# PRAM Model

## PRAM Model
(PP Appendix D)

- PRAM = Parallel, Random-Access Machine
- Idealized model introduced in 1978, based on theoretical RAM model
- Unbounded number of processors, to fit problem
- **Shared** common memory
  + local memories per processor
- Processors operate synchronously (like SIMD, with selective writing); could be loosened to SPMD with synchronization routines.
- Writing to common memory is **synchronous**

## Use of PRAM Model

- Simple and elegant for some problems
- Can tell us certain things about structuring, especially for synchronous computation
- Can be simulated on parallel machines (e.g. by rescheduling, Brent's lemma, etc.)
- At least one is being constructed

## SB-PRAM at Universität des Saarlandes Inst. of Parallel Computing

http://www-wjp.cs.uni-sb.de/sbpram/sbpram.html

Constructed a 64 physical (2048 virtual) processor machine, butterfly switch, with 4 GByte of global memory and 256 hard disks. The machine is available on the internet for free access. Software: PRAMOS, P4, Fork compiler (C extension).

## Memory-Conflicts

- All processors can read or write to distinct shared memory locations in **one time step**.
- What if two processors try to **read** from the **same** memory location in the same time step?
- What if two processors try to **write** to the **same** memory location in the same time step?

## PRAM Varieties Based on Memory-Conflict Models

- **EREW** (Exclusive-Read, Exclusive-Write) Concurrent reading from or writing to a location is disallowed.
- **CREW** (Concurrent-Read, Exclusive-Write) Concurrent writing to a location is disallowed.
- **CRCW** (Concurrent-Read, Concurrent-Write) Concurrent writing to a location is allowed.

## Sub-varieties of CRCW (1) indicate how conflict is resolved

- **CRCW-Common**: Concurrent writing is allowed only if it is known that all processors will be writing the ***same*** value (writing **no** value is always an option).
- **CRCW-Arbitrary**: If multiple processors attempt to write, one will be chosen arbitrarily as the winner and the others ignored.

## Sub-varieties of CRCW (2) indicate how conflict is resolved

- **CRCW-Priority**: If multiple processors attempt to write, the highest-priority will be chosen as the winner and the others ignored.
- **CRCW-Sum**: If multiple processors attempt to write, the values will be summed and the sum written instead.
- **Variants on Sum**: Any binary operator (or, and, xor, min, max, product, …)

## Why does it matter?

- To **physically** realize any approximation to a PRAM requires an understanding of the memory conflict model.
- There is a **time cost to resolving memory conflicts**, which varies depending on the model.

## PRAM Preferences

- It is preferable to assume as little as possible for algorithms.
- Therefore, **prefer**
  - CREW over CRCW
  - CRCW-arbitrary over CRCW-common
  - CRCW-common over CRCW-sum
  - etc.

## PRAM Algorithm Examples

- Computing max of n numbers:
  - O(log n) time on EREW (and by implication CREW, CRCW, …)
  - Assume the numbers are in shared memory locations 0, 1, …, n-1.
  - Even numbered processors fetch “their” numbers to their local memory (other processors are idle).
  - Even numbered processes fetch “their neighbors numbers to their local memory.
  - Even numbered processors write the max of the two numbers to “their” locations.
  - Repeat with processors divisible by 4, 8, 16, …

## PRAM Max

Essentially we have a **subtree** of the prefix-sum tree (using max instead of add).

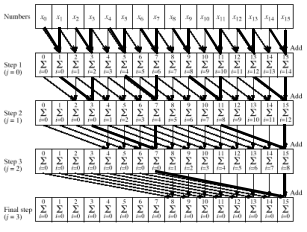



Figure 6.8 Data parallel prefix sum operation.

## PRAM Prefix Sum

- Obviously an EREW PRAM can compute any prefix-sum type computation in O(log n).
- More processors are busy than in the max case.

## Better (?) ways to do max

- Intuitively $\Omega(\log n)$ seems like a **lower bound** on the max computation of n numbers.
- However, a CRCW-arbitrary PRAM can do better.

## CRCW-arbitrary max computation

- O(1)
- using $n^2$ processors

## CRCW-arbitrary max computation setup

- Let the data be in shared memory locations x[0], …, x[n-1].
- Use n bit locations: b[0], …, b[n-1], all set to 1 (in one step).
- b[i] is associated with x[i].

## CRCW-arbitrary max computation

- The meaning is that, at the end of the computation, b[i] will be 0 iff x[i] is less than **some** x[j], where j != i.
- So elements x[i], where b[i] == 1, will be the max.
- In three steps: n*(n-1)/2 processors each fetch, then compare a different x[i] with an x[j]. If x[i] < x[j], the processor sets b[i] to 0, and vice-versa.

## CRCW-arbitrary max computation

- Each processor either writes 0 or does nothing.
- If two processors write to the **same** location, they will both be writing the **same** thing.
- Therefore the CRCW-arbitrary assumption is honored.

# “Parallel sorting in O(1)”

(from http://www.cs.uku.fi/~penttone/parallel/sort.html, has demo)

```
procedure Sort(modifies A: array 1..n of integer)
for i in 1..n pardo K[i]:=0
for i in 1..n pardo
  for j in 1..n pardo
    if A[i]<=A[j] then K[j]:=K[j]+1
for i in 1..n pardo A[K[i]]:=A[i]
```

How many processors?
What kind of conflict resolution?
How much effort?

# Step 1

| A[i] | 3 | 1 | 4 | 2 | 5 |
|---|---|---|---|---|---|
| 3 | | | | | |
| 1 | | | | | |
| 4 | | | | | |
| 2 | | | | | |
| 5 | | | | | |

Input vector to be sorted

# Step 2

| A[i] | 3 | 1 | 4 | 2 | 5 |
|---|---|---|---|---|---|
| 3 | 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 |
| 4 | 0 | 0 | 1 | 0 | 1 |
| 2 | 1 | 0 | 1 | 1 | 1 |
| 5 | 0 | 0 | 0 | 1 | 1 |

processor P[i,j] compares A[i] and A[j]:
P[i,j]:="A[i]<=A[j]"

# Step 3

| A[i] | 3 | 1 | 4 | 2 | 5 |
|---|---|---|---|---|---|
| 3 | 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 |
| 4 | 0 | 0 | 1 | 0 | 1 |
| 2 | 1 | 0 | 1 | 1 | 1 |
| 5 | 0 | 0 | 0 | 0 | 1 |
| K[i] | 3 | 1 | 5 | 2 | 5 |

Add columnwise to K[i]

# Step 4

| A[i] | 3 | 1 | 4 | 2 | 5 |
|---|---|---|---|---|---|
| 3 | 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 |
| 4 | 0 | 0 | 1 | 0 | 1 |
| 2 | 1 | 0 | 1 | 1 | 1 |
| 5 | 0 | 0 | 0 | 0 | 1 |
| K[i] | 3 | 1 | 4 | 2 | 5 |
| A[i] | 1 | 2 | 3 | 4 | 5 |

A[K[j]]:=A[j]; output

# What happened to Ω(log n)?

- In an implementation of CRCW-common, it isn’t physically realizable to have an arbitrary number of processors write to the same location at once, even if they do write the same value.
- We have replaced what would have been binary ops with a single op of arbitrary arity.
- We could implement this op as a fan-in tree, which would recover the Ω(log n).
  [$O(n^2)$ processors fanning in, $\log(n^2) = O(\log n)$].

## Simulation Theorem (see Cormen, et al., p706-708)

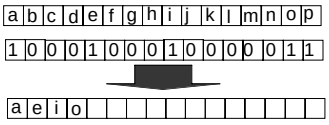
- Any CRCW-common PRAM algorithm using p processors

  can be simulated by an EREW PRAM with a slowdown factor of log(p).

- The proof simulates a CRCW memory by sorting the addresses, using log(p) time to sort on EREW.

## Array Compression

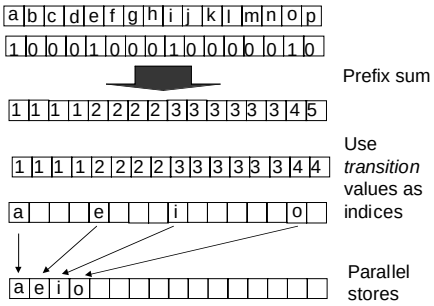
- Problem:
  Given an array in shared memory and a bit vector indicating the elements to be compressed, create an array containing only those elements contiguously.




## Technique

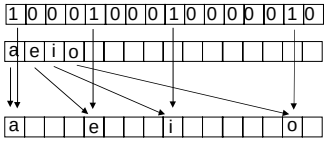
- Use prefix-sum + indexing (parallel)
- Compute the prefix sum of the bit array
- Use the values as indexes of where to store the corresponding item.
- Only use the index at transitions (to avoid the need to use CRCW).

## Array Compression Exposed




## Exercise

- How would you do array ***expansion*** (the "inverse" of compression): distribute array elements according to a bit vector?




## PRAM Computations

- So far have seen:
  - Binary-tree expressions (max, etc.)
  - Prefix sum
  - Vector compression and expansion (uses prefix sum)
  - Max-finding in O(1) (using CRCW-arb)

## PRAM Computations

- Next:
  - Parallel merging
  - Pointer techniques
  - Tree-traversal
  - Quicksort (1-version)

## Parallel Merging

- How can we **merge** two *ordered* arrays in parallel on a PRAM?
- One means is to compute the *indices* of where the elements of one array are inserted into the other array.
- We can then use something similar to array compression to do the actual insertion.

## Computing indices for Parallel Merging

- A single index can be computed with **binary search**: Find the *position* in the other array at which the element would be inserted.
- Add the element's current index to it to get the net final position.
- Do all binary searches in parallel.
- Each array computes the final indexes of its elements in the merged array
- Each processor stores its elements simultaneously

## Parallel Merging

- Cost: n binary searches in parallel
- O(log n) time (on CREW PRAM)

## Exercise

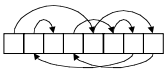
- What is an upper bound for sorting using parallel merging?

## Using **Pointers** in a PRAM

- "Pointer Jumping" technique
- As with prefix sum, this has many uses.
- Basic idea: in a chain of pointers stored in the common memory, the extremities of a chain can be determined in a way that **doubles** the length of the chain at each step:
  - If a location points "N hops away" now, it can point "2N hops away" on the next step.
  - This is because concatenating two N-hop chains gives a 2N-hop chain.

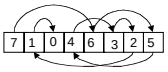
## List-Ranking Problem

- A pointer-jumping application
- Given a chain of pointers, determine the rank of each element in the chain



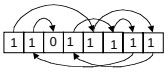
## List-Ranking Problem

- A pointer-jumping application
- Given a chain of pointers, determine the rank of each element in the chain




## List-Ranking Step-by-Step

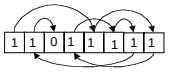
Step 0



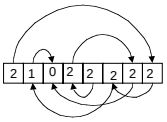

Set your value to 1 if you point to something, 0 otherwise.

## List-Ranking Step-by-Step

Step 1



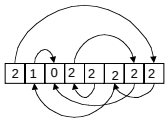

Add your value to the value of your target (if any).



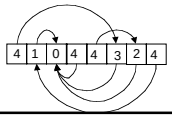

Then point to where your target points.

## List-Ranking Step-by-Step

Step 2

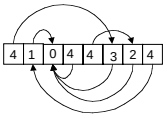



Repeat previous.

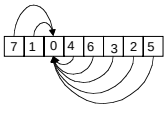



## List-Ranking Step-by-Step

Step 3




Repeat previous.



## Summary

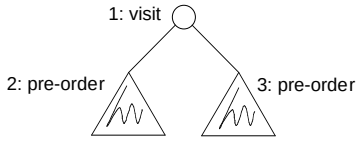
- A list can be ranked in O(log n) time on an EREW PRAM.

## Exercises

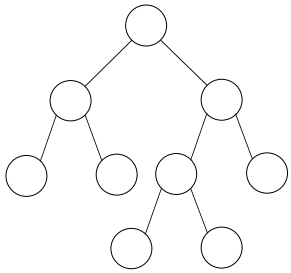
- Show that a *list* can be prefixed-summed in O(log n).

## Pre-Ordering a Tree

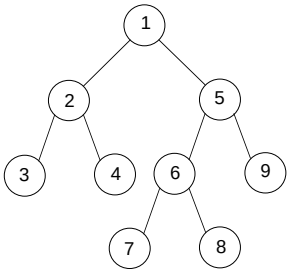
- Recall: Pre-Order:
  - Visit the root
  - Recursively pre-order the left sub-tree.
  - Recursively pre-order the right sub-tree.




## Pre-Order Example



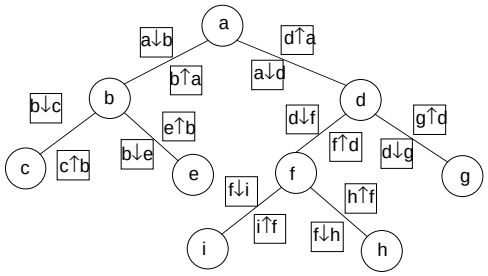
## Pre-Order Example



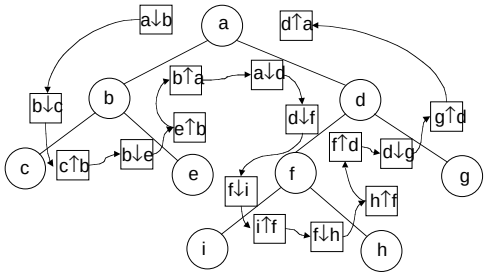

## How to Construct a Pre-Order on a PRAM in O(log n)?

- Create a **list** of nodes, two per arc of the original graph:
  - One node for the arc in the normal (downward) direction
  - A second for the arc in the other direction
- Add a direction-indicator to each node.
- Connect the nodes to represent the original arcs.

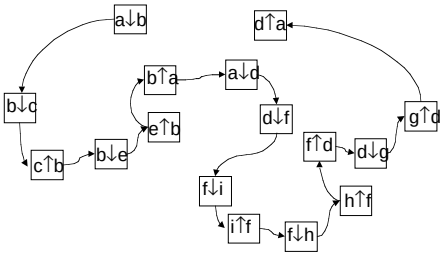
## Pre-Order Example



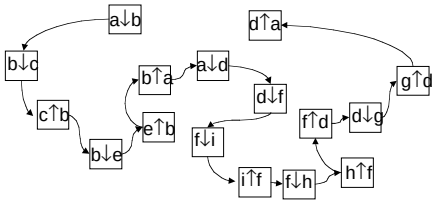

## Pre-Order Example



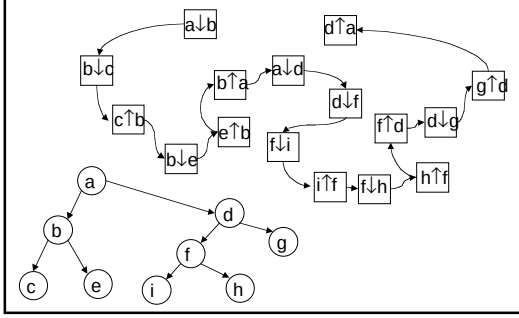

We now have an
alternate representation
from which we can recover the original tree




## Try it




## Try it

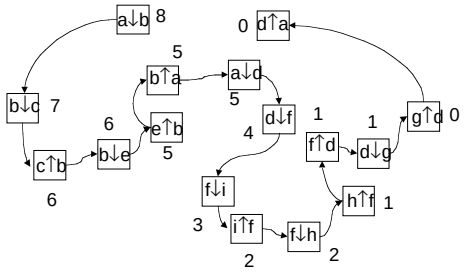



## How much time used so far?

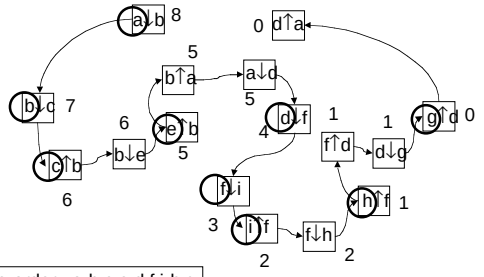
# What now?

## List ranking variation

Count *downward* nodes only when adding values



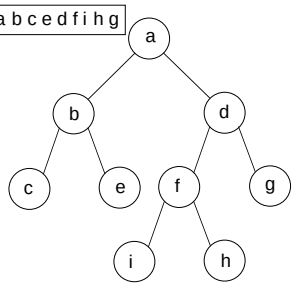

Pre-order of original is reverse of this order
using first component of the root and the *targets* of the ↓ nodes only




Pre-order: a b c e d f i h g

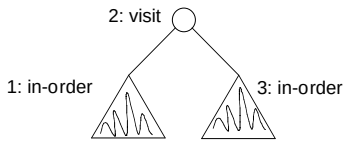
# Check with Original

Pre-order: a b c e d f i h g




# Exercise

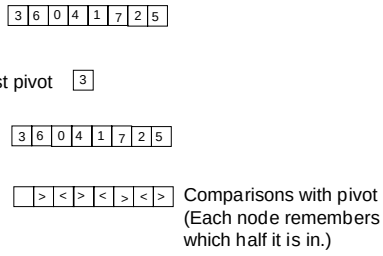
- We just showed that a pre-order traversal can be done in time O(log n). Do the same for an *in-order* traversal.



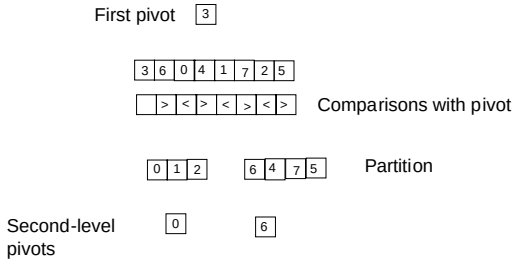

# Application of In-Order Traversal

- A PRAM version of Quicksort
- Construct a tree indicating how the nodes partition (without actually moving any data)
- An in-order traversal of the tree gives the nodes in sorted order.

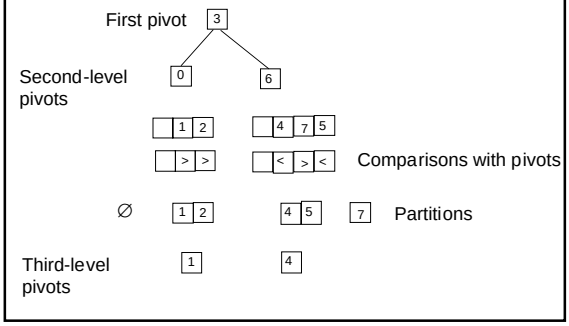
## Quicksort Partition Tree



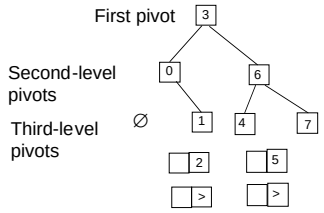

## Quicksort Partition Tree



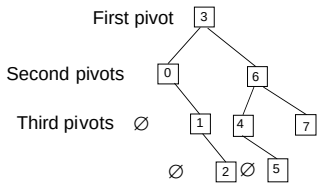

## Quicksort Partition Tree




## Quicksort Partition Tree




## Quicksort Partition Tree




In-order traversal is sorted array

## Exercise

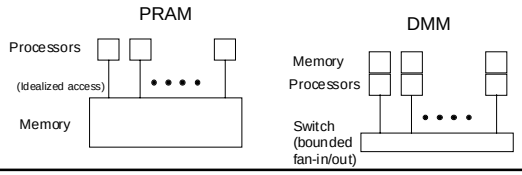
- Assuming that the tree splits fairly evenly across each level, what is the time taken to do this version of Quicksort (assuming the asserted bound for in-order traversal).

## Misc. Notes on PRAM

- Cole's sorting method based on merging: O(log n) on a CREW PRAM (c.f. Gibbons, A.M. & Rytter, W. (1988) Efficient Parallel Algorithms. Cambridge University Press.)
- Parallel recognition of a context-free language: $P(\log^2 n)$ time using $n^6$ processors.
- Other problems/algorithms are known.

## Using PRAM results in real life

- What are some problems of simulating PRAM's on real multiprocessors, say a on a Distributed-Memory Machine (DMM)?




## PRAM ->DMM problems

- Memory conflict resolution at word-level
- Memory conflict resolution at memory-module level
- Communication delays

## PRAM ->DMM possible resolutions

- Memory conflict resolution at word-level:
  Use only EREW model
- Memory conflict resolution at memory-module level
  Split memory into multiple modules;
  Use multiple copies of contended data (must provide for reconciling)
- Communication delays
  Use 2-phase, random routing

## Efficient Simulation of PRAM

- Karp, et al. STOC 1991
- An n log log(n) log*(n) processor CRCW-arb PRAM is simulated on an n-processor DMM (Distributed memory machine).
- Average slowdown O(log log(n) log*(n)).
- Survey of related results: Tim Harris, ACM Computing Surveys, **26**, 2, June 1994, 187-206.