frac{n}{2\Delta^2 z} \right\rceil$$

where the last step follows from $|E| \leq \Delta n$.

We choose $z$ such that $z = \frac{n}{2\Delta^2 z}$ (i.e., $z = \sqrt{n}/(\sqrt{2}\Delta)$).

This means $e'$ is $\lceil z \rceil$-popular for $\lceil z \rceil$ nodes.

We can construct a permutation such that $z$ paths go through $e'$.

Deterministic oblivious routing may perform very poorly.

What happens if we have a random routing problem in a butterfly?

Suppose every source on level 0 has $p$ packets, that are routed to random destinations.

How many packets go over node $v$ on level $i$?

From $v$ we can reach $2^d/2^i$ different targets.

Hence,

$$\Pr[\text{packet goes over } v] \leq \frac{2^{d-i}}{2^d} = \frac{1}{2^i}$$

Expected number of packets:

$$\mathrm{E}[\text{packets over } v\,] = p \cdot 2^i \cdot \frac{1}{2^i} = p$$

since only $p2^i$ packets can reach $v$.

But this is trivial.

What is the probability that at least $r$ packets go through $v$.

$$\begin{aligned}\Pr[\text{at least } r \text{ path through } v] &\leq \binom{p \cdot 2^i}{r} \cdot \left(\frac{1}{2^i}\right)^r \\ &\leq \left(\frac{p2^i \cdot e}{r}\right)^r \cdot \left(\frac{1}{2^i}\right) \\ &= \left(\frac{pe}{r}\right)^r\end{aligned}$$

$$\begin{aligned}&\Pr[\text{there exists a node } v \text{ sucht that at least } r \text{ path through } v] \\ &\leq d2^d \cdot \left(\frac{pe}{r}\right)^r\end{aligned}$$

$$\Pr[\text{there exists a node } v \text{ sucht that at least } r \text{ path through } v\,]$$
$$\leq d2^d \cdot \left(\frac{pe}{r}\right)^r$$

Choose $r$ as $2ep + (\ell+1)d + \log d = \mathcal{O}(p + \log N)$, where $N$ is number of sources in $\mathsf{BF}(d)$.

$$\Pr[\text{exists node } v \text{ with more than } r \text{ paths over } v\,] \leq \frac{1}{N^\ell}$$

# Scheduling Packets

Assume that in every round a node may forward at most one packet but may receive up to two.

We select a random rank $R_p \in [k]$. Whenever, we forward a packet we choose the packet with smaller rank. Ties are broken according to packet id.

Random Rank Protocol

**Definition 77 (Delay Sequence of length $s$)**

- delay path $\mathcal{W}$
- lengths $\ell_0, \ell_1, \ldots, \ell_s$, with $\ell_0 \geq 1$, $\ell_1, \ldots, \ell_s \geq 0$ lengths of delay-free sub-paths
- collision nodes $v_0, v_1, \ldots, v_s, v_{s+1}$
- collision packets $P_0, \ldots, P_s$

## Properties

- $\text{rank}(P_0) \geq \text{rank}(P_1) \geq \cdots \geq \text{rank}(P_s)$
- $\sum_{i=0}^{s} \ell_i = d$
- if the routing takes $d + s$ steps than the delay sequence has length $s$

## Definition 78 (Formal Delay Sequence)

- a path $\mathcal{W}$ of length $d$ from a source to a target
- $s$ integers $\ell_0 \geq 1, \ell_1, \ldots, \ell_s \geq 0$ and $\sum_{i=0}^{s} \ell_i = d$
- nodes $v_0, \ldots v_s, v_{s+1}$ on $\mathcal{W}$ with $v_i$ being on level $d - \ell_0 - \cdots - \ell_{i-1}$
- $s+1$ packets $P_0, \ldots, P_s$, where $P_i$ is a packet with path through $v_i$ and $v_{i-1}$
- numbers $R_s \leq R_{s-1} \leq \cdots \leq R_0$

We say a formal delay sequence is active if $\mathrm{rank}(P_i) = k_i$ holds for all $i$.

Let $N_s$ be the number of formal delay sequences of length at most $s$. Then

$$\Pr[\text{routing needs at least } d+s \text{ steps}] \leq \frac{N_s}{k^{s+1}}$$

## Lemma 79

$$N_s \leq \left(\frac{2eC(s+k)}{s+1}\right)^{s+1}$$

- there are $N^2$ ways to choose $\mathcal{W}$
- there are $\binom{s+d-1}{s}$ ways to choose $\ell_i$'s with $\sum_{i=0}^{s} \ell_i = d$
- the collision nodes are fixed
- there are at most $C^{s+1}$ ways to choose the collision packets where $C$ is the node congestion
- there are at most $\binom{s+k}{s+1}$ ways to choose $0 \leq k_s \leq \cdots \leq k_0 < k$

Hence the probability that the routing takes more than $d + s$ steps is at most

$$N^3 \cdot \left( \frac{2e \cdot C \cdot (s+k)}{(s+1)k} \right)^{s+1}$$

We choose $s = 8eC - 1 + (\ell + 3)d$ and $k = s + 1$. This gives that the probability is at most $\frac{1}{N^\ell}$.

- With probability $1 - \frac{1}{N^{\ell_1}}$ the random routing problem has congestion at most $\mathcal{O}(p + \ell_1 d)$.
- With probability $1 - \frac{1}{N^{\ell_2}}$ the packet scheduling finishes in at most $\mathcal{O}(C + \ell_2 d)$ steps.

Hence, with high probability routing random problems with $p$ packets per source in a butterfly requires only $\mathcal{O}(p + d)$ steps.

What do we do for arbitrary routing problems?

# Valiants Trick

Where did the scheduling analysis use the butterfly?

We only used

- all routing paths are of the same length $d$
- there are a polynomial number of delay paths

Choose paths as follows:

- route from source to random destination on target level
- route to real target column (albeit on source level)
- route to target

All phases run in time $\mathcal{O}(p + d)$ with high probability.

# Valiants Trick

**Multicommodity Flow Problem**

- undirected (weighted) graph $G = (V, E, c)$
- commodities $(s_i, t_i)$, $i \in \{1, \ldots, k\}$
- a multicommodity flow is a flow $f : E \times \{1, \ldots, k\} \to \mathbb{R}^+$
  - for all edges $e \in E$: $\sum_i f_i(e) \leq c(e)$
  - for all nodes $v \in V \setminus \{s_i, t_i\}$:
    $\sum_{u:(u,v)\in E} f_i((u,v)) = \sum_{w:(v,w)\in E} f_i((v,w))$

**Goal A** (Maximum Multicommodity Flow)
maximize $\sum_i \sum_{e=(s_i,x)\in E} f_i(e)$

**Goal B** (Maximum Concurrent Multicommodity Flow)
maximize $\min_i \sum_{e=(s_i,x)\in E} f_i(e)/d_i$ (throughput fraction), where $d_i$ is demand for commodity $i$

# Valiants Trick

A Balanced Multicommodity Flow Problem is a concurrent multicommodity flow problem in which incoming and outgoing flow is equal to

$$c(v) = \sum_{e=(v,x)\in E} c(e)$$

# Valiants Trick

For a multicommodity flow $S$ we assume that we have a decomposition of the flow(s) into flow-paths.

We use $C(S)$ to denote the congestion of the flow problem (inverse of througput fraction), and $D(S)$ the length of the longest routing path.

For a network $G = (V, E, c)$ we define the characteristic flow problem via

- demands $d_{u,v} = \frac{c(u)c(v)}{c(V)}$

Suppose the characteristic flow problem has a solution $S$ with $C(S) \leq F$ and $D(S) \leq F$.

**Definition 80**

A (randomized) oblivious routing scheme is given by a path system $\mathcal{P}$ and a weight function $w$ such that

$$\sum_{p \in \mathcal{P}_{s,t}} w(p) = 1$$

Construct an oblivious routing scheme from $S$ as follows:

- let $f_{x,y}$ be the flow between $x$ and $y$ in $S$
- $$f_{x,y} \geq d_{x,y}/C(S) \geq d_{x,y}/F = \frac{1}{F}\frac{c(x)c(y)}{c(V)}$$
- for $p \in \mathcal{P}_{x,y}$ set $w(p) = f_p/f_{x,y}$

gives an oblivious routing scheme.

# Valiants Trick

We apply this routing scheme twice:

- first choose a path from $\mathcal{P}_{s,v}$, where $v$ is chosen uniformly according to $c(v)/c(V)$
- then choose path according to $\mathcal{P}_{v,t}$

If the input flow problem/packet routing problem is balanced doing this randomization results in flow solution $S$ (twice).

Hence, we have an oblivious scheme with congestion and dilation at most $2F$ for (balanced inputs).

Example: hypercube.

# Oblivious Routing for the Mesh

We can route any permutation on an $n \times n$ mesh in $\mathcal{O}(n)$ steps, by $x$-$y$ routing. Actually $\mathcal{O}(d)$ steps where $d$ is the largest distance between a source-target pair.

What happens if we do not have a permutation?

$x - y$ routing may generate large congestion if some pairs have a lot of packets.

Valiants trick may create a large dilation.

Let for a multicommodity flow problem $P$ $C_{\text{opt}}(P)$ be the optimum congestion, and $D_{\text{opt}}(P)$ be the optimum dilation (by perhaps different flow solutions).

**Lemma 81**

*There is an oblivious routing scheme for the mesh that obtains a flow solution $S$ with $C(S) = \mathcal{O}(C_{\text{opt}}(P) \log n)$ and $D(S) = \mathcal{O}(D_{\text{opt}}(P))$.*

**Lemma 82**

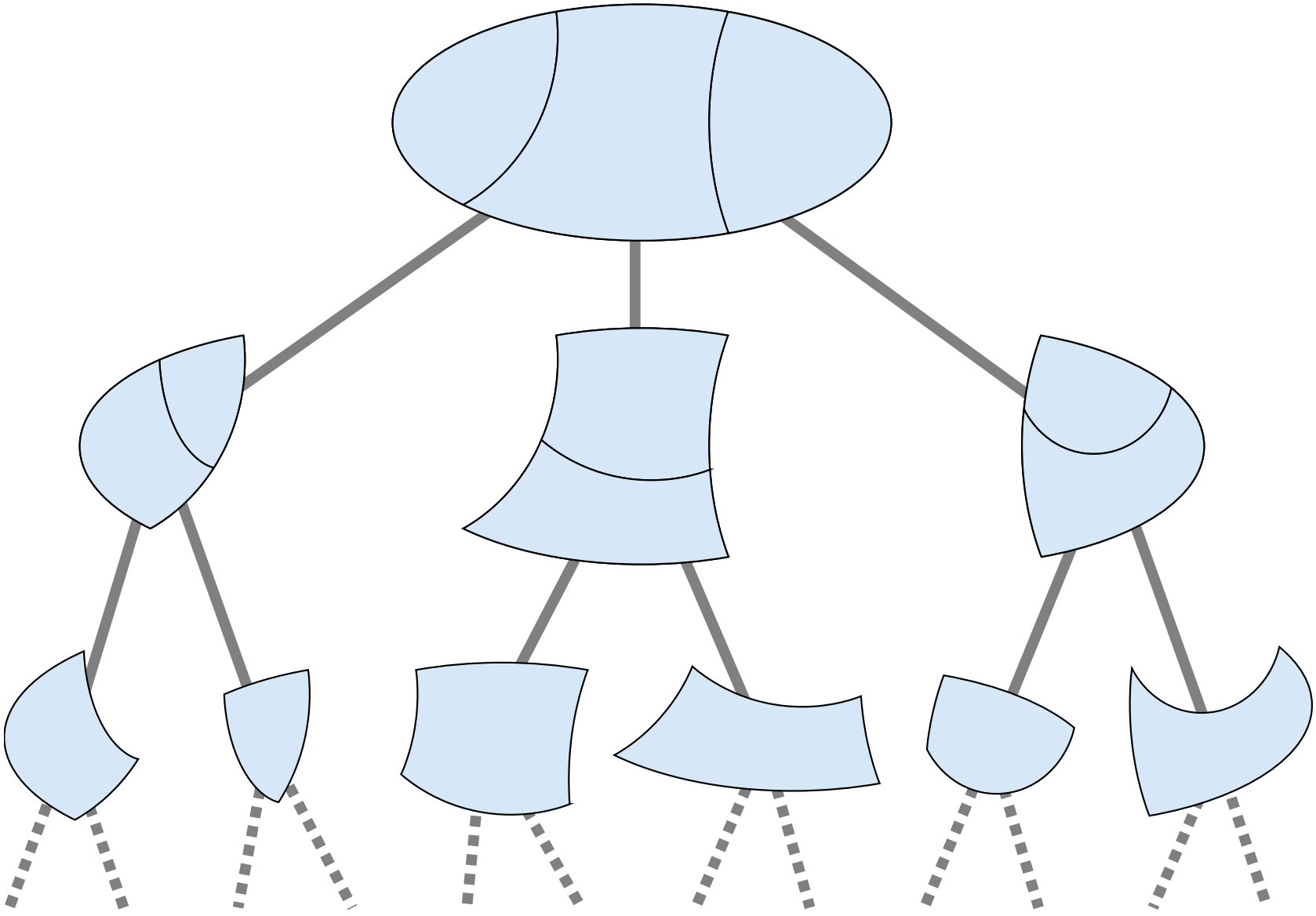
*For any oblivious routing scheme on the mesh there is a demand $P$ such that routing $P$ will give congestion $\Omega(\log n \cdot C_{\text{opt}})$.*

In the following we design oblivious algorithms that obtain close to optimum congestion (no bounds on dilation).

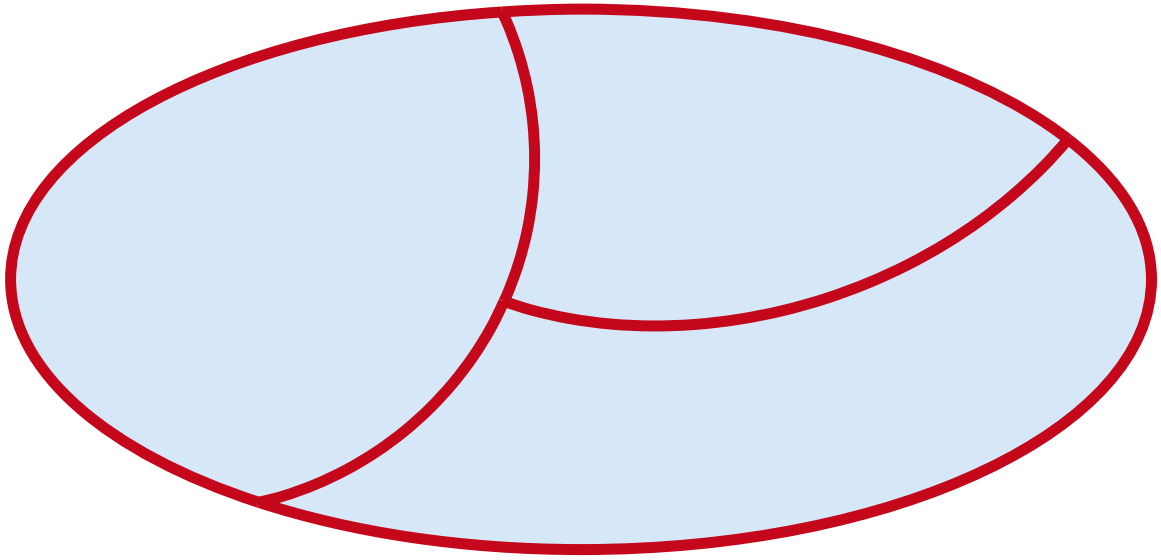
We always assume that we route a flow (instead of packet routing).

We can also assume this is a randomized path-selection scheme that guarantees that the expected load on an edge is close to the optimum congestion.

# Hierarchical Decompositions



# Hierarchical Decompositions & Oblivious Routing



define multicommodity flow problem for every cluster:

- every border edge of a sub-cluster injects one unit and distributes it evenly to all others

Formally

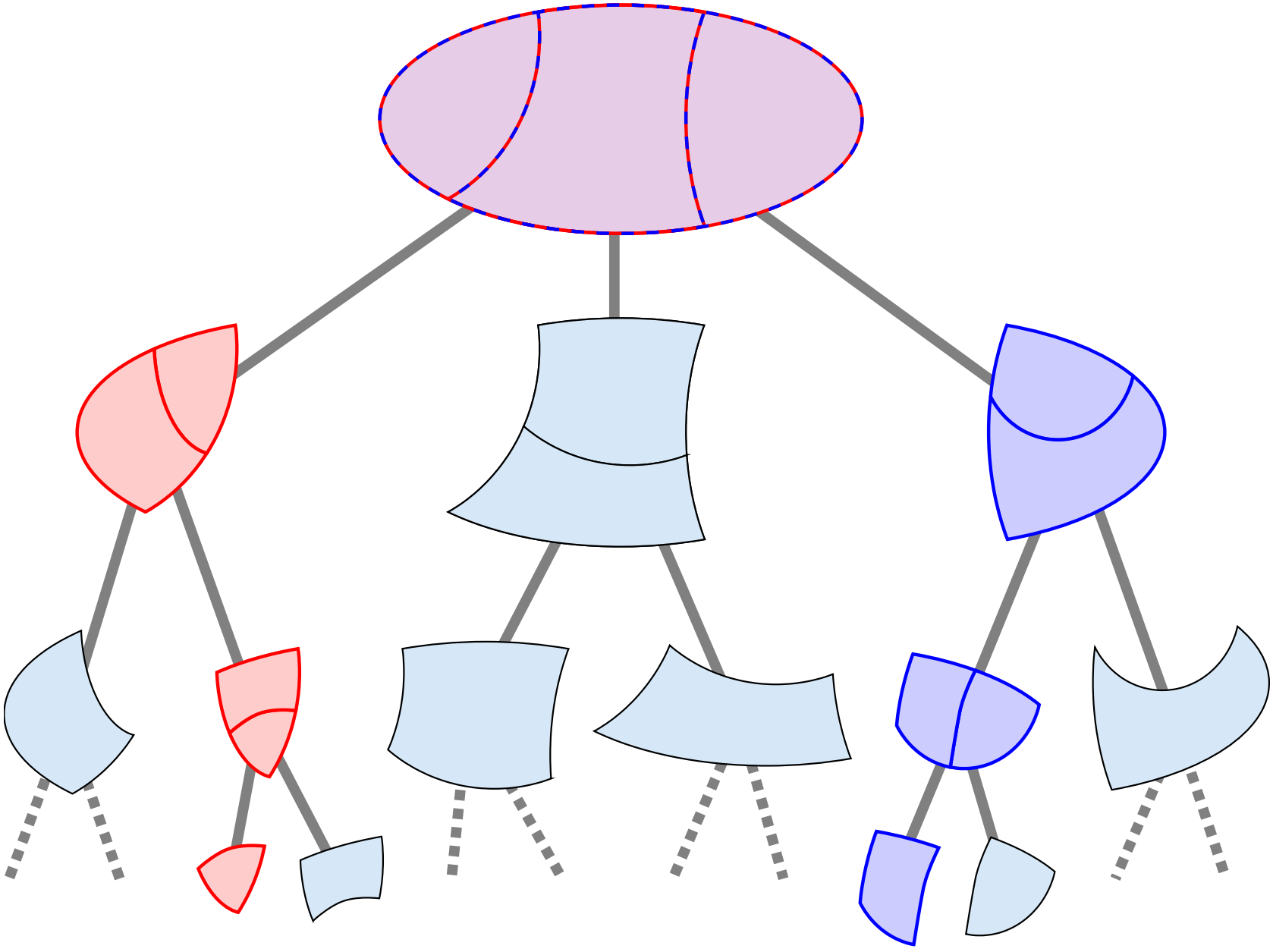
- cluster $S$ partitioned into clusters $S_1, \ldots, S_\ell$
- weight $w_S(v)$ of node $v$ is total capacity of edges connecting $v$ to nodes in other sub-clusters or outside of $S$
- demand for pair $(x, y) \in S \times S$

$$\frac{w_S(x) w_S(y)}{w_S(S)}$$

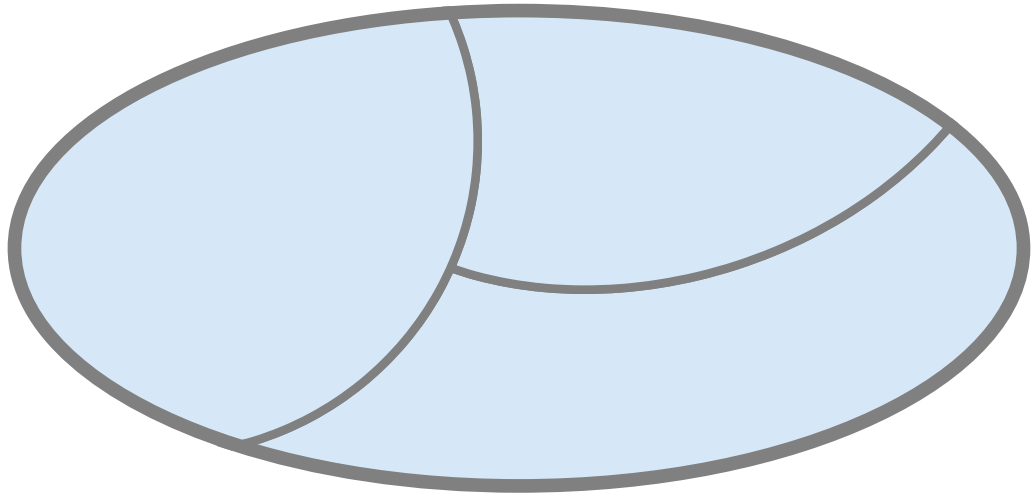
- gives flow problem for every cluster
- if every flow problem can be solved with congestion $C$ then there is an oblivious routing scheme that always obtains congestion

$$\mathcal{O}(\text{height}(T) \cdot C \cdot C_{\text{opt}}(\mathcal{P}))$$

# Oblivious Routing Scheme



# Oblivious Routing Scheme — A Single Cluster $S$



**Input:**
Messages from sub-clusters have been routed to random border-edges of corresponding sub-cluster.

1. forward messages to random intra sub-cluster edge
2. delete messages for which source and target are in $S$
3. forward remaining messages to random border edge

all performed by applying flow problem for cluster several times

# Sparsest Cut

**Definition 83**

Given a multicommodity flow problem $\mathcal{P}$ with demands $D_i$ between source-target pairs $s_i, t_i$. A sparsest cut for $\mathcal{P}$ is a set $S$ that minimizes

$$\Phi(S) = \frac{\text{capacity}(S, V \setminus S)}{\text{demand}(S, V \setminus S)} \ .$$

$\text{demand}(S, V \setminus S)$ is the demand that crosses cut $S$.
$\text{capacity}(S, V \setminus S)$ is the capacity across the cut.

# Sparsest Cut

Clearly,

$$1/\Phi_{\min} \leq C_{opt}(\mathcal{P})$$

For single-commodity flows we have $1/\Phi_{\min} = C_{opt}(\mathcal{P})$.

In general we have

$$\frac{1}{\Phi_{\min}} \leq C_{opt}(\mathcal{P}) \leq \mathcal{O}(\log n) \cdot \frac{1}{\Phi_{\min}} \ .$$

This is known as an approximate maxflow mincut theorem.

# LP Formulation

**Maximum Concurrent Flow:**

$$
\begin{array}{lrlll}
\max & & \lambda & & \\
\text{s.t.} & \forall i & \sum_{p \in P_{s_i,t_i}} f_p & \geq & D_i \\
& \forall e \in E & \sum_{p:e \in p} f_p & \leq & c(e) \\
& & f_p, \lambda & \geq & 0
\end{array}
$$

$\mathcal{P}_{s,t}$ is the set of path that connect $s$ and $t$.

**The Dual:**

$$
\begin{array}{lrlll}
\min & & \sum_e c(e)\ell(e) & & \\
\text{s.t.} & \forall p \in \mathcal{P} & \sum_{e \in P} \ell(e) & \geq & \text{dist}_i \\
& & \sum_i D_i \text{dist}_i & \geq & 1 \\
& & \text{dist}_i, \ell(e) & \geq & 0
\end{array}
$$

# Duality

**Primal:**

$$\begin{array}{lrcl} \max & c^t x & & \\ \text{s.t.} & Ax & \leq & b \\ & x & \geq & 0 \end{array}$$

**Dual:**

$$\begin{array}{lrcl} \min & b^t y & & \\ \text{s.t.} & A^t y & \geq & c \\ & y & \geq & 0 \end{array}$$

# Metric Embeddings

## Definition 84

A metric $(V, d)$ is an $\ell_1$-embeddable metric if there exists a function $f : V \to \mathbb{R}^m$ for some $m$ such that

$$d(u, v) = \|f(u) - f(v)\|_1$$

## Definition 85

A metric $(V, d)$ embeds into $\ell_1$ with distortion $\alpha$ if there exists a function $f : V \to \mathbb{R}^m$ for some $m$ such that

$$\frac{1}{\alpha}\|f(u) - f(v)\|_1 \leq d(u, v) \leq \|f(u) - f(v)\|$$

**Theorem 86**

*Any metric $(V, d)$ on $|V| = n$ points is embeddable into $\ell_1$ with distortion $\mathcal{O}(\log n)$.*

**Theorem 87**

*For any flow problem $\mathcal{P}$ one can obtain at least a throughput of $\Phi_{min}/\log n$, where $\Phi_{\min}$ denotes the sparsity of the sparsest cut. In other words*

$$C_{opt}(\mathcal{P}) \leq \mathcal{O}(\log n)\frac{1}{\Phi_{min}}$$

# LP Formulation

The optimum throughput is given by

$$\begin{array}{lrcl} \min & \sum_e c(e)d(e) & & \\ \text{s.t.} & d \text{ metric} & & \\ & \sum_i D_i d(s_i,t_i) & \geq & 1 \end{array}$$

or

$$\begin{aligned} C_{\text{opt}}(\mathcal{P}) &= \frac{\sum_i D_i d(s_i,t_i)}{\sum_{e=(u,v)} c(e)d(u,v)} \\ &\leq \alpha \frac{\sum_i D_i \cdot \|f(s_i)-f(t_i)\|}{\sum_{e=(u,v)} c(e) \cdot \|f(u)-f(v)\|} \\ &= \alpha \frac{\sum_i D_i \cdot \sum_S \gamma_S \chi_S(s_i,t_i)}{\sum_{e=(u,v)} c(e) \cdot \sum_S \gamma_S \chi_S(u,v)} \\ &= \alpha \frac{\sum_S \gamma_S \sum_i D_i \chi_S(s_i,t_i)}{\sum_S \gamma_S \sum_{e=(u,v)} c(e)\chi_S(u,v)} \\ &\leq \alpha \max_S \frac{\sum_i D_i \chi_S(s_i,t_i)}{\sum_{e=(u,v)} c(e)\chi_S(u,v)} = \alpha \cdot \frac{1}{\Phi_{\min}} \end{aligned}$$

# Fréchet Embedding

Given a set $A$ of points we define a mapping

$$f(x) := d(x, A)$$

The mapping $f$ is contracting this means

$$\|f(x) - f(y)\| \leq d(x, y)$$

Suppose we have a probability distribution $p$ over sets $A_1, \ldots, A_k$:

Then define $f : V \to \mathbb{R}^k$ by

$$f(x)_i : V = p(A_i) \cdot d(x, A_i)$$

$f$ is still contracting.

We use a probability distribution over sets such that the expected distance between $x$ and $y$ is at least

$$d(x,y)/\mathcal{O}(\log n)$$