Timing Complexity for Some Parallel Applications

Parallel Time Complexity

- We assume familiarity with O, Ω, and Θ notation.
- Their use is to bound the time complexity as a function of the problem size “n”.

Complexity Example

- Matrix-vector multiplication:
  - n x n matrix (n^2 elements)
  - n element vector
- Assume n processors
  - Every processor has a row of the matrix
  - Suppose row is multiplied by the vector **simultaneously**
  - If it takes O(n) to multiply one row then overall

$$T(n) \in O(n)$$

Effort or “Cost”

- Let T_p be the time a particular algorithm takes on p processors.
- Assuming the processors don’t have other work to do, the **effort** expended is thus

$$p*T_p$$

Cost Optimality

A **cost-optimal parallel algorithm** is defined to be one in which the **effort**, as a function of problem size, is bounded by a **constant** times the **sequential effort**:

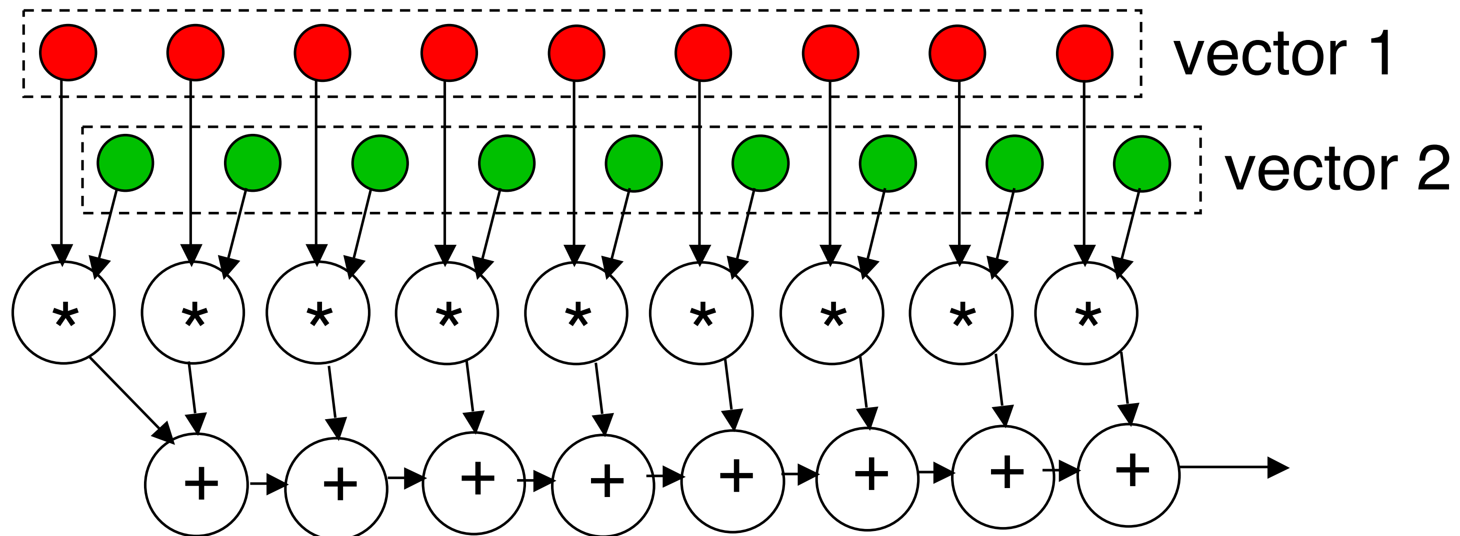
$$\text{Effort}_{\text{parallel}}(n) \in O(\text{Effort}_{\text{sequential}}(n))$$

Is the matrix-vector algorithm alluded to cost-optimal?

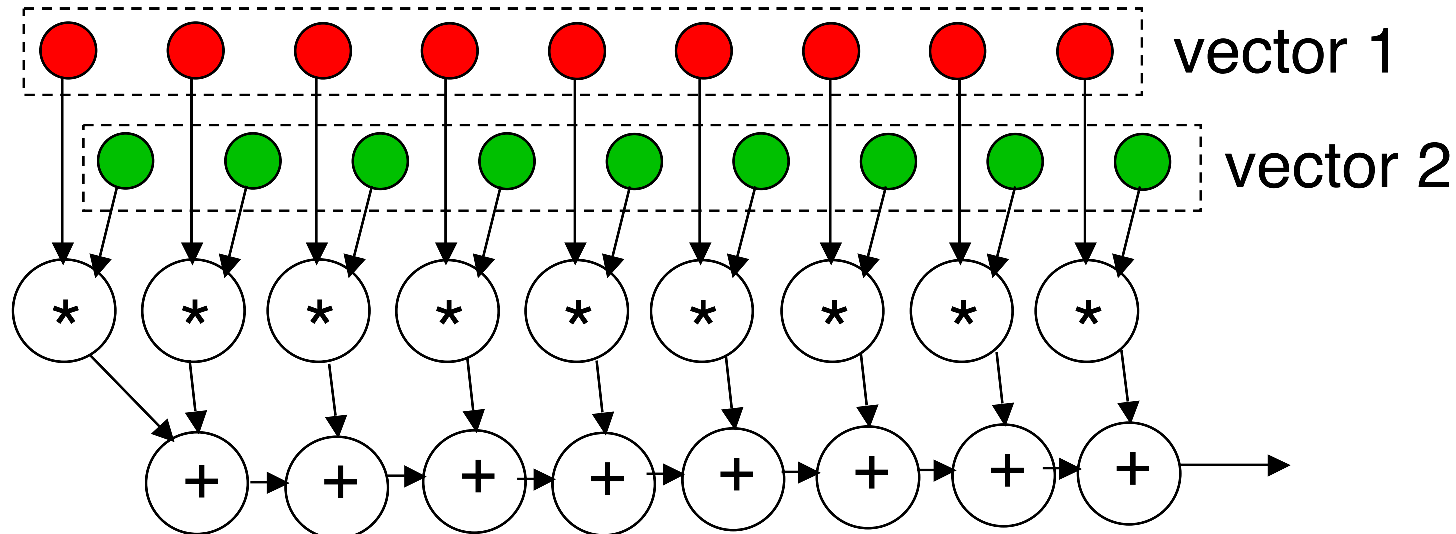
Dependency Graphs

Using **Dependency Graphs** to Illustrate Algorithms

- A vector inner product can be shown thus (ignoring distribution for now):



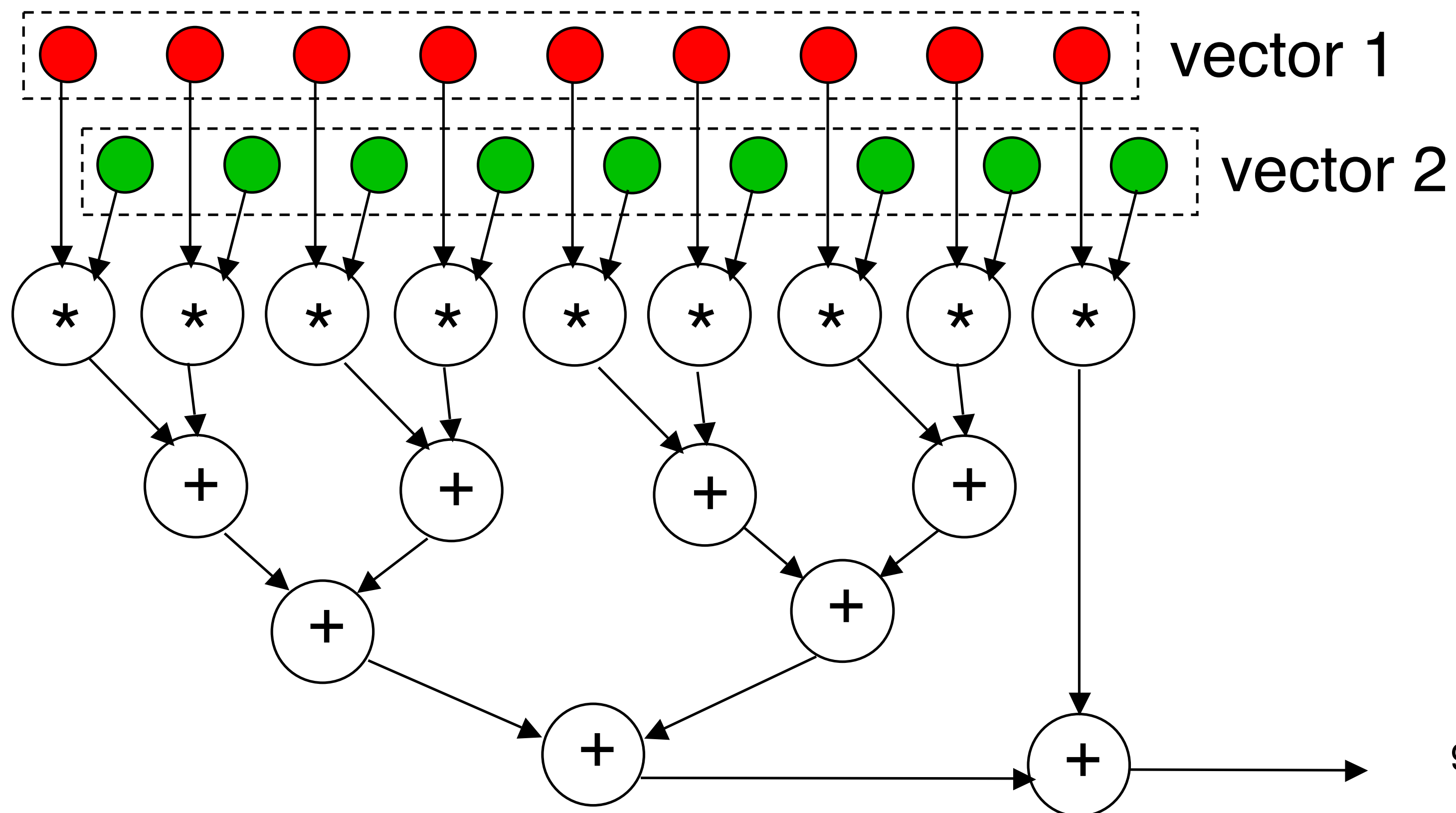
Using **Dependency Graphs** to Illustrate Algorithms



- Assume **unit time** for each operation.
- The time is proportional to path length.
- The **longest path** length for an n-element vector is O(n), similarly to serial.

Using Graphs to Illustrate Algorithms

- Restructuring the + nodes as a **tree** gives us **faster** performance on n processors.



Algorithm Analysis

- The previous tree gives us O(log n) time on n processors.

- Is this inner-product cost optimal?

Scaling Down Processors

- As the size n of the vector grows very large, we can divide the additions up among p processors, $p \ll n$,

  adding the elements within each processor sequentially and **only funnelling to a tree at the end**.

- The time for large n is then dominated by the sequential adds, which is $O(n)$ for a given p.

Generalization of Scale-down

- Whenever the total number of operations (including communication as an operation) in the parallel case is linearly proportional to the serial complexity, we can achieve cost optimality by scaling down the number of processors relative to the problem size.

- The general concept is captured by **Brent's Lemma**.

Brent's Lemma

- If an algorithm A entails m operations and can be done in parallel time t with *some* number of processors,

  then p processors can execute A in time t'

  $$t' = t + (m-t)/p$$

  assuming any added scheduling time can be ignored.

Brent's Lemma Summarized

- t = time on *some* number of processors (could be arbitrarily-large)
- m = number of operations
  (each taking unit time)
- time on p processors is $\leq t + (m-t)/p$

Application of Brent's Lemma

- To achieve cost optimality for **vector inner product**, use **n/(log n)** processors.
- We observed that product can be done in time O(log n) with arbitrarily-many processors.
- Brent's lemma says it can be done with p = n/(log n) processors in time

$$\underbrace{\log n}_{t} + (\underbrace{2n-1}_{m} - \underbrace{\log n}_{t}) / \underbrace{(n/\log n)}_{p}$$

Application of Brent's Lemma

$\log n + (2n-1-\log n) / (n/\log n)$

$= \log n + 2 \log n - (\log n)/n - (\log n)^2/n$

which is $O(\log n)$.

Proof of Brent's Lemma (1)

- Consider the graph of the algorithm done with some number of processors in time t.
- Let s_i be the number of operations done at the ith **level**, i.e. at "time" i.
- On p processors, we can **reschedule** the s_i operations in time $\lceil s_i/p \rceil$ since they are **independent**.

Proof of Brent's Lemma (2)

- The total computation can therefore be done on p processors in time

$$\text{sum}(i = 1 \text{ to } t, \lceil s_i/p \rceil)$$

Proof of Brent's Lemma (3)

$\sum \lceil s_i/p \rceil$ (where the summation is i = 1 to t)

is bounded by

$\sum (s_i+p-1)/p$

$= \sum s_i/p$
$+ \sum p/p$
$- \sum 1/p$

$= m/p + t - t/p$
$= t + (m-t)/p$, as stated.

Illustration of Brent

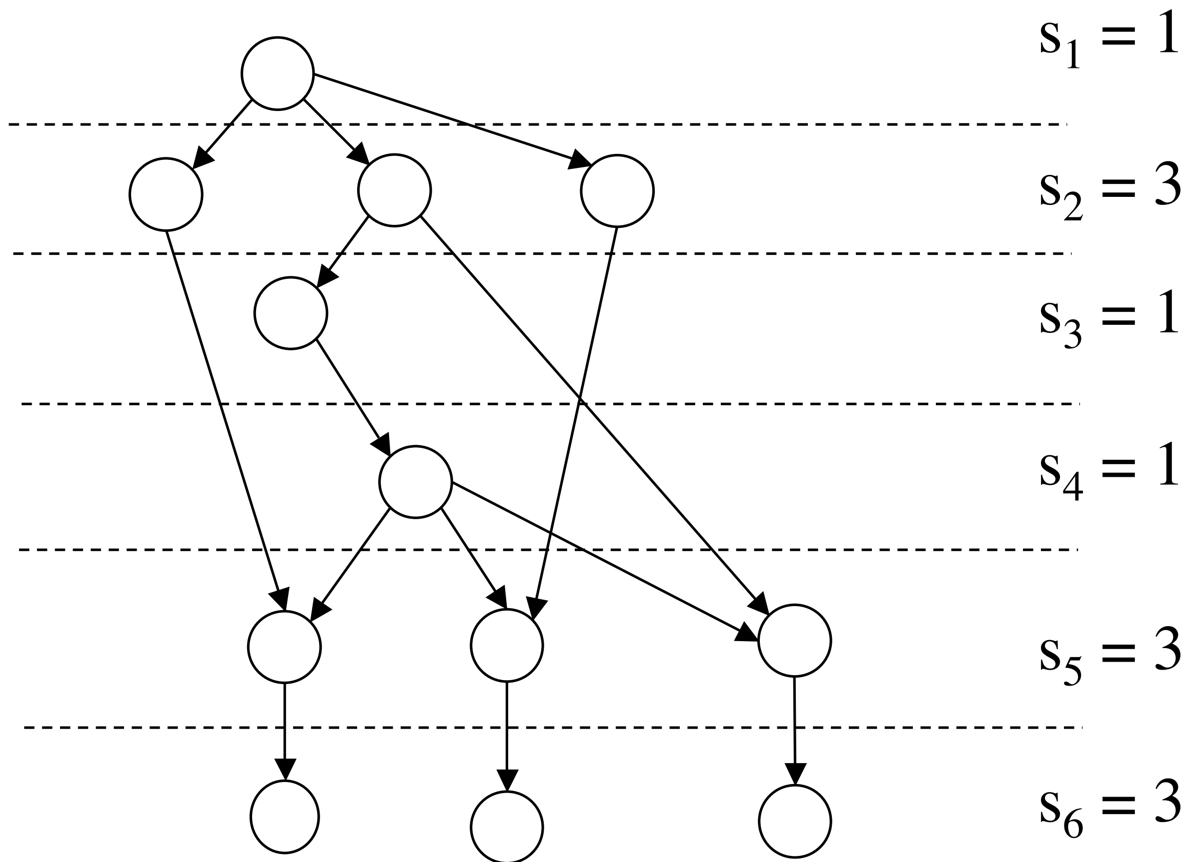
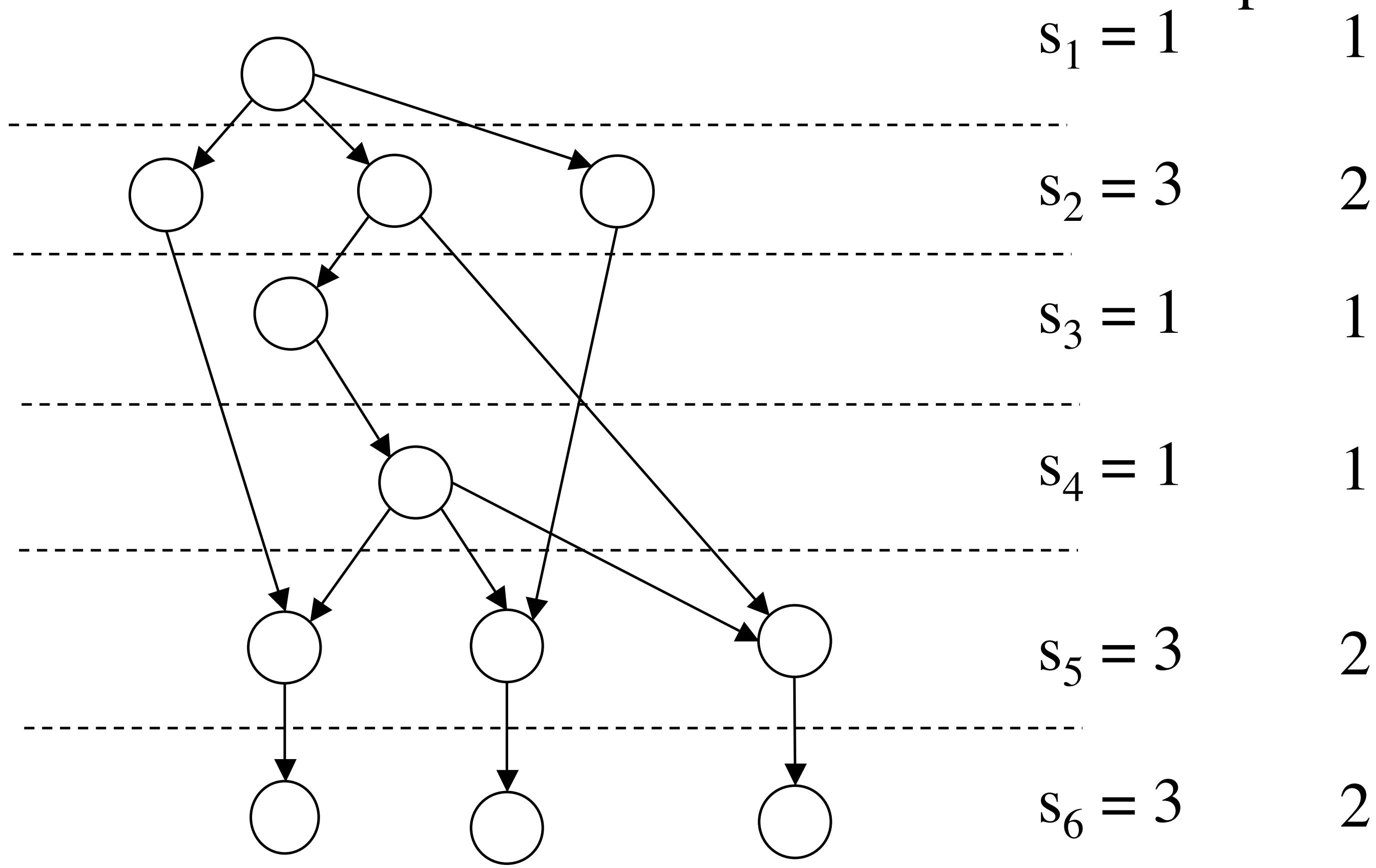


Illustration of Brent for 2 processors

Brent's bound predicts: $t + (m-t)/p = 6 + (12-6)/2 = 9$

time on 2 processors



total time = 9

Graph Exercise

- By the **prefix sum problem**, we mean that of computing from an array

  $x_0, x_1, x_2, \ldots, x_{n-1}$

  the array

  $(x_0), (x_0+x_1), (x_0+x_1+x_2), \ldots,(x_0+x_1+x_2+\ldots+ x_{n-1})$
- Can this problem be sped up using parallelism?
- Is there a cost optimal version?

Related Side-Topic: Scheduling Anomalies

Ron Graham, 1960's

Deterministic Scheduling

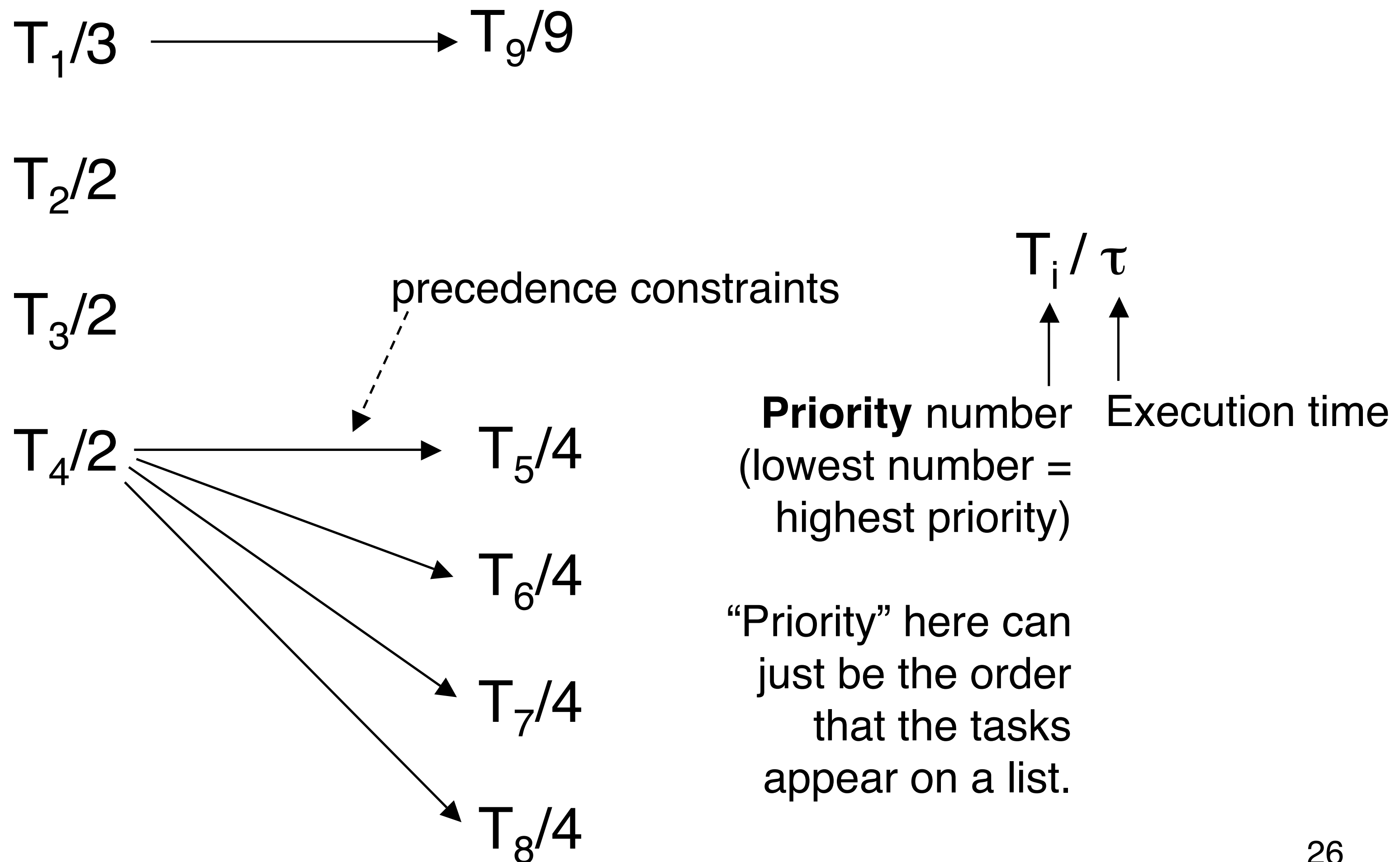
- Assume times of tasks (and communication) are known, which they often aren't.
- Problem is NP-hard for all but the most trivial classes of assumptions.
- Unexpected scheduling anomalies.

Scheduling Anomalies

- The following are *intuitively expected* to reduce overall execution time:
  - Reducing execution times of individual tasks
  - Relaxing precedence constraints between tasks
  - Adding more processors

- But for some graphs, these can actually *increase* the execution time.

Priority Scheduling Anomalies



Consider Scheduling on 3 processors

$T_1/3$

$T_2/2$

$T_3/2$

$T_1/3 \rightarrow T_9/9$

$T_2/2$

$T_3/2$

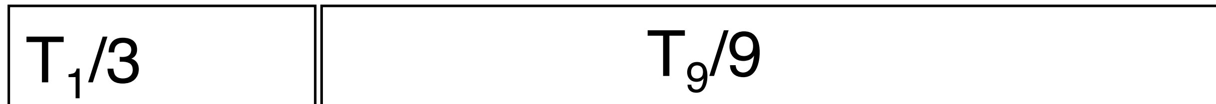
$T_4/2 \rightarrow T_5/4$

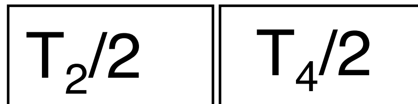
$T_4/2 \rightarrow T_6/4$

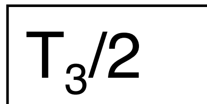
$T_4/2 \rightarrow T_7/4$

$T_4/2 \rightarrow T_8/4$

Consider Scheduling on 3 processors







$T_1/3 \rightarrow T_9/9$

$T_2/2$

$T_3/2$

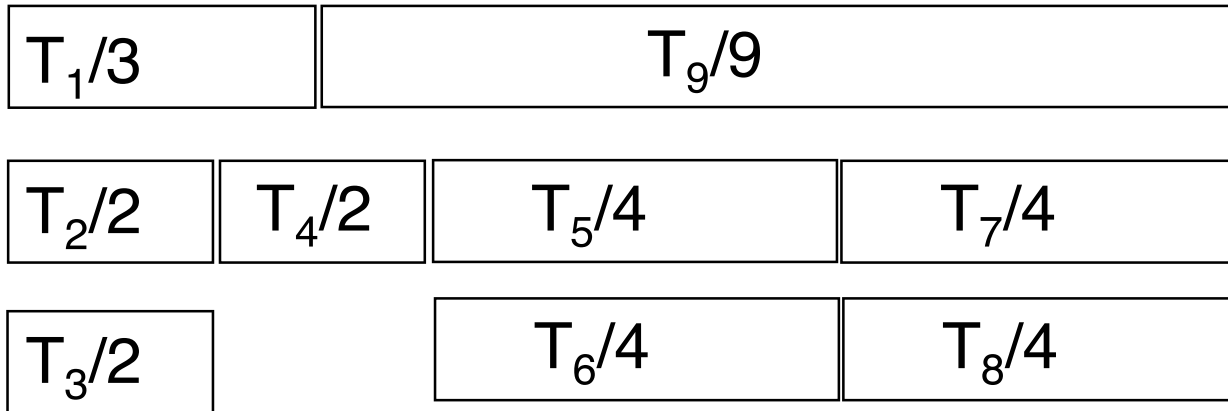
$T_4/2 \rightarrow T_5/4$

$T_4/2 \rightarrow T_6/4$

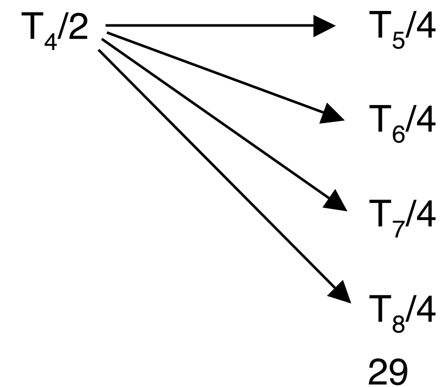
$T_4/2 \rightarrow T_7/4$

$T_4/2 \rightarrow T_8/4$

Consider Scheduling on 3 processors



Total time = 12



Consider Scheduling on 4 processors

$T_1/3$

$T_2/2$

$T_3/2$

$T_4/2$

$T_1/3 \rightarrow T_9/9$

$T_2/2$

$T_3/2$

$T_4/2 \rightarrow T_5/4$

$T_4/2 \rightarrow T_6/4$

$T_4/2 \rightarrow T_7/4$

$T_4/2 \rightarrow T_8/4$

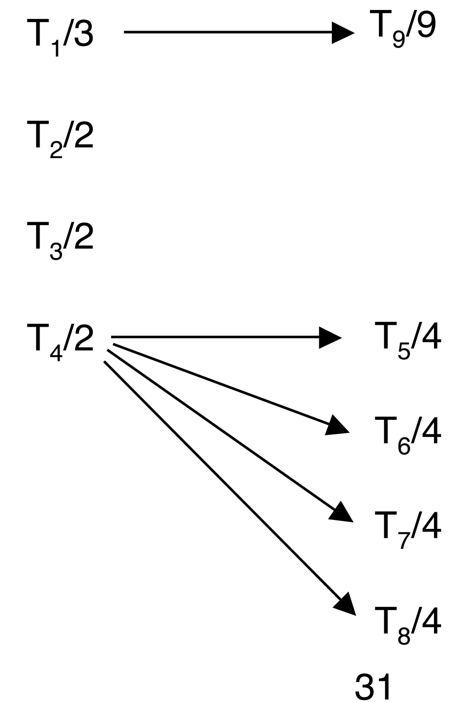
Consider Scheduling on 4 processors

$T_1/3$ | $T_8/4$

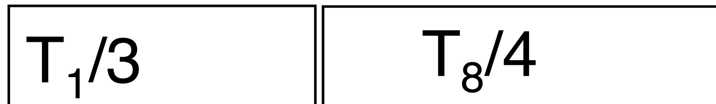
$T_2/2$ | $T_5/4$

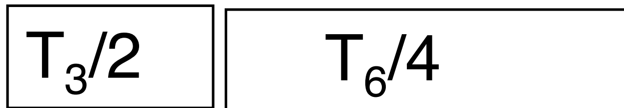
$T_3/2$ | $T_6/4$

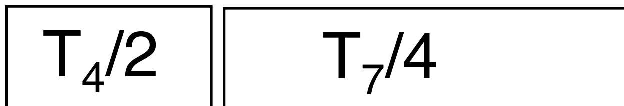
$T_4/2$ | $T_7/4$



Consider Scheduling on 4 processors







Total time = 15
vs. 12 on 3 proc.

Consider Relaxing Constraints

$T_1/3$ → $T_9/9$

$T_2/2$

$T_3/2$

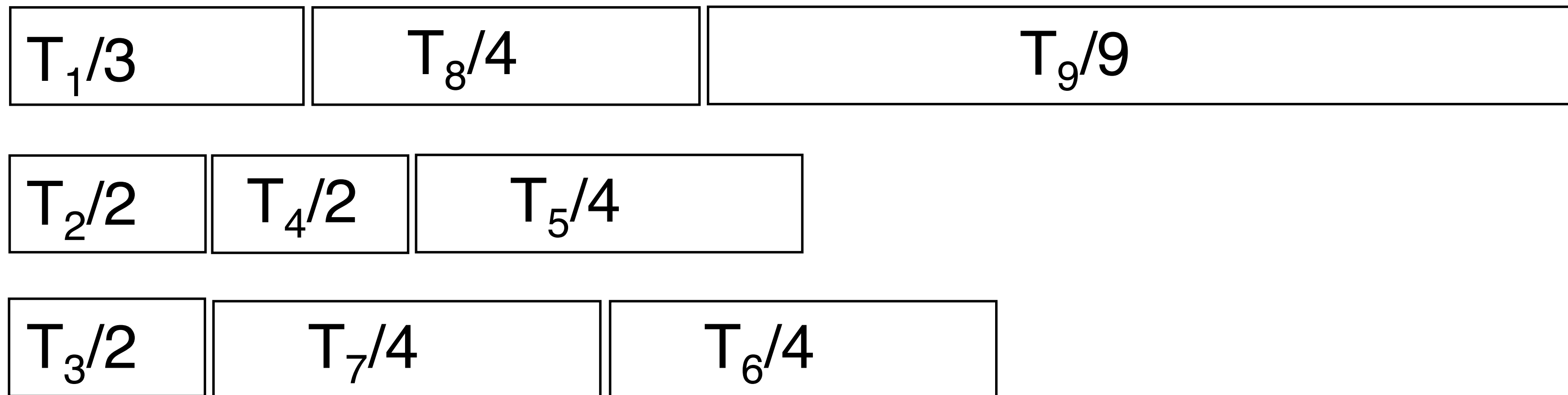
$T_4/2$ → $T_5/4$

$T_4/2$ → $T_6/4$

$T_4/2$ ⇢ $T_7/4$ (remove)

$T_4/2$ ⇢ $T_8/4$ (remove)

Consider the relaxed constraints on 3 processors



Total time = 16

vs. 12 before constraints relaxed

Cause of Anomalies

- Obviously the anomalies are caused by the use of the **priority or list rule** in scheduling:
  - This rule is cheap to implement.
  - It does not take into account optimizations that would be possible by violating strict priority.

- In general, finding true optimum would entail a search, which tends to be much more expensive.

Bounds on Anomalies

- Let unprimed designate times for system with relaxed constraints and shorter individual times. Then
- Time(p')/Time(p) $\leq$ 1 + (p'-1)/p,
  where p $\geq$ p' are numbers of processors.

- Example: Time(2)/Time(3) $\leq$ 4/3.
- Worst case: Time(p')/Time(p) $<$ 2.

- Brent's lemma says $t' = t + (m-t)/p$
- Graham says $t' \leq t + t(p'-1)/p$

PRAM Model

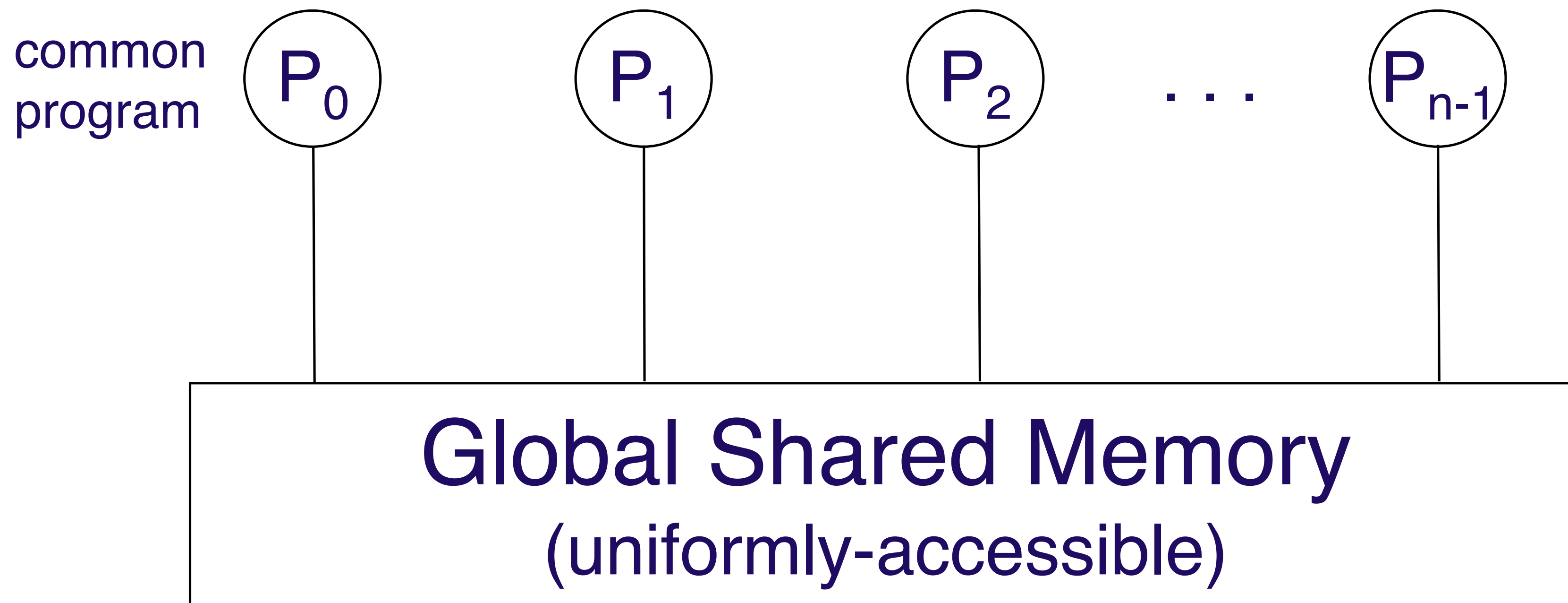
Useful as a Baseline for Time Complexity

PRAM Model

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

PRAM Diagram

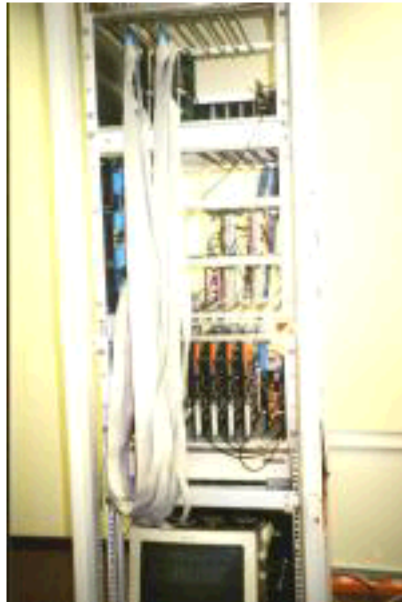
Processors (with local dedicated memory)
n adjusted to problem size

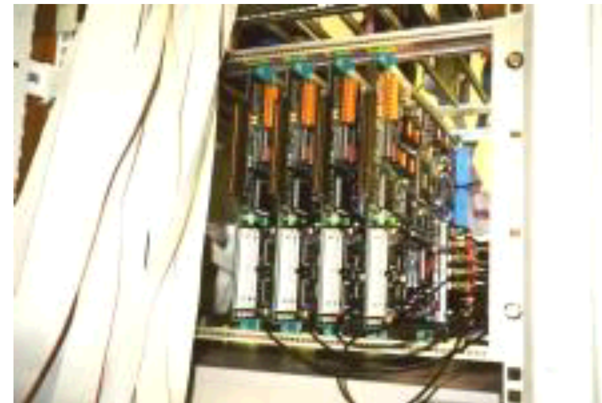


Use of PRAM Model

- 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 was being constructed

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





The project goal was to achieve a 64 physical (2048 virtual) processor machine with 2 GByte of global memory and 256 hard disks.

Current Status of SB_PRAM

Project overview

The SB-PRAM is a MIMD parallel computer with shared address space and uniform memory access time (CRCW-PRAM-Model). Processors and memory modules are connected by a butterfly network. Each SB-PRAM processor module consists of a custom ASIC processor with extended Berkeley-RISC instruction set, a local program memory and SCSI interface. Network nodes and memory modules provide hardware support for concurrent read and concurrent write memory access and parallel prefix operations. Network latency is hidden by pipelining several virtual processors (hardware threads with zero switching overhead) on one physical processor. Network congestion is reduced by hashed addresses. Hot spots are avoided by combining.

Project status

We have succeeded in building a 64 physical (2048 virtual) processor machine with 4 GByte of global memory. The machine is currently switched off.

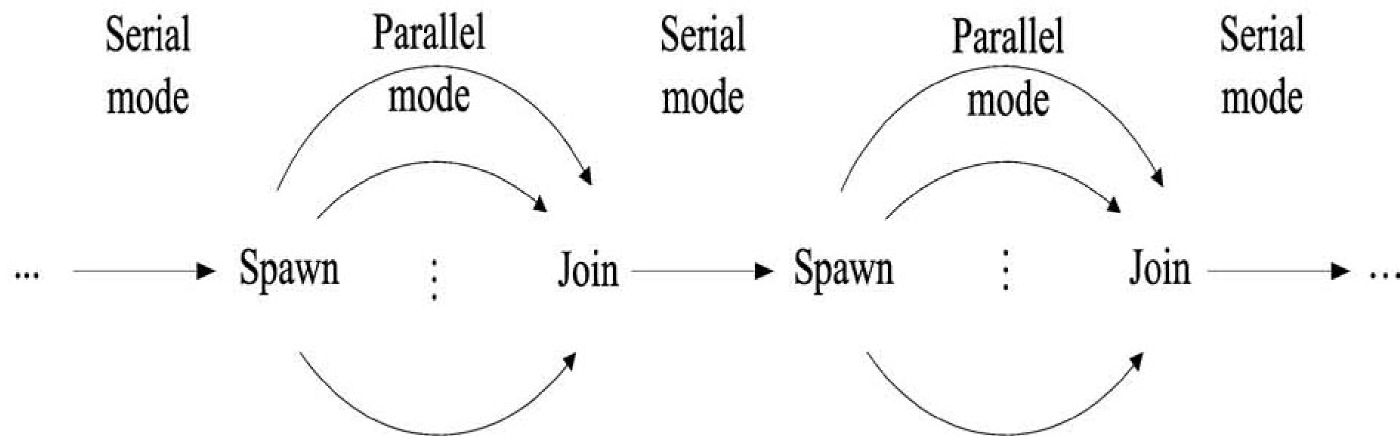
Project members

Prof. Dr. Wolfgang J. Paul

Vishkin's xmt Machine

http://www.umiacs.umd.edu/~vishkin/XMT/index.shtml

- XMT = "Explicit Multithreading"
- PRAM-on-chip Vision of supercomputing
- FPGA version implemented
- Language and xmtc compiler



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

- Generally concurrent reading and writing to a single location is disallowed.
- **EREW** (**E**xclusive-**R**ead, **E**xclusive-**W**rite) Concurrent reading or writing to a location is disallowed.
- **CREW** (**C**oncurrent-**R**ead, **E**xclusive-**W**rite) Concurrent writing to a location is disallowed.
- **CRCW** (**C**oncurrent-**R**ead, **C**oncurrent-**W**rite) 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.

- **Combining**: 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 use as little machinery 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:
  - 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).

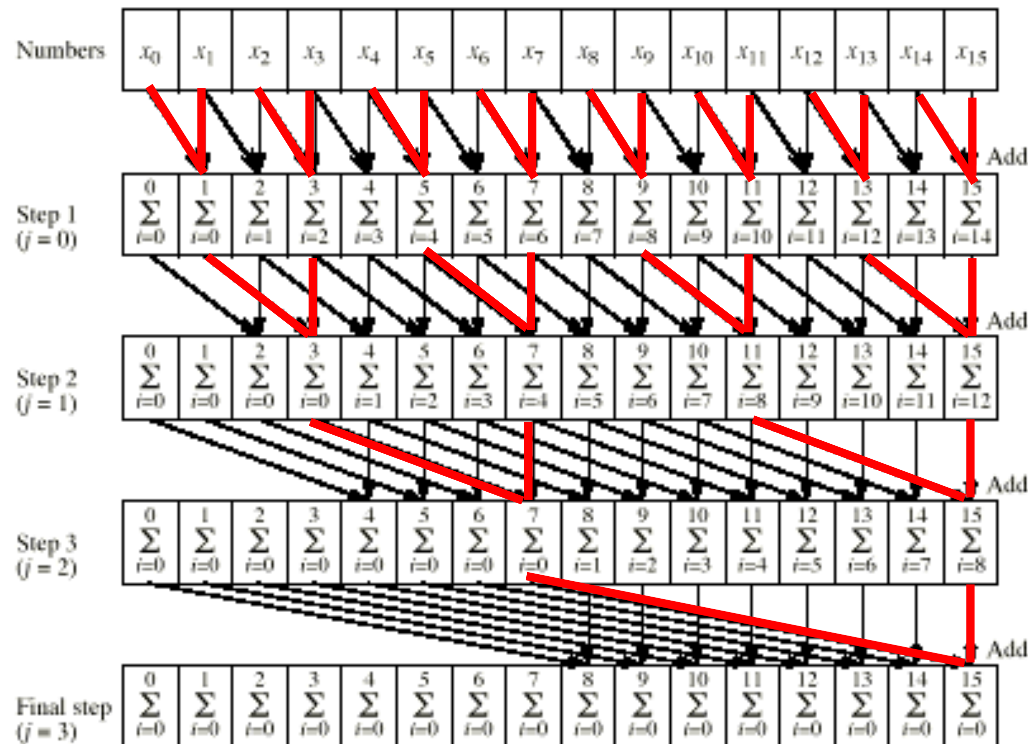


Figure 6.8 Data parallel prefix sum operation.

PRAM Prefix Sum

- Obviously an EREW PRAM can compute **any prefix-sum** type computation in O(log n).
- More processors are busy here 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)
- but using n^2 processors
- (so hardly cost-optimal)

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.

What happened to $\Omega(\log n)$?

- 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 $\Omega(\log n)$.
  [$O(n^2)$ 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)**.

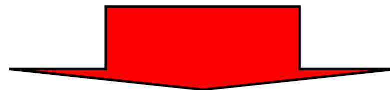
- The proof introduces an intermediate parallel sorting step.

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**.

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p

1	0	0	0	1	0	0	0	1	0	0	0	0	0	1	1



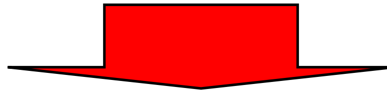
a	e	i	o												

Technique

- Use prefix-sum + indexing (parallel).
- Compute the prefix sum of the bit array.
- Use the computed values as **indexes** of where to store the corresponding item.

Array Compression Exposed

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p
1	0	0	0	1	0	0	0	1	0	0	0	0	0	1	0

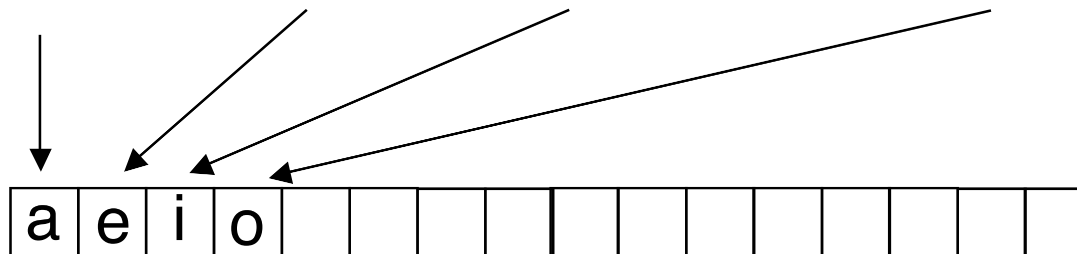


Prefix sum

1	1	1	1	2	2	2	2	3	3	3	3	3	3	4	5

1	1	1	1	2	2	2	2	3	3	3	3	3	3	4	4
a				e				i						o	

Transitions (where to use indices)

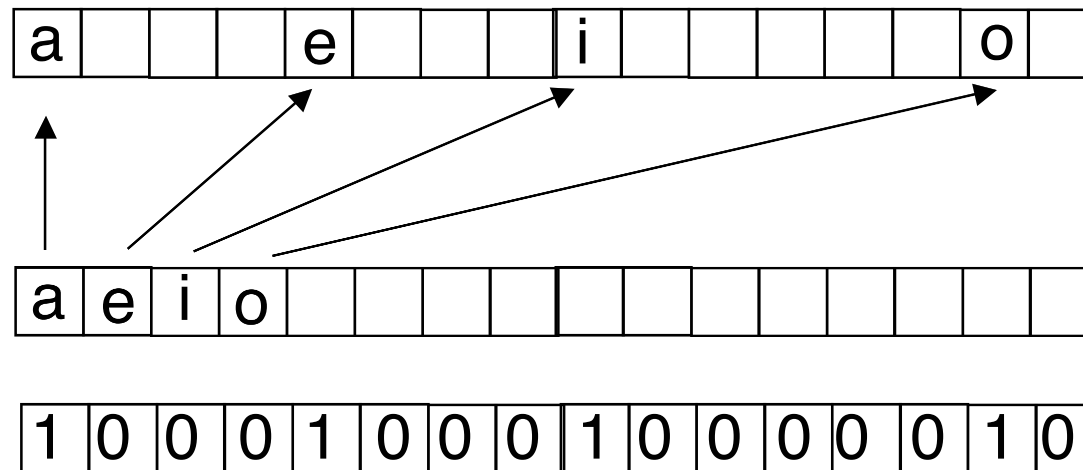


Parallel stores

Exercise

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

Array Expansion



Parallel stores

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

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

Exercise

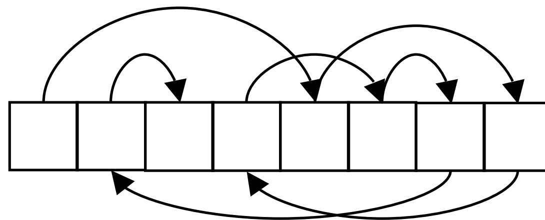
- What is an upper bound for ***sorting*** using parallel merging?

Using Pointers in a PRAM

- “Pointer Jumping” technique is common.
- As with prefix sum, this has many uses.
- Basic idea: in a chain of pointers stored in the shared 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 to data “N hops away” now,
    it will point to data “2N hops away” on the next step
  - because concatenating two N-hop chains gives a 2N-hop chain.

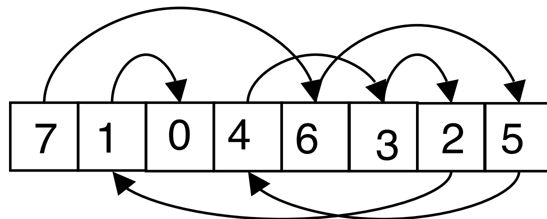
List-Ranking Problem

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



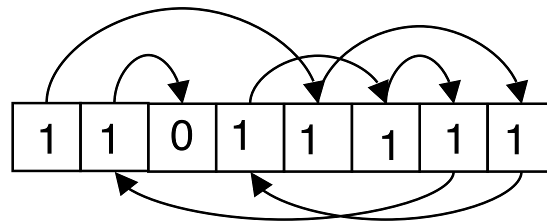
List-Ranking Problem

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



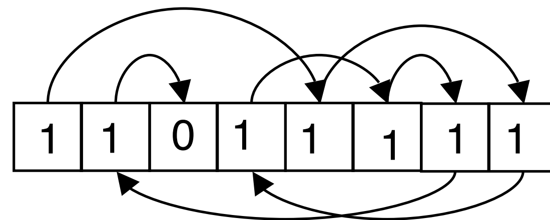
List-Ranking Step-by-Step

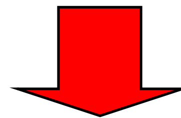
Step 0: If a node has no target, its value is 0;
otherwise its value is 1.

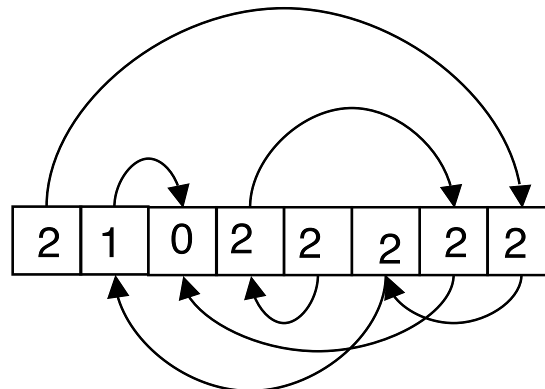


List-Ranking Step-by-Step

Step 1: Add the value of the target to the nodes value and replace your pointer with your target's pointer, if is non-null.

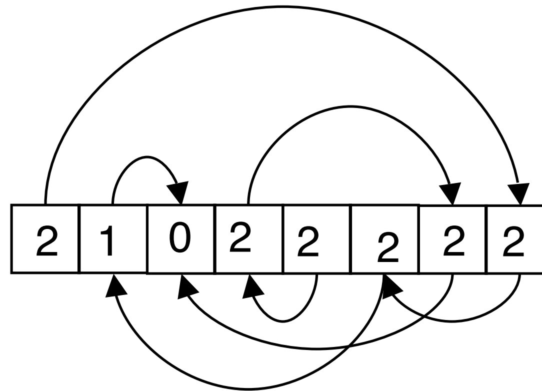


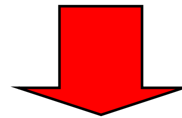


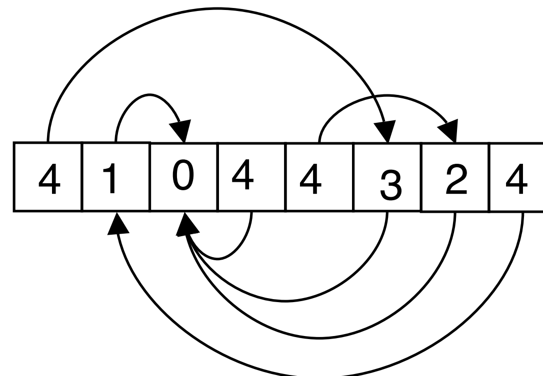


List-Ranking Step-by-Step

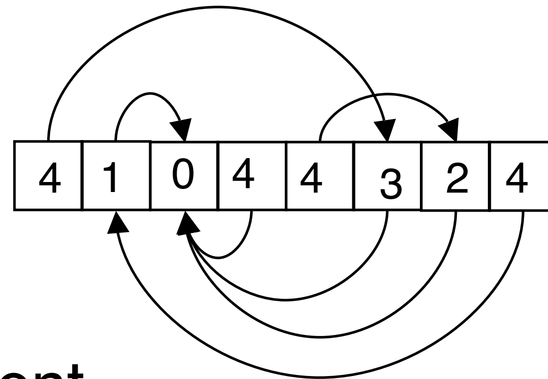
Repeat Step 2



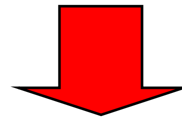


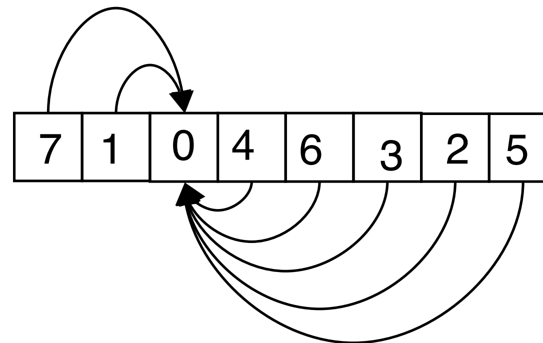


List-Ranking Step-by-Step



until every element points to the 0 element





Summary

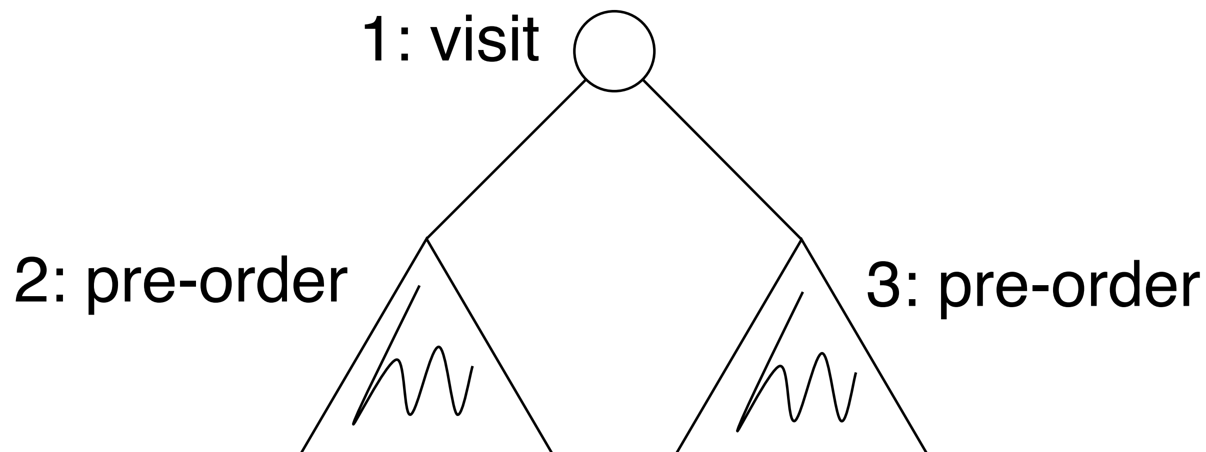
- A list can be ranked in O(log n) time on a PRAM.

Exercises

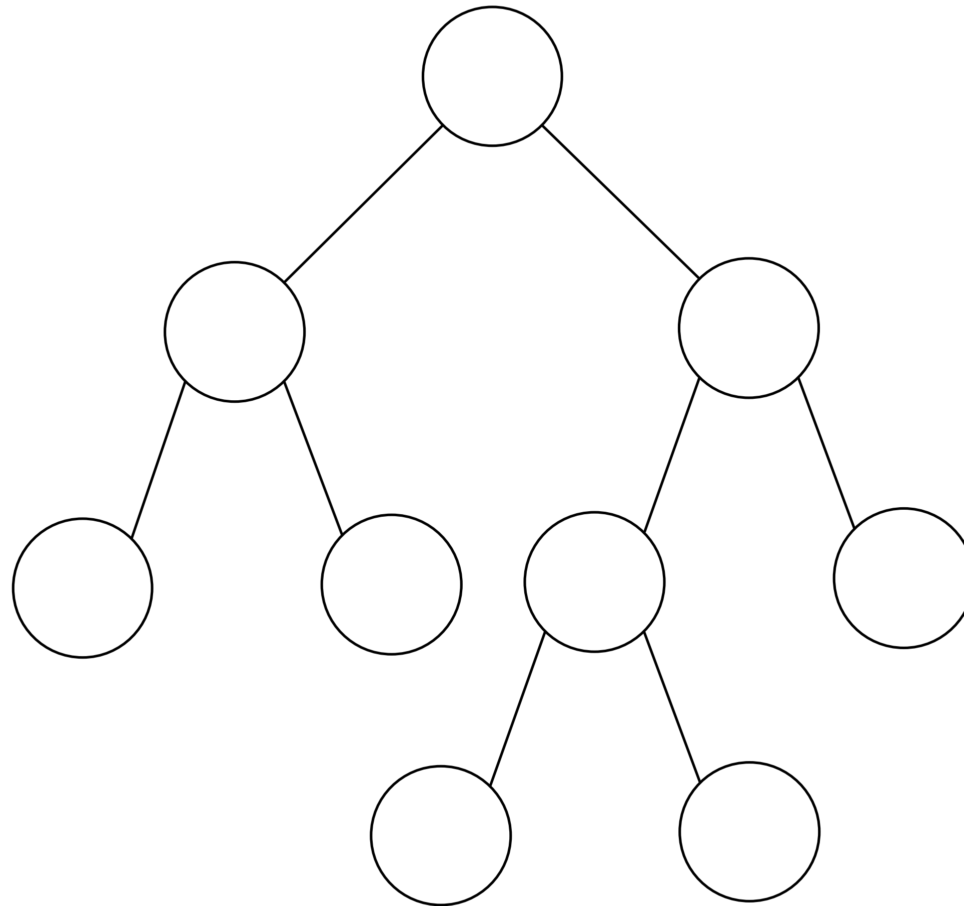
- Show that the elements of a *list* can be prefixed-summed in O(log n).

Pre-Ordering a Tree

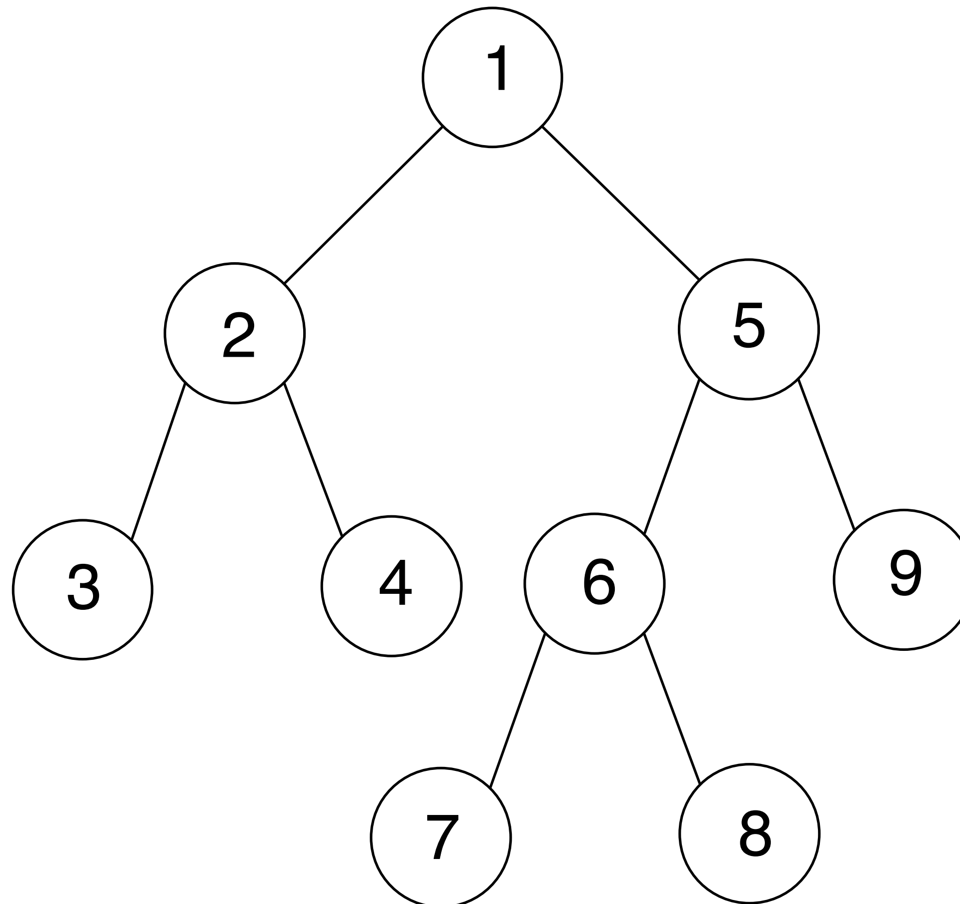
- Recall: Pre-Order:
  - Visit the root
  - Visit recursively the left sub-tree in pre-order
  - Visit recursively the right sub-tree in pre-order



Pre-Order Example



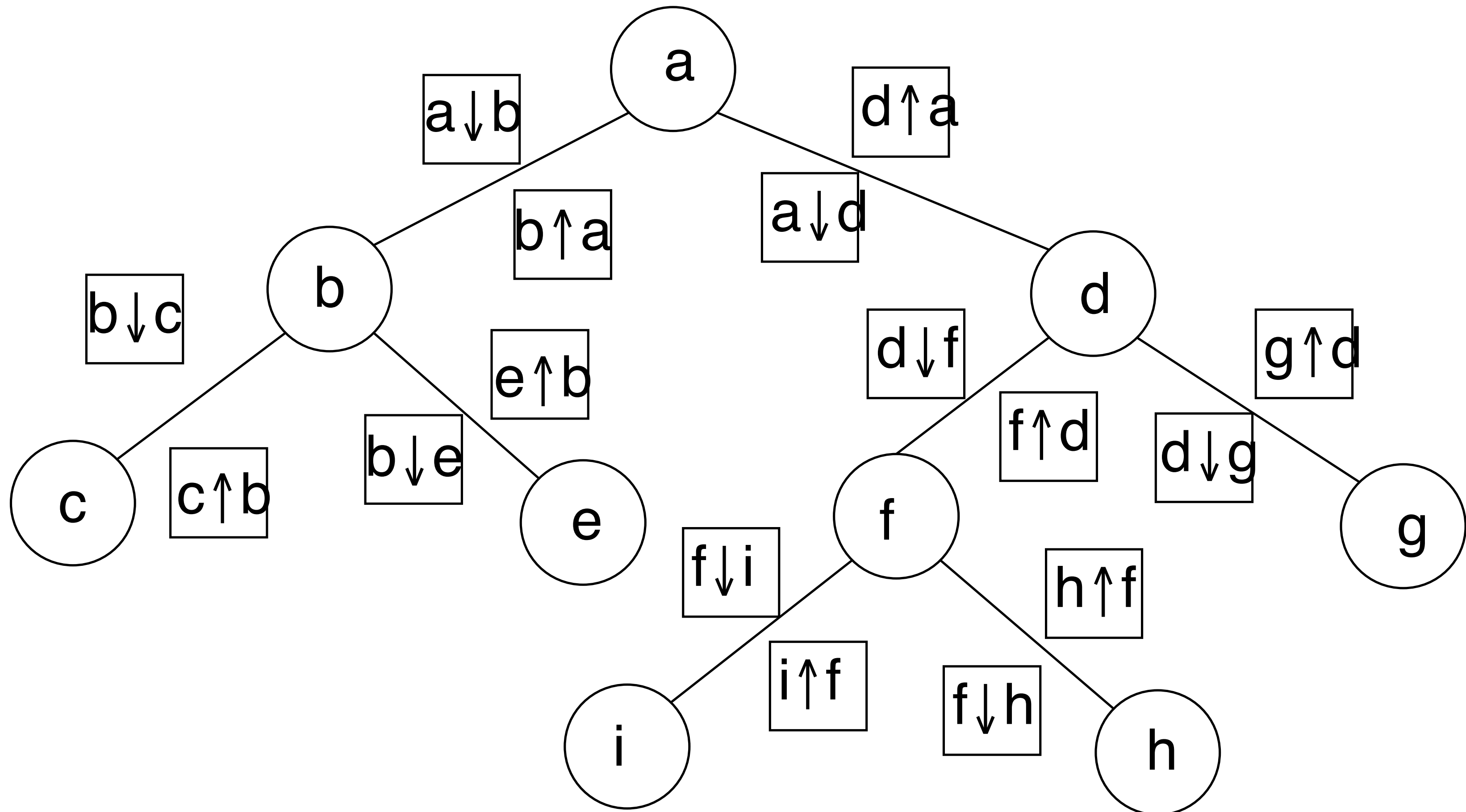
Pre-Order Example



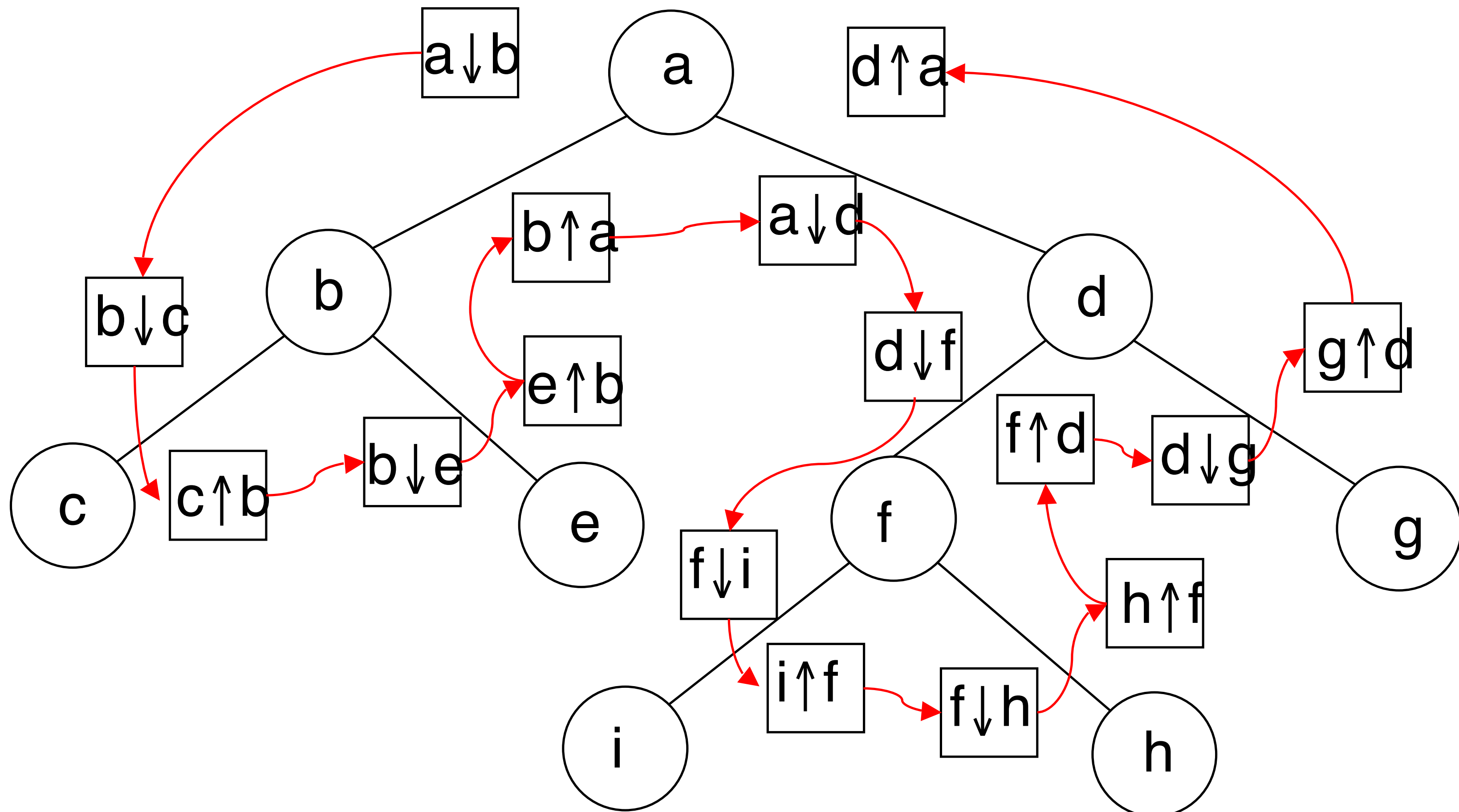
How to Construct a Pre-Order on a PRAM in O(log n)?

- 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 new node.
- Connect the new nodes to represent the original arcs.

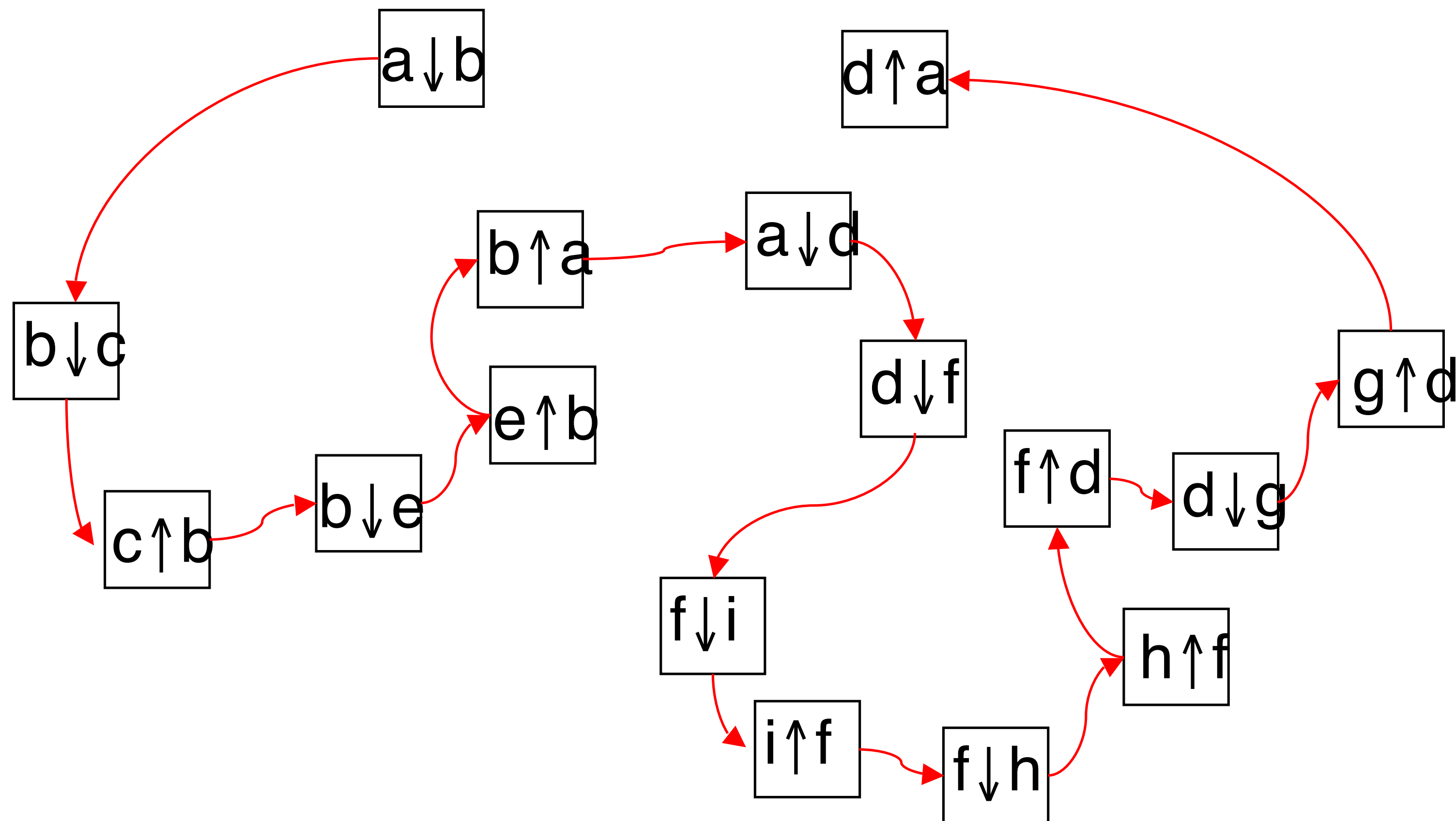
Pre-Order Example



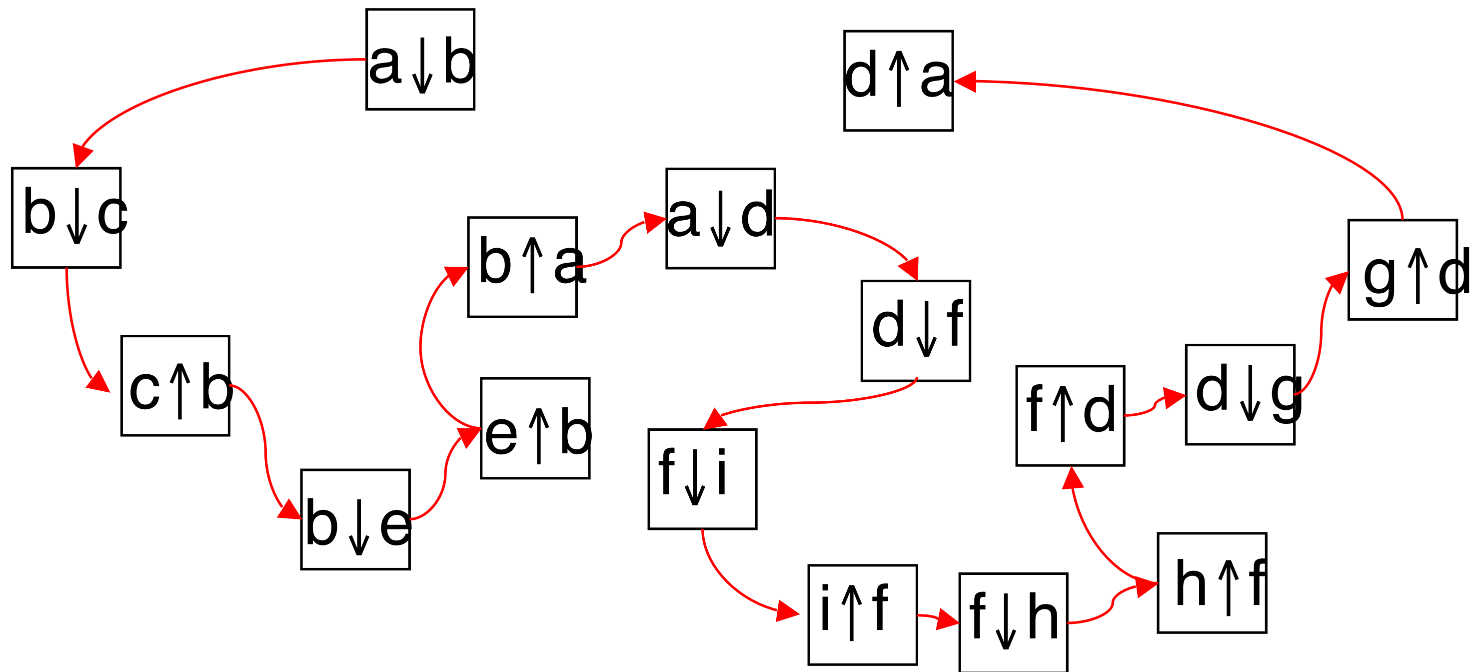
Pre-Order Example connect new nodes step



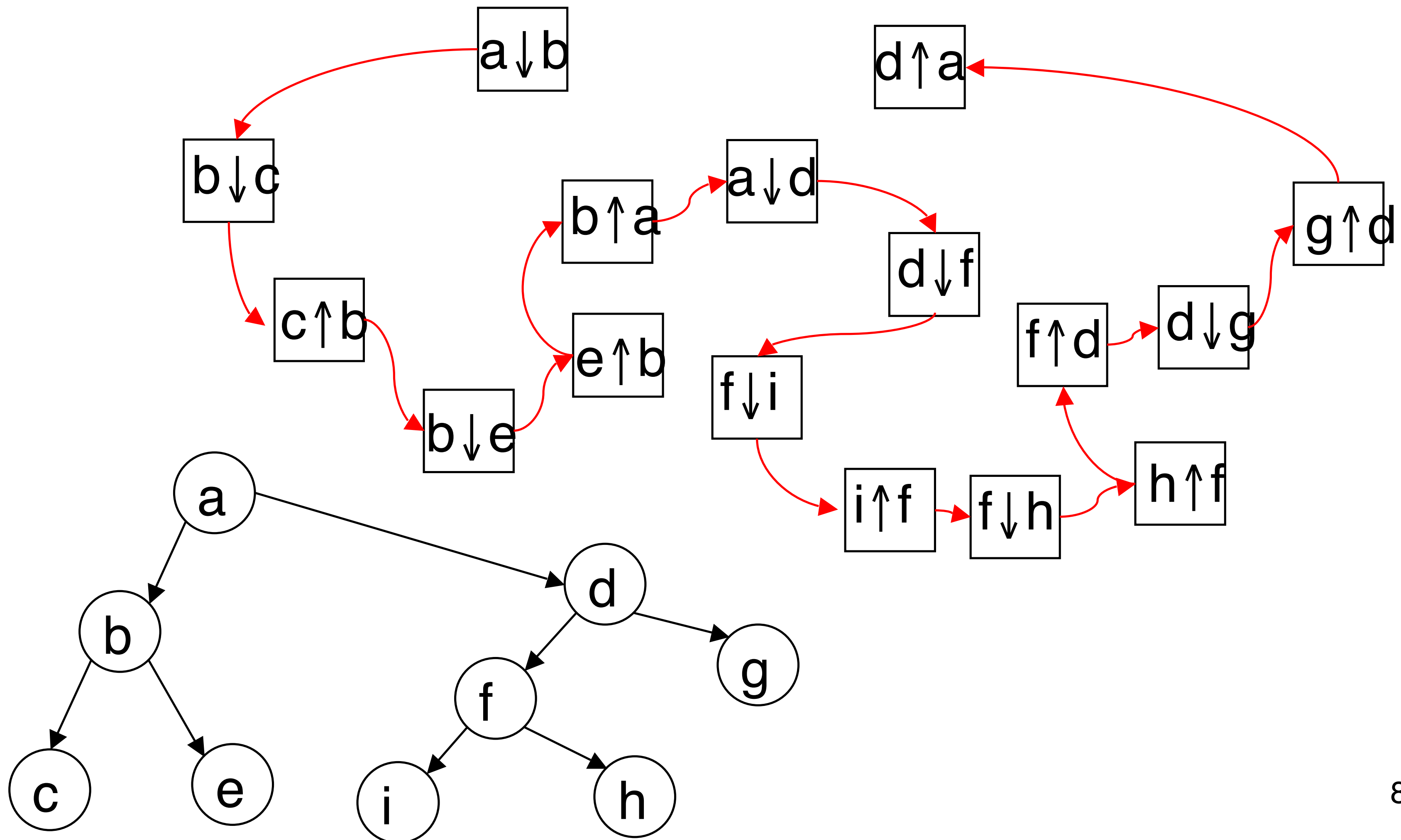
We now have an alternate representation from which we can recover the original tree



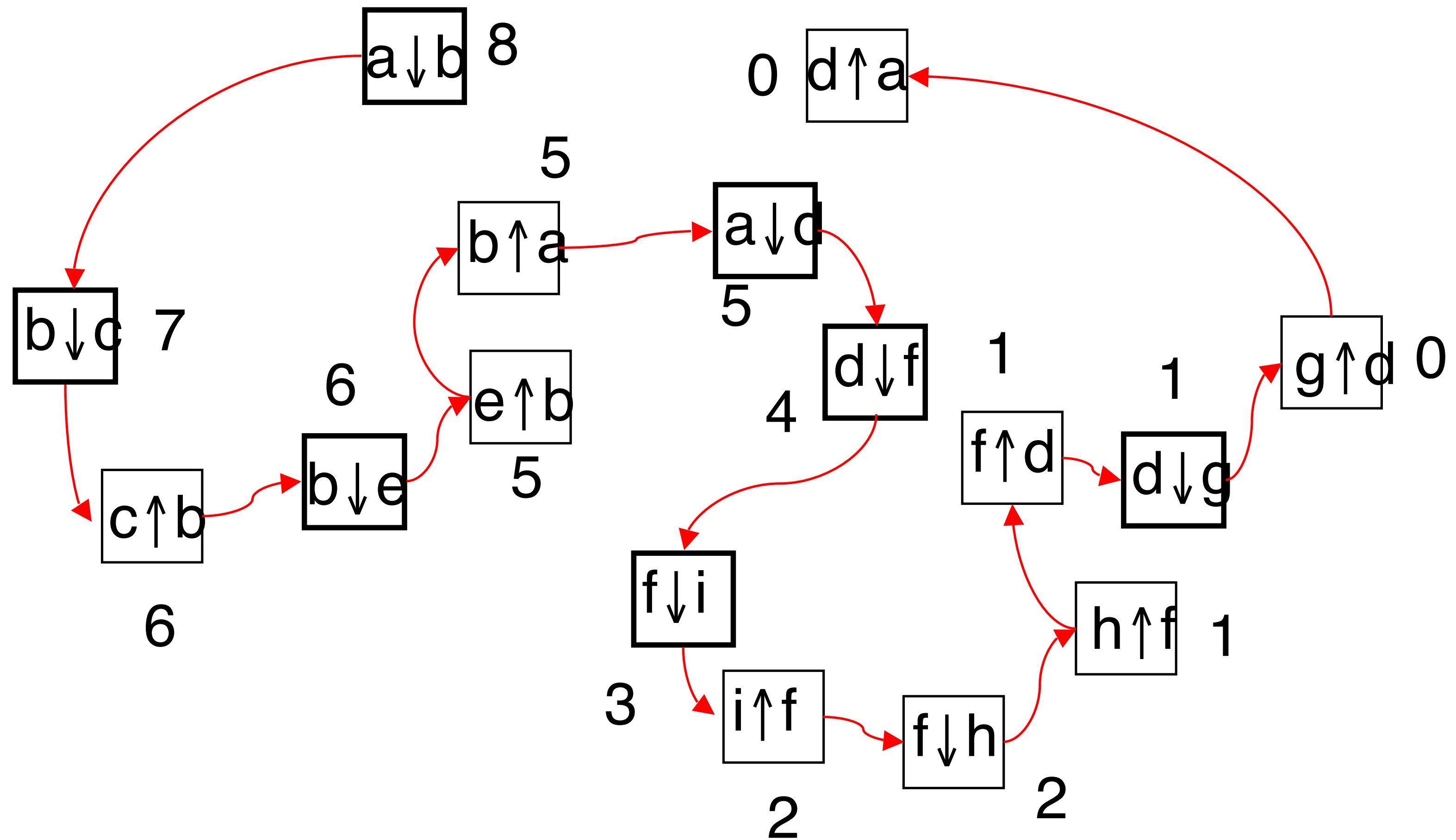
Try it



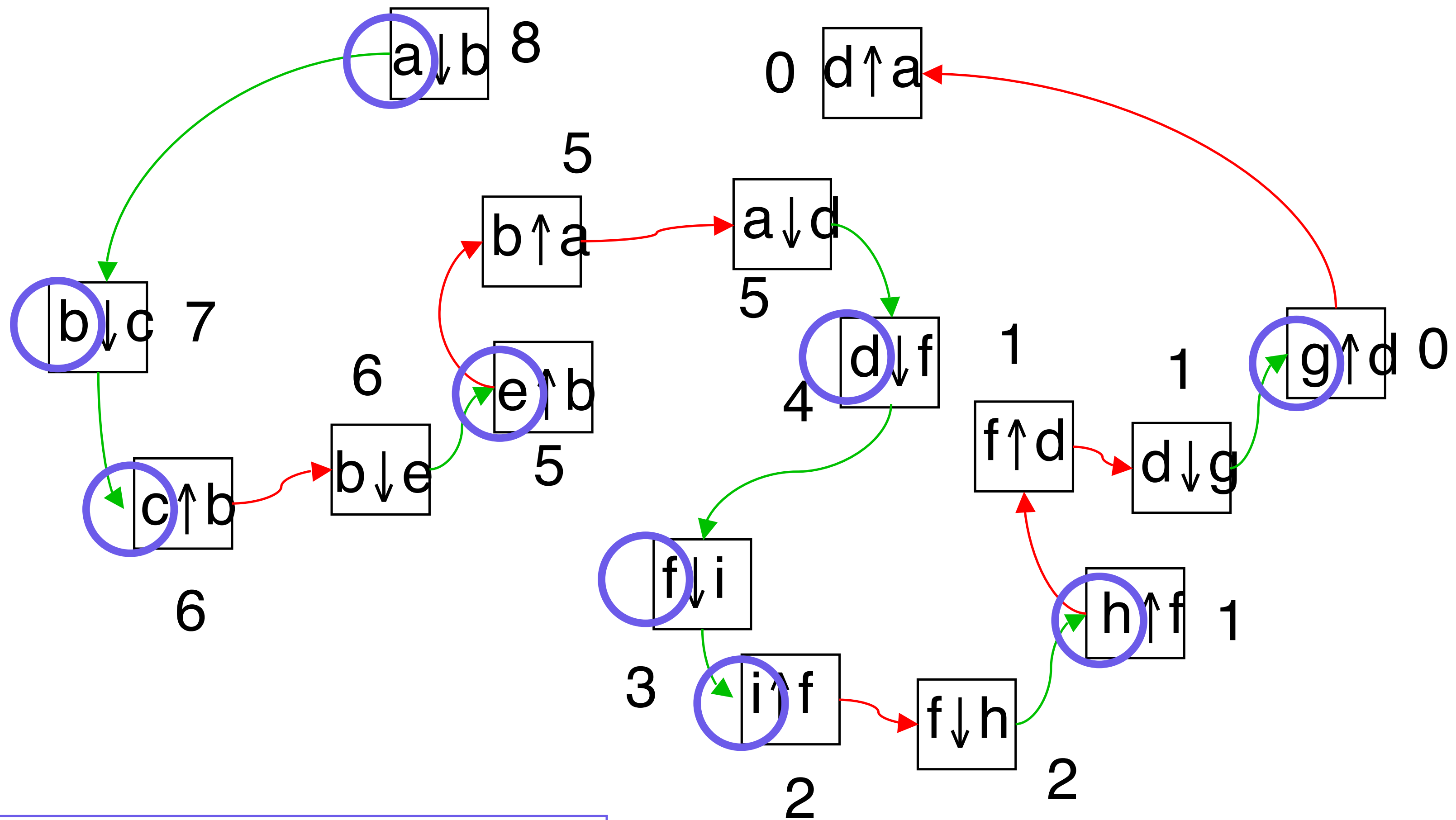
Try it



Use a list-ranking variation
Count *downward* nodes only



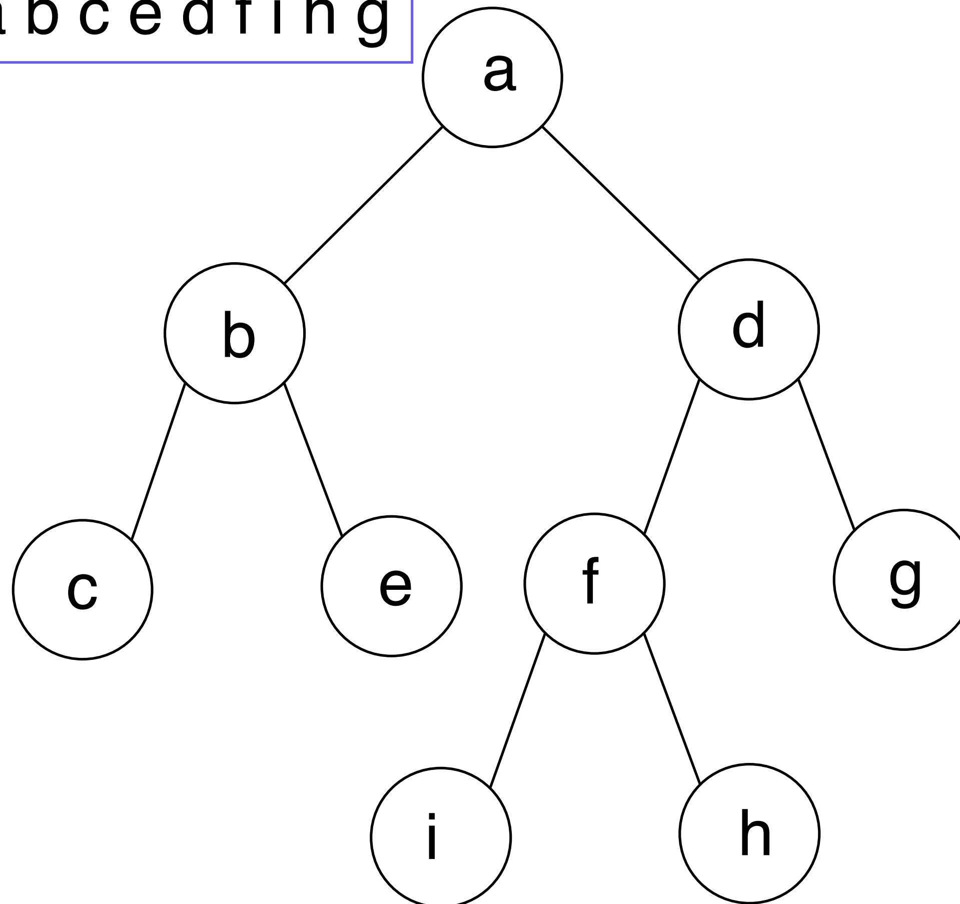
Pre-order of original is **reverse** of this order using first component of the root and the *targets* of the ↓ nodes only



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

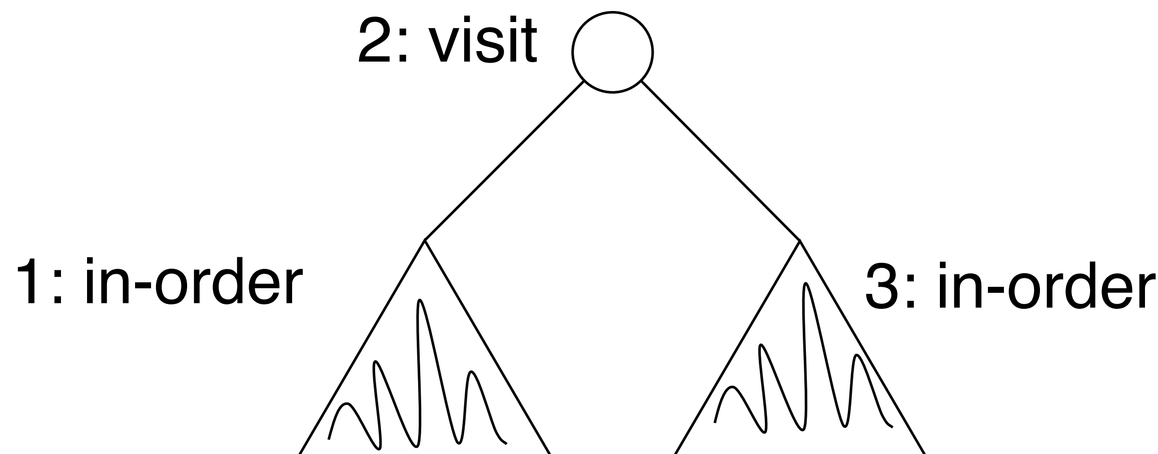
Check with Original

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

3	6	0	4	1	7	2	5

First pivot

3

3	6	0	4	1	7	2	5

	>	<	>	<	>	<	>

Comparisons with pivot (Each node remembers which half it is in.)

Quicksort Partition Tree

First pivot 3

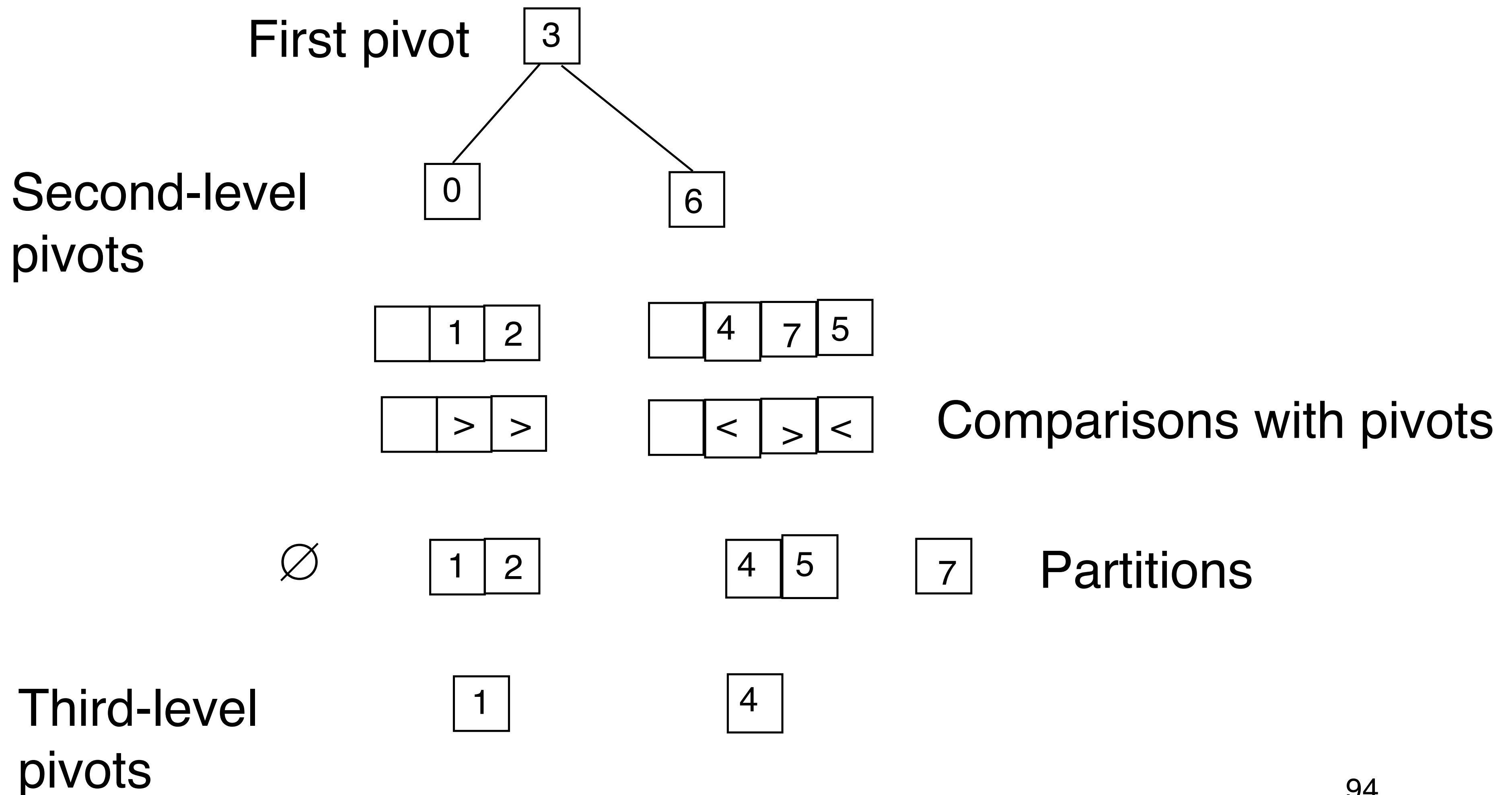
3 6 0 4 1 7 2 5

> < > < > < > Comparisons with pivot

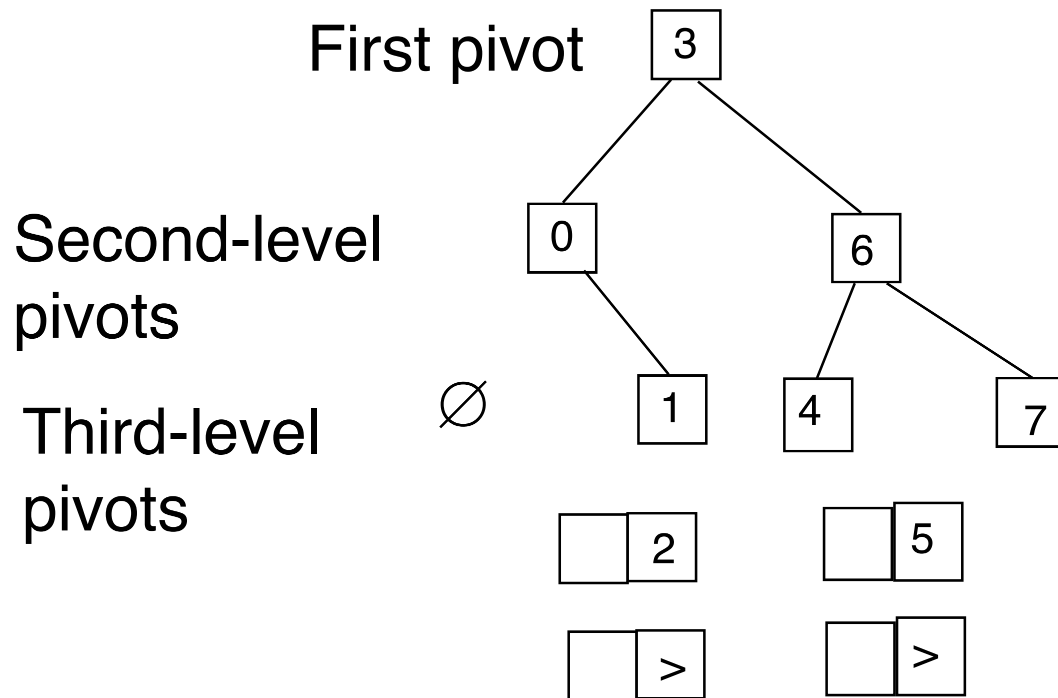
0 1 2 6 4 7 5 Partition

Second-level pivots 0 6

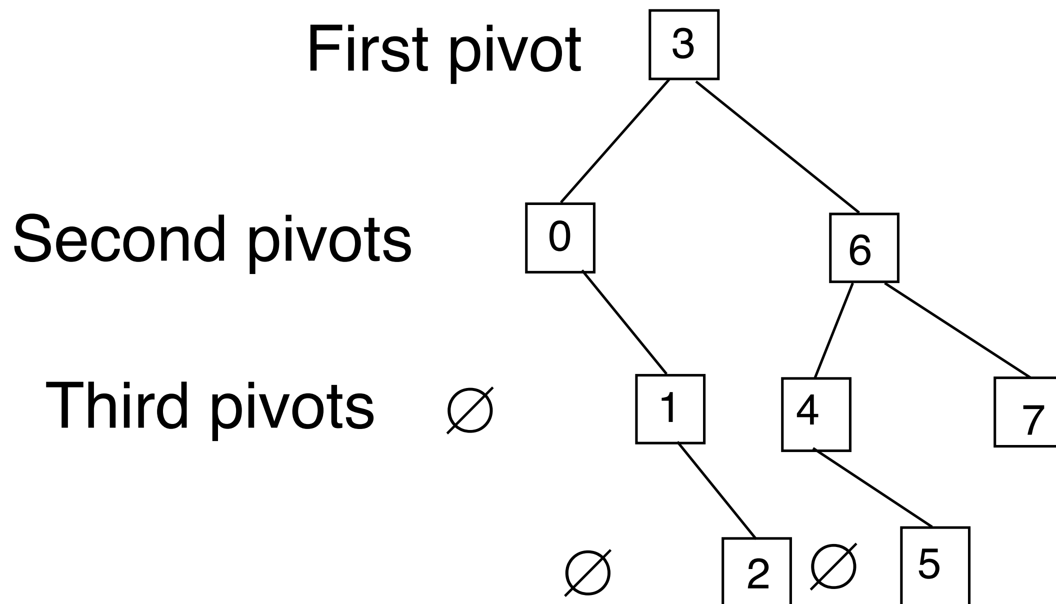
Quicksort Partition Tree



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).

“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?

Misc. Notes on PRAM

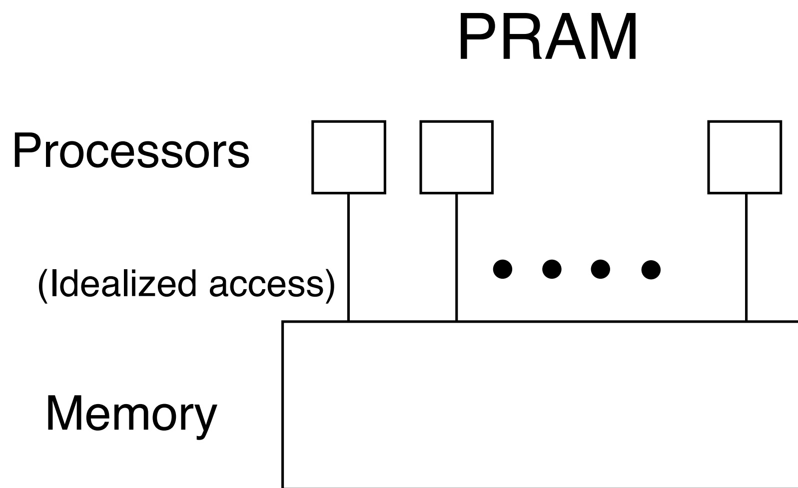
- Cole's sorting method based on merging: O(log n) on a CREW PRAM (c.f. Gibbons, A.M. & Rytter, W. (1988) Efficient Parallel Algorithms. Cambridge University Press.)

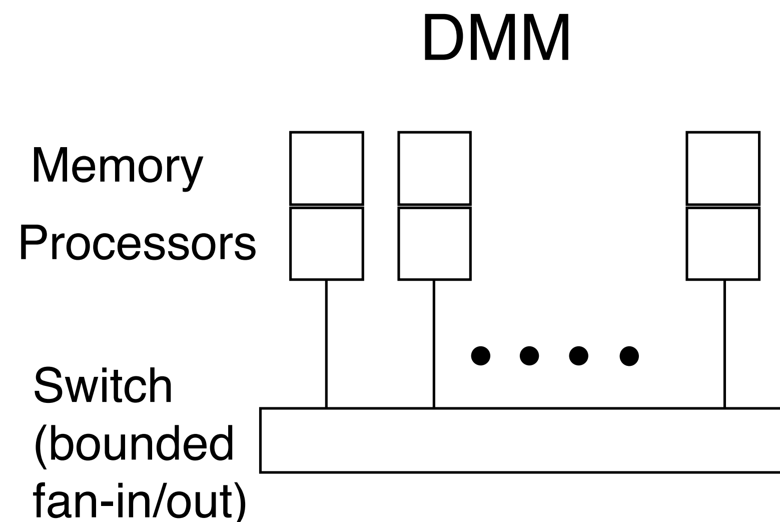
- Parallel recognition of a context-free language: $P(\log^2 n)$ time using n^6 processors.

- Many 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.

References

- Selim Akl, Parallel Computation: Models and Methods, Prentice Hall, 1997.
- Selim Akl, The Design of Efficient Parallel Algorithms, Chapter 2 in “Handbook on Parallel and Distributed Processing” edited by J. Blazewicz, K. Ecker, B. Plateau, and D. Trystram, Springer Verlag, 2000.
- Selim Akl, Design & Analysis of Parallel Algorithms, Prentice Hall, 1989.
- Cormen, Leisterson, and Rivest, Introduction to Algorithms, 1st edition (i.e., older), 1990, McGraw Hill and MIT Press, Chapter 30 on parallel algorithms.
- Phillip Gibbons, Asynchronous PRAM Algorithms, Ch 22 in Synthesis of Parallel Algorithms, edited by John Reif, Morgan Kaufmann Publishers, 1993.
- Joseph JaJa, An Introduction to Parallel Algorithms, Addison Wesley, 1992.
- Michael Quinn, Parallel Computing: Theory and Practice, McGraw Hill, 1994.