Complexity and Advanced Algorithms
Monsoon 2011

Parallel Algorithms
Lecture 3

The Power of CRCW – Minima

- Two points of interest
  - Illustrate the power of CRCW models
  - Illustrate another optimality technique.
- Find minima of n elements.
  - Input: An array A of n elements
  - Output: The minimum element in A.
- From what we already know:
  - Standard sequential algorithm not good enough
  - Can use an upward traversal, with min as the operator at each internal node. Time = O(log n), work = O(n).

The Power of CRCW – Minima

- Our solution steps:
  - Design a $O(n^2)$ work, $O(1)$ time algorithm.
  - Gain optimality by sacrificing runtime to $O(\log \log n)$.

An O(1) Time Algorithm

	12	17	8	18	26
12	--	1	0	1	1
17	0	--	0	1	1
8	1	1	--	1	1
18	0	0	0	--	1
26	0	0	0	0	--

- Use n^2 processors.
- Compare A[i] with A[j] for each i and j.
- Now can identify the minimum.

An O(1) Time Algorithm

	12	17	8	18	26
12	--	1	0	1	1
17	0	--	0	1	1
8	1	1	--	1	1
18	0	0	0	--	1
26	0	0	0	0	--

- Use n^2 processors.
- Compare A[i] with A[j] for each i and j.
- Now can identify the minimum.
  - How?

An O(1) Time Algorithm

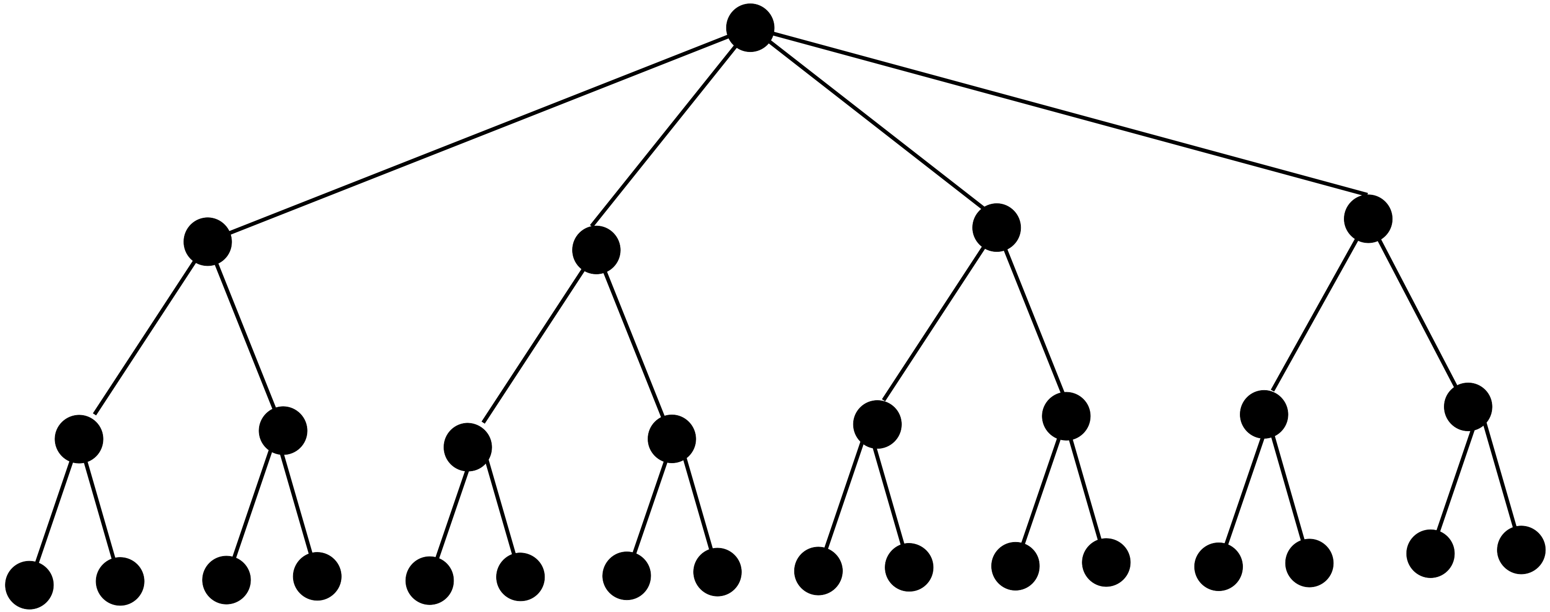
	12	17	8	18	26
12	--	1	0	1	1
17	0	--	0	1	1
8	1	1	--	1	1
18	0	0	0	--	1
26	0	0	0	0	--

- Use n^2 processors.
- Compare A[i] with A[j] for each i and j.
- Now can identify the minimum.
  - How?
- Where did we need the CRCW model?

Towards Optimality

- The earlier algorithm is heavy on work.
- To reduce the work, we proceed as follows.
- We derive an O(nlog log n) work algorithm running in O(log log n) time.
- For this, use a doubly logarithmic tree.
  - Defined in the following.

Doubly Logarithmic Tree



- Let there be $n = 2^{2^k}$ leaves, the root is level 0. The root has $\sqrt{n} = 2^{2^{k-1}}$ children.
- In general, a node at level i has $n/2^{i-1} = 2^{2^{k-i-1}}$ children, for $0 \leq i \leq k-1$.
- Each node at level k has two leaf nodes as children.

Doubly Logarithmic Tree

- Some claims:
  - Number of nodes at level i is $2^{2^k - 2^{k-i}}$.
  - Number of nodes at the kth level is n/2.
  - Depth of a doubly logarithmic tree of n nodes is k+1 = log log n + 1.
- To compute the minimum using a doubly logarithmic tree:
  - Each internal node performs the min operation does not suffice.
  - Why?

Minima Using the Doubly Logarithmic Tree

- Intuition:
  - Should spend only O(1) time at each internal node.
  - Use the O(1) time algorithm at each internal node.
- At each internal node of level i, if there are c_i children, use c_i^2 processors.
  - Minima takes O(1) time at each level.
  - Also, No. of nodes at level i x No. of processors used $= 2^{2^k - 2^{k-i}} \cdot (2^{2^{k-i-1}})^2 = 2^{2^k} = n$.

Minima Using a Doubly Logarithmic Tree

- Second, slightly improved result:
  - With n processors, can find the minima of n numbers in O(log log n) time.
  - Total work = O(n log log n).
- Still suboptimal by a factor of O(log log n).
- We now introduce a technique to achieve optimality.

Accelerated Cascading

- Our two algorithms:
  - Algorithm 1: A slow but optimal algorithm.
    - Binary tree based: O(log n) time, O(n) work.
  - Algorithm 2: A fast but non-optimal algorithm
    - Doubly Logarithmic tree based: O(log log n) time, O(nlog log n) work.
- The accelerated cascading technique suggests one to combine two such algorithms to arrive at an optimal algorithm
  - Start with the optimal algorithm till the problem is small enough
  - Switch over to the fast but non-optimal algorithm.

Accelerated Cascading

- The binary tree based algorithm starts with an input of size n.
- Each level up the tree reduces the size of the input by a factor of 2.
- In log log log n levels, the size of the input reduces to $n/2^{\text{logloglog } n} = n/\text{loglog } n$.

- Now switch over to the fast algorithm with n/loglog n processors, needing O(log log (n/log log n)) time.

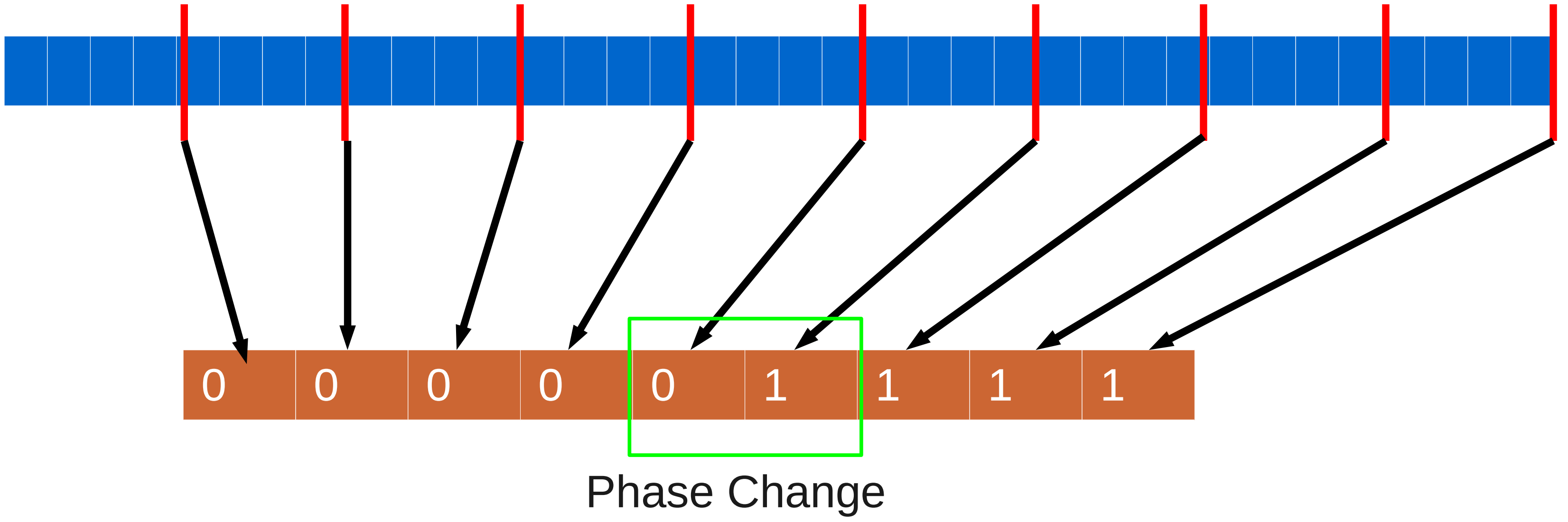
Final Result

- Total time = O(log log log n) + O(log log n).
- Total work = O(n).
- Need CRCW model.
- Where did we need the CRCW model?

Parallel Search

- Search for an item in a sorted array
  - Input: A sorted array of n elements, and an item x.
  - Output: 1 if x is in A, 0 otherwise.
- Other output models possible,
  - Return the index at which x is found in A
  - Return the index of the largest (resp. smallest) element smaller (greater) than x.
- Binary search in the sequential setting takes O(log n) time.
- What is the scope for parallelism?

Parallel Search



- p-way search for a given p.
- Compare x with A[i.n/p] for $1 \leq i \leq p$.
- Record the phase change and recurse, if more than n/p elements.

Parallel Search

- Time taken:
  - $T(n) = T(n/p) + O(1)$
  - Solution: $T(n) = O(\log_p n)$.
  - Work = $O(p \log_p n)$.
  - Model: CREW.
- Optimal only when $p = O(1)$!
- But we will see that this has some applications for non-bounded p.

List Ranking

- List ranking is a fundamental problem in parallel computing.
- Given a list of elements, find the distance of the elements from one end of the list.
- In sequential computation, not a serious problem.
  - Can simply traverse the list from one end.
- But this approach does not scale well for parallel architectures.

List Ranking

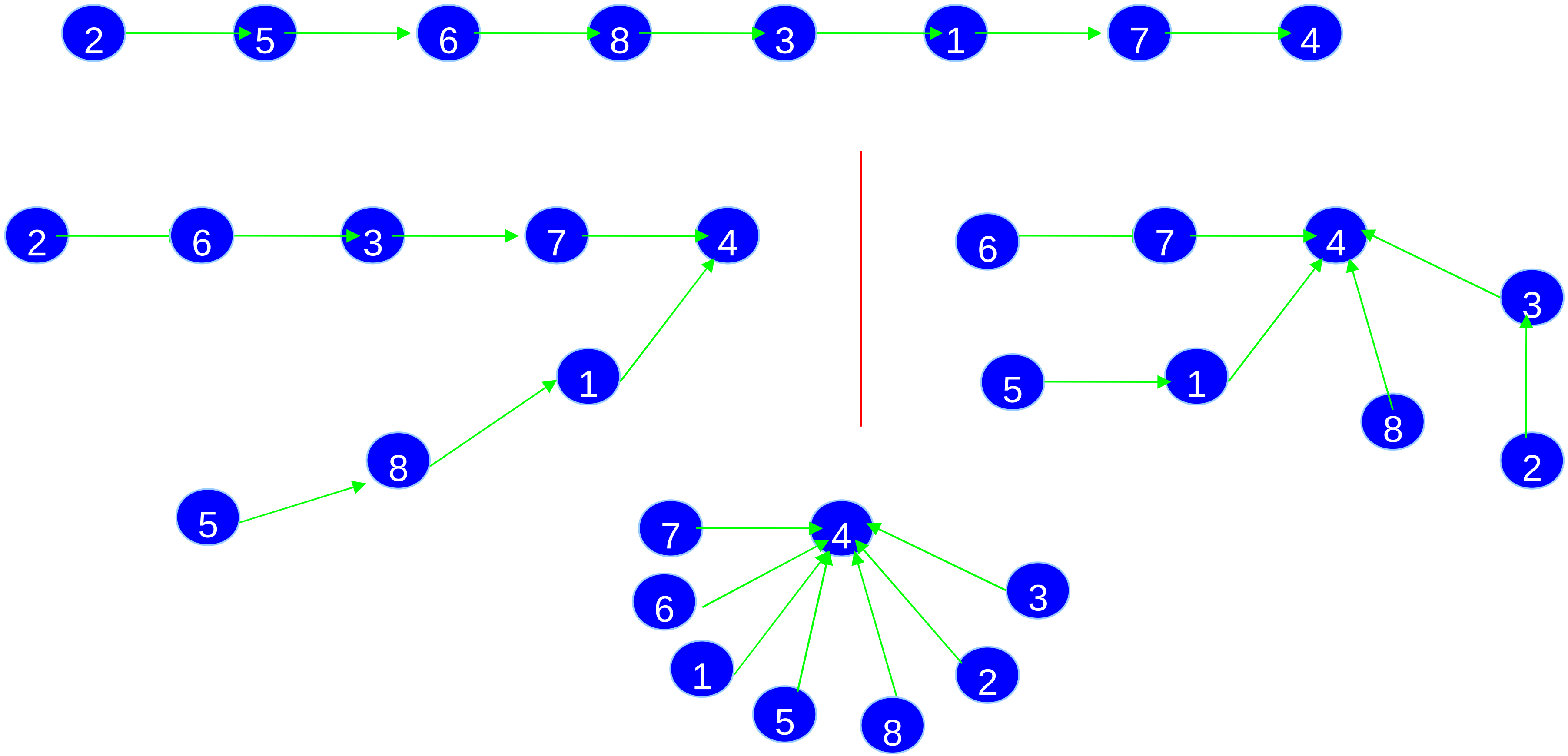
List

1 → 8 → 5 → 11 → 2 → 6 → 10 → 4 → 3 → 7 → 12 → 9

Succ	8	6	7	3	11	10	12	5	--	4	2	9
Rank	1	5	9	8	3	6	10	2	12	7	4	11

- Representation via an array of successor pointers.

Pointer Jumping Solution



- Each node updating its parent to be its grandparent.

Pointer Jumping Solution

```
Algorithm FindRoot
   for 1 ≤ i ≤  n do in parallel
      R(i) = 1;
      while P(i) ≠ P(P(i)) do
         R(i) = R(i) + R(P(i))
         P(i) = P(P(i))
end.
```

- The pseudo code above computes the rank of every element in parallel.
  - R() refers to the rank, P() refers to the parent.

Pointer Jumping Solution

```
Algorithm FindRoot
   for 1 ≤ i ≤  n do in parallel
      R(i) = 1
      while R(i) ≠ R(R(i)) do
         R(i) = R(i) + R(P(i))
         P(i) = P(P(i))
end.
```

- Claim: The above algorithm can finish in O(log n) time.
- Proof: Show that the distance between a node and its parent doubles every iteration of the while loop.
  - Maximum distance is n.

Pointer Jumping Solution

```
Algorithm FindRoot
   for 1 ≤ i ≤  n do in parallel
      R(i) = 1
      while R(i) ≠ R(R(i)) do
         R(i) = R(R(i))
end.
```

- Claim: The above algorithm has a work complexity of O(n log n).
- Proof: Each processor needs at most O(log n) work.
- Therefore, our algorithm is sub-optimal.
  - Can be made optimal using Technique 1. Details follow.

Pointer Jumping Solution

```
Algorithm FindRoot
   for 1 ≤ i ≤  n do in parallel
      R(i) = 1
      while R(i) ≠ R(R(i)) do
         R(i) = R(i) + R(P(i))
         P(i) = P(P(i))
end.
```

- Few implementation issues
  - In the PRAM model, synchronous execution means that all n processors execute each step in the while loop at the same time.
  - Any problems otherwise?

Pointer Jumping Solution

```
Algorithm FindRoot
   for 1 ≤ i ≤  n do in parallel
      R(i) = 1
      while R(i) ≠ R(R(i)) do
         R(i) = R(i) + R(P(i))
         P(i) = P(P(i))
end.
```

- Few implementation issues
  - In the PRAM model, synchronous execution means that all n processors execute each step in the while loop at the same time.
- Any problems otherwise?
  - Inconsistent results!

Pointer Jumping Solution

```
Algorithm FindRoot
   for 1 ≤ i ≤  n do in parallel
      R(i) = 1
      while R(i) ≠ R(R(i)) do
         R(i) = R(i) + R(P(i))
         P(i) = P(P(i))
end.
```

- To get around, one can consider packing R and P values of a node into a single word.
- If list has no more than 2^{32} elements, can use 64 bit architectures with each word packing two 32 bit numbers.

Advanced Optimal Solutions

$1 \rightarrow 8 \rightarrow 5 \rightarrow 11 \rightarrow 2 \rightarrow 6 \rightarrow 10 \rightarrow 4 \rightarrow 3 \rightarrow 7 \rightarrow 12 \rightarrow 9$

- General technique suggests that we solve a smaller problem and extend the solution to the larger problem.
- To apply our technique we should use the pointer jumping based solution on a sub-list of size n/log n.
- How to identify such a sublist?

Advanced Solutions

1 → 8 → 5 → 11 → 2 → 6 → 10 → 4 → 3 → 7 → 12 → 9

1 → 5 → 11 → 6 → 4 → 3 → 12 → 9

- Cannot pick equidistant as earlier.
- However, can pick independent nodes.
  - Removing independent nodes is easy!
  - Formally, an independent set of nodes.
  - Can extend the solution easily in such a case.

Advanced Solutions

- Formally, in a graph $G = (V, E)$, a subset of nodes $U \subseteq V$ is called an independent set if for ever pair of vertices u,v in U, $(u,v) \notin E$.
- Linked lists (viewed as a graph) have the property that they have large independent sets.

Advanced Solutions

1 → 8 → 5 → 11 → 2 → 6 → 10 → 4 → 3 → 7 → 12 → 9

List with elements removed

1 →	5 →	11 →	6 →	4 →	3 →	12 →	9
(1)	(2)	1)	(2)	(2)	(1)	(2)	(1)

Ranked short list

1 →	5 →	11 →	6 →	4 →	3 →	12 →	9
(1)	(3)	(4)	(6)	(8)	(9)	(11)	(12)

8
(2)

reintroduce element 8

- Transfer current rank along with successor during removal.