

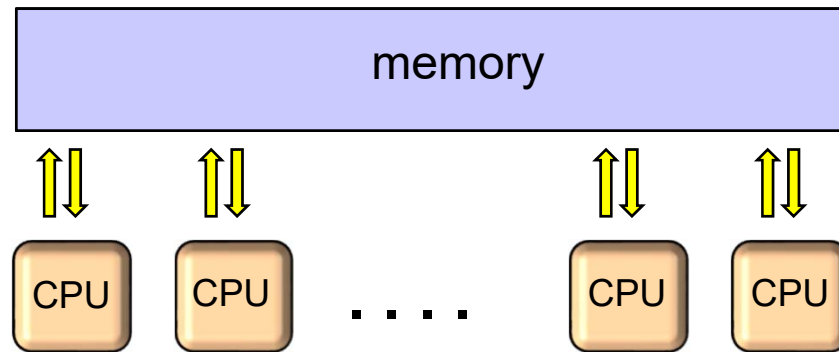


# PRAM 1

## Model and basic algorithms

CS121 Parallel Computing  
Spring 2021

# PRAM



- Parallel Random Access Machine, generalizes von Neumann model for sequential computing.
  - Given input of size  $n$ , we have  $f(n)$  processors accessing a shared memory.
    - $f(n)$  can be very large, even larger than  $n$ .
  - All processors execute in synchronized steps.
  - In each step, each processor reads a memory location, computes, then writes a memory location.



# PRAM

- Theoretically interesting model, but not practical.
  - Assumes unrealistically large number of processors.
  - Also assumes all processors can communicate every time step; ignores memory latency and bandwidth.
- PRAM's main use is as a simple, clean model to develop parallel algorithms.
  - First maximize parallelism inherent in problem using PRAM.
  - Then simulate the algorithm with real hardware, i.e. map it onto hardware with limited processors / communication.
  - **Ex** Some GPU algorithms are adaptations of PRAM algorithms.



# Memory conflicts

- What if processors read / write to the same memory location in same time step?
- EREW Exclusive read exclusive write.
  - Most restrictive model. Algorithm returns error if processors read/write same location simultaneously.
- CREW Concurrent read exclusive write.
  - Several processors can read same location simultaneously, but error if they write.
- ERCW Exclusive read concurrent write.
  - Uncommon.
- CRCW Concurrent read concurrent write.
  - If multiple writes to same location, can either
    - Let an arbitrary write succeed.
    - Choose a write according to some priority to succeed.



# Work and depth

- Depth is the number of (parallel) steps till a PRAM algorithm terminates.
  - Polylogarithmic depth means the algorithm terminates in  $O(\log(n)^k)$  steps, where  $n$  is input size and  $k$  is constant.
  - Goal for PRAM algorithms is often polylog depth using  $O(n^k)$  number of processors.
- Work is total number of steps taken by the algorithm.
  - Work of parallel algorithm  $\geq O(\text{work of best sequential algorithm})$ .
  - If the work is equal, the parallel algorithm is work-efficient.
- In practice, minimizing work of PRAM algorithm is more important than minimizing depth.

# Parallel carry lookahead addition

$$\begin{array}{r} a \quad \quad 1 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \\ b \quad + \quad 1 \ 0 \ 1 \ 0 \ 0 \ 1 \ 1 \\ \hline \text{carry} \quad 1 \ 0 \ 1 \ 0 \ 0 \ 1 \ 1 \\ \text{sum} \quad 1 \ 0 \ 1 \ 0 \ 1 \ 1 \ 0 \ 0 \end{array}$$

- Suppose we want to add two  $n$ -digit binary numbers, but we can only add a single digit at a time and compute its carry.
  - This is what's provided by full adders in a CPU.
- If we add digit by digit using the grade school method, it takes  $O(n)$  time.
  - For  $n=32$  or  $n=64$ , this is much too slow.
- Each digit in the sum depends on the digit from the summands, but also a carry bit from the previous digit.
  - The summand digits can be added in parallel, but it seems the carry bits must be computed sequentially.

$a_i$	$b_i$	$c_i$	$s_i$	$c_{i+1}$
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



# Parallel carry lookahead addition

- We'll show how to compute all the carry bits in parallel in  $O(\log n)$  time using  $n$  processors.
- After this, all the sum bits can be computed in  $O(1)$  parallel time, since  $s_i = a_i \oplus b_i \oplus c_i$ .
- Denote bitwise AND and OR by  $\cdot$  and  $+$ .
- Define  $g_i = a_i \cdot b_i$  as  $i$ 'th "carry generate" bit.
  - If  $a_i = b_i = 1$ ,  $c_{i+1} = 1$  no matter what  $c_i$  is.
- Define  $p_i = a_i \oplus b_i$  as  $i$ 'th "carry propagate" bit.
  - If  $p_i = 1$ , then  $c_{i+1} = c_i$ .

# Parallel carry lookahead addition

- We have  $c_{i+1} = g_i + c_i p_i$ .
- Carry the  $i+1$ 'st bit if
  - $i$ 'th bit of  $a$  and  $b$  generate a carry, OR
  - We carried the  $i$ 'th bit, and this was propagated by  $a$  and  $b$ 's  $i$ 'th bit.
- We can also verify  $c_{i+1} = g_i + c_i p_i$  directly.

$a_i$	$b_i$	$c_i$	$g_i$	$p_i$	$c_{i+1}$
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	1	0
0	1	1	0	1	1
1	0	0	0	1	0
1	0	1	0	1	1
1	1	0	1	0	1
1	1	1	1	0	1



# Parallel carry lookahead addition

- **Observation** Can write  $\begin{bmatrix} p_i & g_i \\ F & T \end{bmatrix} \begin{bmatrix} c_i \\ T \end{bmatrix} = \begin{bmatrix} c_i p_i + g_i \\ T \end{bmatrix} = \begin{bmatrix} c_{i+1} \\ T \end{bmatrix}$ 
  - Recall  $\cdot$  and  $+$  represent AND and OR.
  - Boolean matrix multiplication done same way as for reals.

- Applying this repeatedly, we get

$$\begin{aligned} \begin{bmatrix} c_{i+1} \\ T \end{bmatrix} &= \begin{bmatrix} p_i & g_i \\ F & T \end{bmatrix} \begin{bmatrix} c_i \\ T \end{bmatrix} \\ &= \begin{bmatrix} p_i & g_i \\ F & T \end{bmatrix} \begin{bmatrix} p_{i-1} & g_{i-1} \\ F & T \end{bmatrix} \begin{bmatrix} c_{i-1} \\ T \end{bmatrix} = \dots \\ &= \begin{bmatrix} p_i & g_i \\ F & T \end{bmatrix} \dots \begin{bmatrix} p_1 & g_1 \\ F & T \end{bmatrix} \begin{bmatrix} c_0 \\ T \end{bmatrix} \end{aligned}$$

- Since all the  $p_i$  and  $g_i$  values are known, the final product can be computed using prefix sum in  $O(\log n)$  time with  $n$  processors.
- This algorithm or variants are implemented in most real CPUs.

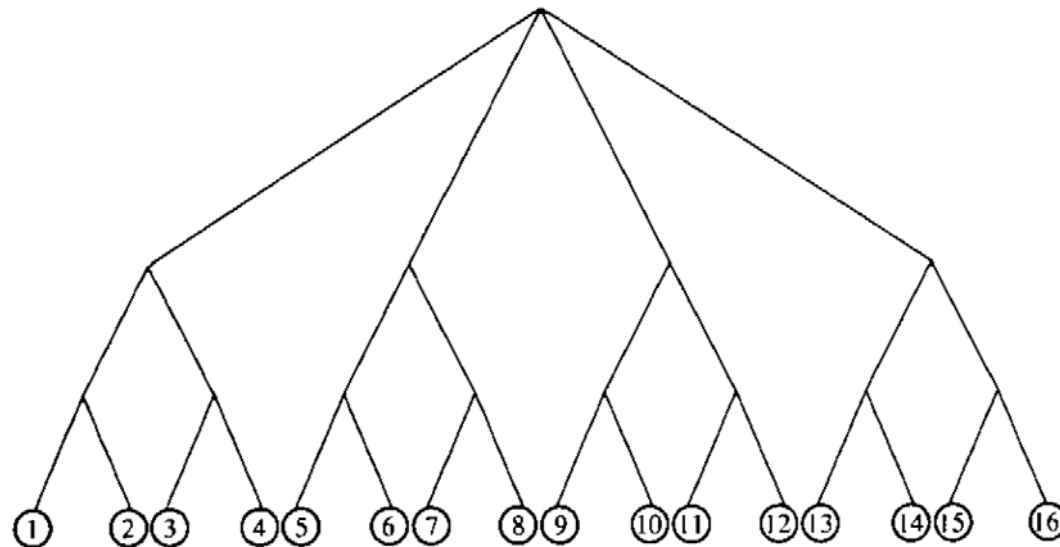


# Constant time max finding

- Using a balanced binary tree we can find the max of  $n$  numbers in  $O(\log n)$  time and  $O(n)$  work.
- We show how to find max in  $O(\log \log n)$  time using  $O(n)$  work on a min priority CRCW PRAM.
  - I.e. when multiple Boolean values are written to same location, the min value wins.
- First, we can find the max of  $p$  numbers  $x_1, \dots, x_p$  in  $O(1)$  time and  $O(p^2)$  work on the CRCW PRAM.
  - For  $1 \leq i, j \leq p$ , in parallel set  $B(i, j) = 1$  if  $x_i \geq x_j$ , and  $B(i, j) = 0$  otherwise.
    - Uses  $p^2$  processors.
  - For  $1 \leq i \leq p$ , in parallel set  $M_i = B(i, 1) \wedge B(i, 2) \wedge \dots \wedge B(i, p)$ .
    - $M_i = 1$  iff  $x_i$  is the max value.
    - This requires that when 0's and 1's are written to the same  $M_i$ , the minimum value (i.e. 0) gets written.

# Doubly logarithmic tree

- Create a tree with the  $x_i$ 's at the leaves.
- For each internal node  $u$ , let  $n_u$  be the number of leaves in the subtree rooted at  $u$ . Make the degree of  $u$  be  $\lceil \sqrt{n_u} \rceil$ .
  - For simplicity, assume  $n = 2^{2^k}$ . Then the tree has  $k = \log \log n$  levels.
  - The root of the tree has degree  $2^{2^{k-1}} = \sqrt{n}$ .
  - Each child of the root has degree  $2^{2^{k-2}}$ .
  - In general, at level  $0 \leq i \leq k - 1$ , each node has degree  $2^{2^{k-i-1}}$ , and there are  $2^{2^k - 2^{k-i}}$  nodes total at the level.



Source: Introduction to  
Parallel Algorithms, Jaja



# Superfast max finding

- Suppose each node computes the max of all its children.
  - Then each node has the max of all the leaf nodes in its subtree, and the root has the overall max value.
  - To compute the max of  $p$  children takes  $O(p^2)$  work.
- Total time for algorithm is  $O(\log \log n)$ .
- Total work per level is  $O\left(\left(2^{2^{k-i-1}}\right)^2 \cdot 2^{2^k - 2^{k-i}}\right) = O\left(2^{2^k}\right) = O(n)$ .
  - Total overall work is  $O(n \log \log n)$ . So the algorithm isn't work efficient.



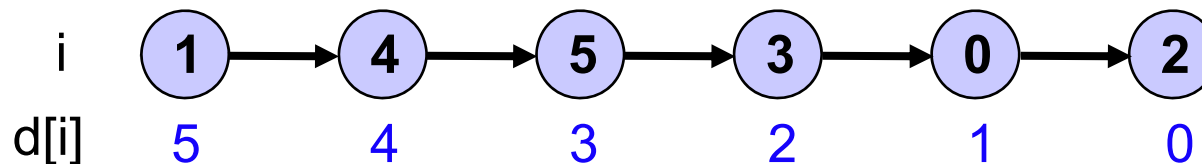
# Superfast max finding

- To make the previous algorithm work efficient, we use a technique called accelerated cascading.
  - Start with an optimal algorithm until problem size is sufficiently small.
  - Then switch to fast but nonoptimal algorithm.
- First, partition the  $n$  values into  $n' = n / \log \log n$  blocks of size  $\log \log n$  each.
  - Use  $n / \log \log n$  processors. Each processor sequentially finds the max of one block of values.
  - This takes  $O(\log \log n)$  time and does  $O(n)$  work.
  - Then use the doubly logarithmic tree on the  $n'$  values.
    - This runs for  $O(\log \log n') = O(\log \log n)$  time.
    - It does  $O(n' \log \log n') = O(n)$  work.

# List ranking

- Given a linked list, compute the distance of each node to the end.
  - Linked list is represented by an array next, where next[i] initially points to node following node i.
  - next[i]=NULL for the last node.
- Let d[i] be i's estimate of its distance to the end.
  - Initially d[i]=0 for the last node, and d[i]=1 for all other nodes.

i	0	1	2	3	4	5
next[i]	2	4	⊥	0	5	3

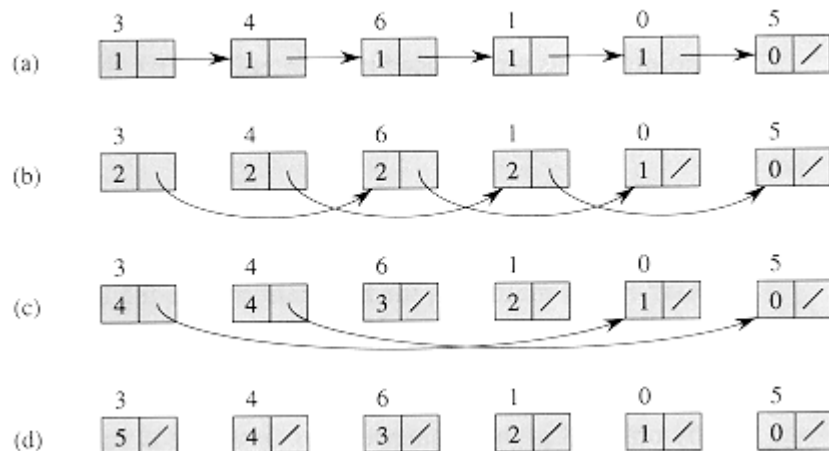


# List ranking

- Repeatedly apply pointer jumping.
  - If currently  $i \rightarrow j$  and  $j \rightarrow k$ , set  $i \rightarrow k$ .
- Let  $k$  be a node that's distance  $m$  away from the end, for some  $m$ .
  - After  $i$  steps,  $d[k] = \min(m, 2^i)$ , and  $\text{next}[k]$  points  $\min(m, 2^i)$  distance away.
- Since  $d[*] \leq n$ , algorithm terminates in  $O(\log n)$  steps.
- Work is  $O(n \log n)$ .
  - Not efficient, since sequential list ranking takes  $O(n)$  work.
- List ranking has many applications, including Euler tour technique, connected components, expression tree evaluation, ear decomposition, etc.

```

while next[i] ≠ NULL for some i
  do parallel for all i
    if next[i] ≠ NULL
      d[i] = d[i] + d[next[i]]
      next[i] = next[next[i]]
  
```

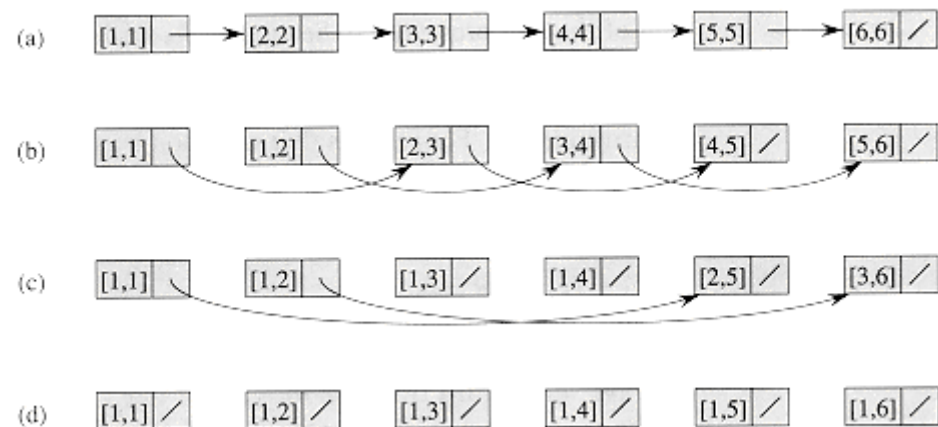


# Prefix sum on linked list

- We've seen how to do prefix sum on an array.
- Using pointer jumping, can also do prefix sum on a linked list.
  - Initially each node  $i$  has a value  $x[i]$ .
  - The output, i.e. prefix sum of node  $i$  is stored in  $d[i]$ .
  - Only difference with list ranking is update  $d[\text{next}[i]]$  instead of  $d[i]$ .
- After  $i$  steps, first  $2^i$  nodes have correct prefix sum, and other nodes have the sum of the preceding  $2^i$  values.
- Takes  $O(\log n)$  time, does  $O(n \log n)$  work.

```

do parallel for all i
  d[i]=x[i]
while next[i]≠NULL for some i
  do parallel for all i
    if next[i]≠NULL
      d[next[i]]=d[i]+d[next[i]]
      next[i]=next[next[i]]
    
```





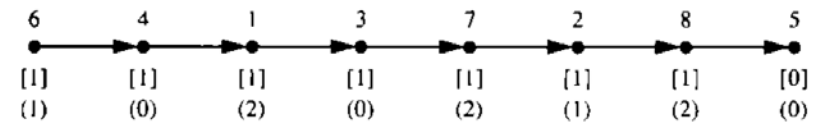


# Work efficient list ranking

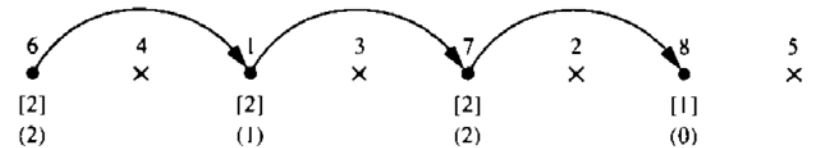
- List ranking using pointer jumping does  $O(n \log n)$  work.
- To make list ranking efficient, we can
  - Shrink the list until only  $O(n / \log n)$  nodes remain.
  - Apply pointer jumping to remaining nodes.
  - Restore the removed nodes and determine their ranks.
- Assume first and third steps take  $O(n)$  work.
- Then second step takes  $O\left(\frac{n}{\log n} \log\left(\frac{n}{\log n}\right)\right) = O(n)$  work, so total work is  $O(n)$ .

# Work efficient list ranking

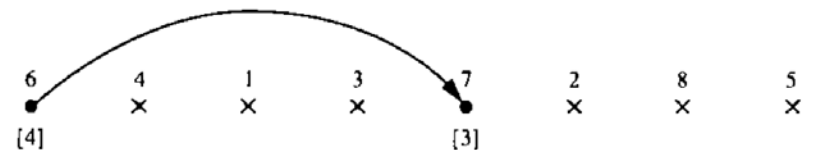
- To shrink the list, we repeatedly remove an independent set of nodes.
  - A set of nodes  $I$  is independent if  $\forall i \in I: (prev(i) \notin I) \wedge (next(i) \notin I)$ .
  - Suppose we have a set of  $n$  nodes. We show next lecture how find  $\Omega(n)$  independent nodes in  $O(\log n)$  time and  $O(n)$  work.
- Given an independent set  $I$ , for each  $i \in I$  set  $dist[prev[i]] = dist[prev[i]] + dist[i]$ .
- To compute distance of a removed node  $i$ , set  $dist[i] = dist[i] + dist[next[i]]$ .



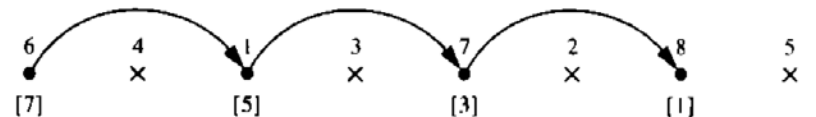
(a)



(b)



(c)



- values in parentheses are used to find independent set.
- dist values are shown in brackets.



# Work efficient list ranking

- Since each round we remove  $\Omega(n)$  number of remaining nodes, it takes  $O(\log \log n)$  rounds to shrink the list to size  $O(n/\log n)$ .
  - After this the pointer jumping takes  $O(\log n)$  time.
- Each round takes  $O(\log n)$  time to find the independent set.
- So total time is  $O(\log n \log \log n)$ .
  - Time can be reduced to  $O(\log n)$  using more efficient algorithm.
- In round  $k$ , number of remaining nodes is  $O(c^k n)$  for some  $c < 1$ .
- So total work to find independent sets in all rounds is  $\sum_{k=0}^{\log \log n} O(c^k n) = O(n)$ .
- Pointer jumping does  $O(n)$  work, so total work is  $O(n)$ .