RAM and Parallel RAM (PRAM)

Why models?

- What is a machine model?
  - A abstraction describes the operation of a machine.
  - Allowing to associate a value (cost) to each machine operation.
- Why do we need models?
  - Make it easy to reason algorithms
  - Hide the machine implementation details so that general results that apply to a broad class of machines to be obtained.
  - Analyze the achievable complexity (time, space, etc) bounds
  - Analyze maximum parallelism
  - Models are directly related to algorithms.

RAM (random access machine) model

- Memory consists of infinite array (memory cells).
- Each memory cell holds an infinitely large number.
- Instructions execute sequentially one at a time.
- All instructions take unit time
  - Load/store
  - Arithmetic
  - Logic

- Running time of an algorithm is the number of instructions executed.
- Memory requirement is the number of memory cells used in the algorithm.

RAM (random access machine) model

- The RAM model is the base of algorithm analysis for sequential algorithms although it is not perfect.
  - Memory not infinite
  - Not all memory access take the same time
  - Not all arithmetic operations take the same time
  - Instruction pipelining is not taken into consideration
- The RAM model (with asymptotic analysis) often gives relatively realistic results.

PRAM (Parallel RAM)

- A model developed for parallel machines
  - An unbounded collection of processors
  - Each processor has an infinite number of registers
  - An unbounded collection of shared memory cells.
  - All processors can access all memory cells in unit time (when there is no memory conflict).
  - All processors execute PRAM instructions synchronously
    - Somewhat like SIMD, except that different processors can run different instructions in the lock step.
    - Some processors may idle.

PRAM (Parallel RAM)

- A model developed for parallel machines
  - Each PRAM instruction executes in 3-phase cycles
    - Read from a share memory cell (if needed)
    - Computation
    - Write to a share memory cell (if needed)
    - Example: for all I, do A[i] = A[i-1]+1;
      - Read A[i-1], compute add 1, write A[i]

- The only way processors exchange data is through the shared memory.

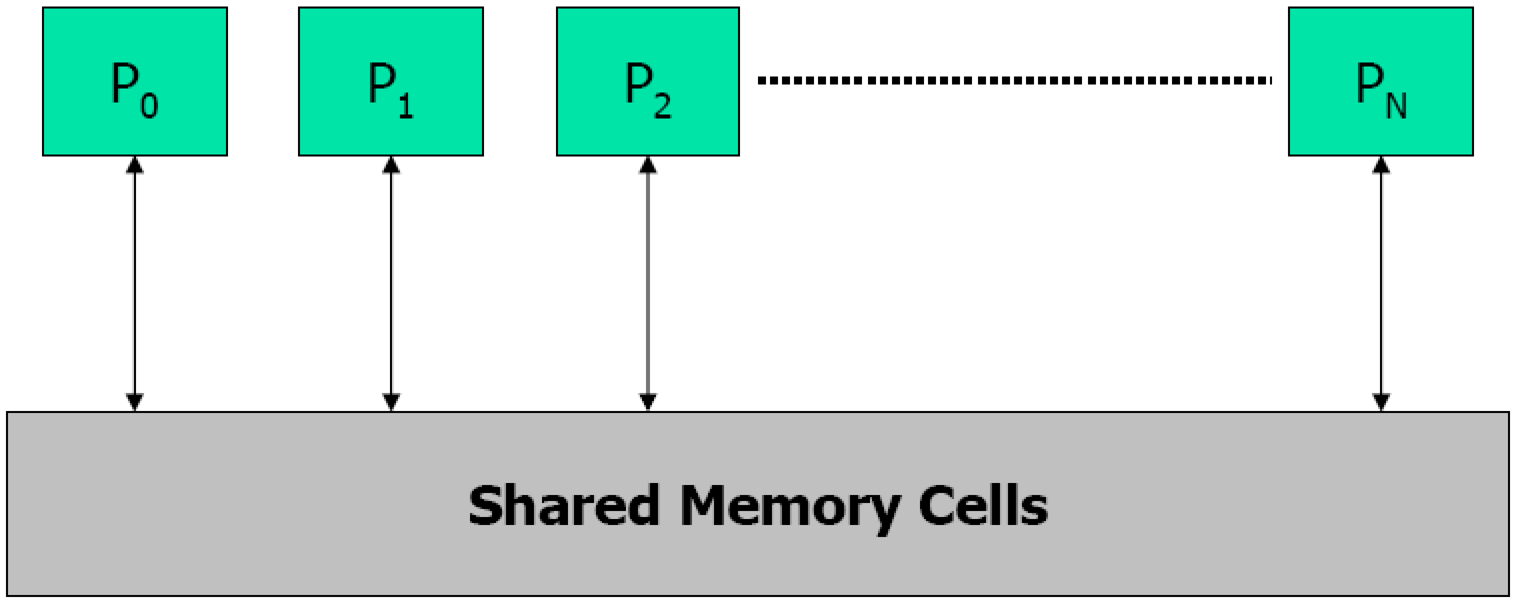
PRAM (Parallel RAM)

Parallel time complexity: the number of synchronous steps in the algorithm

Space complexity: the number of share memory

Parallelism: the number of processors used

PRAM



All processors can do things in a synchronous manner (with infinite shared Memory and infinite local memory), how many steps do it take to complete the task?

PRAM – further refinement

- PRAMs are further classifed based on how the memory conflicts are resolved.
  - Read
    - Exclusive Read (ER) – all processors can only simultaneously read from distinct memory location (but not the same location).
      - What if two processors want to read from the same location?
    - Concurrent Read (CR) – all processors can simultaneously read from all memory locations.

PRAM – further refinement

- PRAMs are further classified based on how the memory conflicts are resolved.
  - Write
    - Exclusive Write (EW) – all processors can only simultaneously write to distinct memory location (but not the same location).
    - Concurrent Write (CW) – all processors can simultaneously write to all memory locations.
      - Common CW: only allow same value to be written to the same location simultaneously.
      - Random CW: randomly pick a value
      - Priority CW: processors have priority, the value in the highest priority processor wins.

PRAM model variations

- EREW, CREW, CRCW (common), CRCW (random), CRCW (Priority)
  - Which model is closer to the practical SMP or multicore machines?
- Model A is computationally stronger than model B if and only if any algorithm written in B will run unchange in A.
  - EREW <= CREW <= CRCW (common) <= CRCW (random)

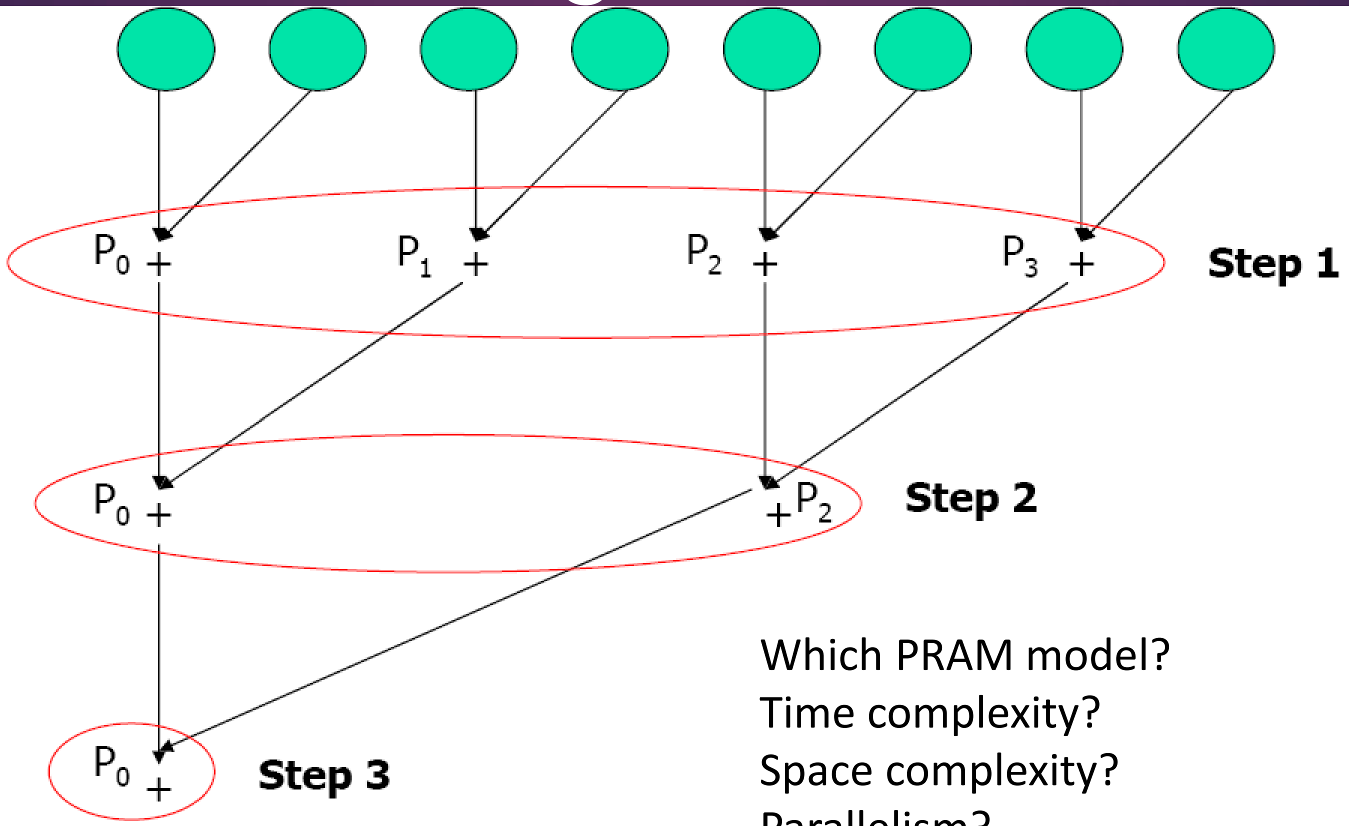
PRAM algorithm example

- SUM: Add N numbers in memory M[0, 1, …, N-1]
- Sequential SUM algorithm (O(N) complexity)

```
for (i=0; i<N; i++) sum = sum + M[i];
```

- PRAM SUM algorithm?

PRAM SUM algorithm



Which PRAM model?
Time complexity?
Space complexity?
Parallelism?
Speedup (.vs. sequential code)?

Parallel addition

- Time complexity: log(n) steps
- Parallelism: n/2 processors
- Speed-up (vs sequential algorithm): n/log(n)

Broadcast in PRAM

- EREW
  - double the number of processors that have the value in each steps
  - Log(P) steps
- CREW
  - Broadcaster sends value to shared memory
  - All processors read from shared memory
  - O(1) steps

Parallel search algorithm

- P processors PRAM with unsorted N numbers (P<=N)
- Does x exist in the N numbers?
- p_0 has x initially, p_0 must know the answer at the end.
- PRAM Algorithm:
  - Step 1: Inform everyone what x is
  - Step 2: every processor checks N/P numbers and sets a flag
  - Step 3: Check if any flag is set to 1.

Parallel search algorithm

- PRAM Algorithm:
  - Step 1: Inform everyone what x is
  - Step 2: every processor checks N/P numbers and sets a flag
  - Step 3: Check if any flag is set to 1.
- EREW: O(log(p)) step 1, O(N/P) step 2, and O(log(p)) step 3.
- CREW: O(1) step 1, O(N/P) step 2, and O(log(p)) step 3.
- CRCW (common): O(1) step 1, O(N/P) step 2, and O(1) step 3.

Find Max of N items

- CRCW algorithm with O(1) time using N^2 processors
  - Processor (r, 1) do A[s] = 1
  - Process (r,s) do if (X[r] < X[s]) A[r] = 0;
  - Process (r, 1) do: If A[r] = 1, max = X[r];

PRAM matrix-vector product

- Given an n x n matrix A and a column vector X = (x[0], x[1], …, x[n-1]), B = A X
- Sequential code:

```
For(i=0; i<n; i++) for (j=0; j<n; j++) B[i] += A[i][j] * X[j];
```

- CREW PRAM algorithm
  - Time to compute the product?
  - Time to compute the sum?
  - Number of processors needed?
  - Why CREW instead of EREW?

PRAM matrix multiplication

- CREW PRAM algorithm?
  - Time to compute the product?
  - Time to compute the sum?
  - Number of processors needed?

PRAM strengths

- Natural extension of RAM
- It is simple and easy to understand
  - Communication and synchronization issues are hided.
- Can be used as a benchmark
  - If an algorithm performs badly in the PRAM model, it will perform badly in reality.
  - A good PRAM program may not be practical though.
- It is useful to reason threaded algorithms for SMP/multicore machines.

PRAM weaknesses

- Model inaccuracies
  - Unbounded local memory (register)
  - All operations take unit time
  - Processors run in lock steps
- Unaccounted costs
  - Non-local memory access
  - Latency
  - Bandwidth
  - Memory access contention

PRAM variations

- Bounded memory PRAM, PRAM(m)
  - In a given step, only m memory accesses can be serviced.
  - Lemma: Assume *m'<m*. Any problem that can be solved on a *p*-processor and *m*-cell PRAM in *t* steps can be solved on a *max(p,m')*-processor *m'*-cell PRAM in *O(tm/m')* steps.
- Bounded number of processors PRAM
  - Lemma: Any problem that can be solved by a p processor PRAM in t steps can be solved by a p' processor PRAM in t = O(tp/p') steps.
    - E.g. Matrix multiplication PRAM algorithm with time complexity O(log(N)) on N^3 processors→ on P processors, the problem can be solved in O(log(N)N^3/P).
- LPRAM
  - L units to access global memory
  - Lemma: Any algorithm that runs in a p processor PRAM can run in LPRAM with a loss of a factor of L.

PRAM summary

- The RAM model is widely used.
- PRAM is simple and easy to understand
  - This model never reaches beyond the algorithm community.
  - It is getting more important as threaded programming becomes more popular.
- The BSP (bulk synchronous parallel) model is another try after PRAM.
  - Asynchronously progress
  - Model latency and limited bandwidth

Question

- Why is PRAM attractive and important model for designers of parallel algorithms ?
  - It is **natural**: the number of operations executed per one cycle on p processors is at most p
  - It is **strong**: any processor can read/write any shared memory cell in unit time
  - It is **simple**: it abstracts from any communication or synchronization overhead, which makes the complexity and correctness of PRAM algorithm easier
  - It can be used as a **benchmark**: If a problem has no feasible/efficient solution on PRAM, it has no feasible/efficient solution for any parallel machine

Concurrent vs. Exclusive Access

Four models

- EREW: exclusive read and exclusive write
- CREW: concurrent read and exclusive write
- ERCW: exclusive read and concurrent write
- CRCW: concurrent read and concurrent write

- Handling write conflicts
  - Common-write model: only if they write the same value.
  - Arbitrary-write model: an arbitrary one succeeds.
  - Priority-write model: the one with smallest index succeeds.
- EREW and CRCW are most popular.

Synchronization and Control

- Synchronization:
  - A most important and complicated issue
  - Suppose all processors are inherently tightly synchronized:
    - All processors execute the same statements at the same time
    - No race among processors, i.e, same pace.
- Termination control of a parallel loop:
  - Depend on the state of all processors
  - Can be tested in $O(1)$ time.

Pointer Jumping – list ranking

- Given a single linked list L with n objects, compute, for each object in L, its distance from the end of the list.
- Formally: suppose next is the pointer field

$$d[i] = \begin{cases} 0 & \text{if } next[i]=nil \\ d[next[i]]+1 & \text{if } next[i] \neq nil \end{cases}$$

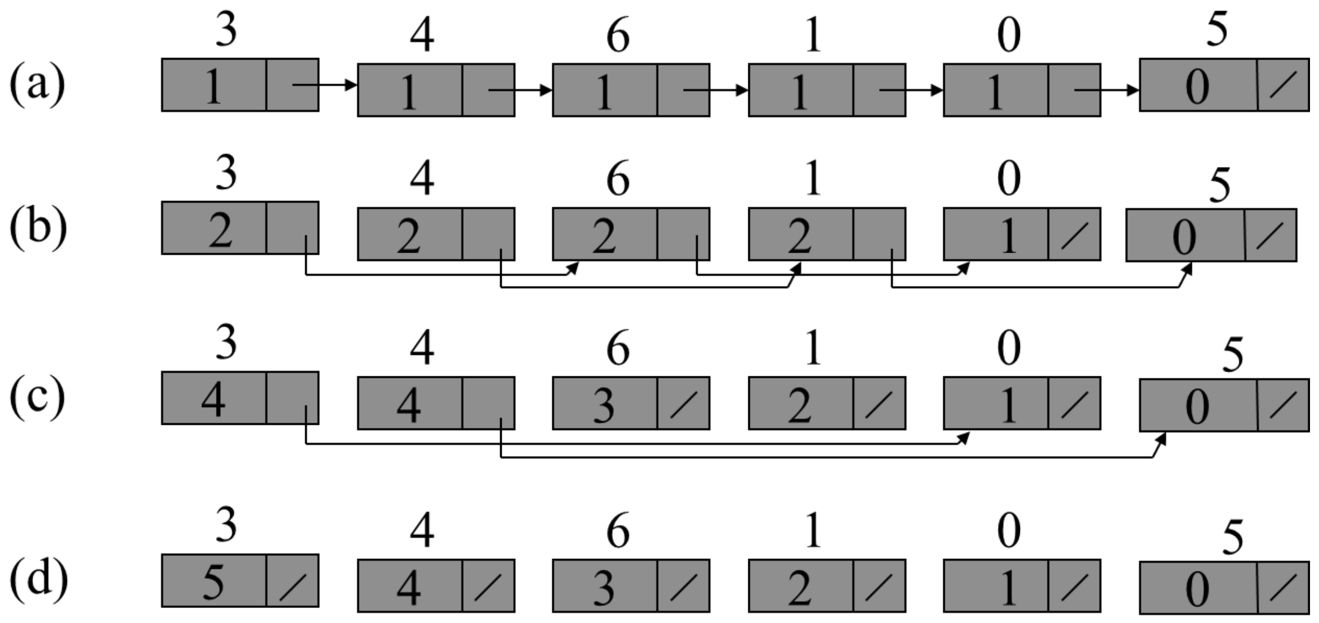
- Serial algorithm: $\Theta(n)$.

List Rank – EREW Algorithm

LIST-RANK(L)   (in $O(\log n)$ time)

```
1.  for each processor $i$, in parallel
2.      do if $next[i]=nil$
3.          then $d[i] \leftarrow 0$
4.          else $d[i] \leftarrow 1$
5.  while there exists an object $i$ such that $next[i] \neq nil$
6.      do for each processor $i$, in parallel
7.          do if $next[i] \neq nil$
8.              then $d[i] \leftarrow d[i] + d[next[i]]$
9.                  $next[i] \leftarrow next[next[i]]$
```

List Rank – EREW Algorithm



List ranking –correctness of EREW algorithm

- **Loop invariant:** for each i, the sum of d values in the sub-list headed by i is the correct distance from i to the end of the original list L.
- **Parallel memory must be synchronized:** the reads on the right must occur before the writes on the left. Moreover, read d[i] and then read d[next[i]].
- **An EREW algorithm:** every read and write is exclusive. For an object i, its processor reads d[i], and then its precedent processor reads its d[i]. Writes are all in distinct locations.

LIST ranking EREW algorithm running time

- $O(\log n)$:
  - The initialization for loop runs in $O(1)$.
  - Each iteration of while loop runs in $O(1)$.
  - There are exactly $\lceil \log n \rceil$ iterations:
    - Each iteration transforms each list into two interleaved lists: one consisting of objects in even positions, and the other odd positions. Thus, each iteration double the number of lists but halves their lengths.
  - The termination test in line 5 runs in $O(1)$.
  - Define *work* = #processors × running time. $O(n \log n)$.

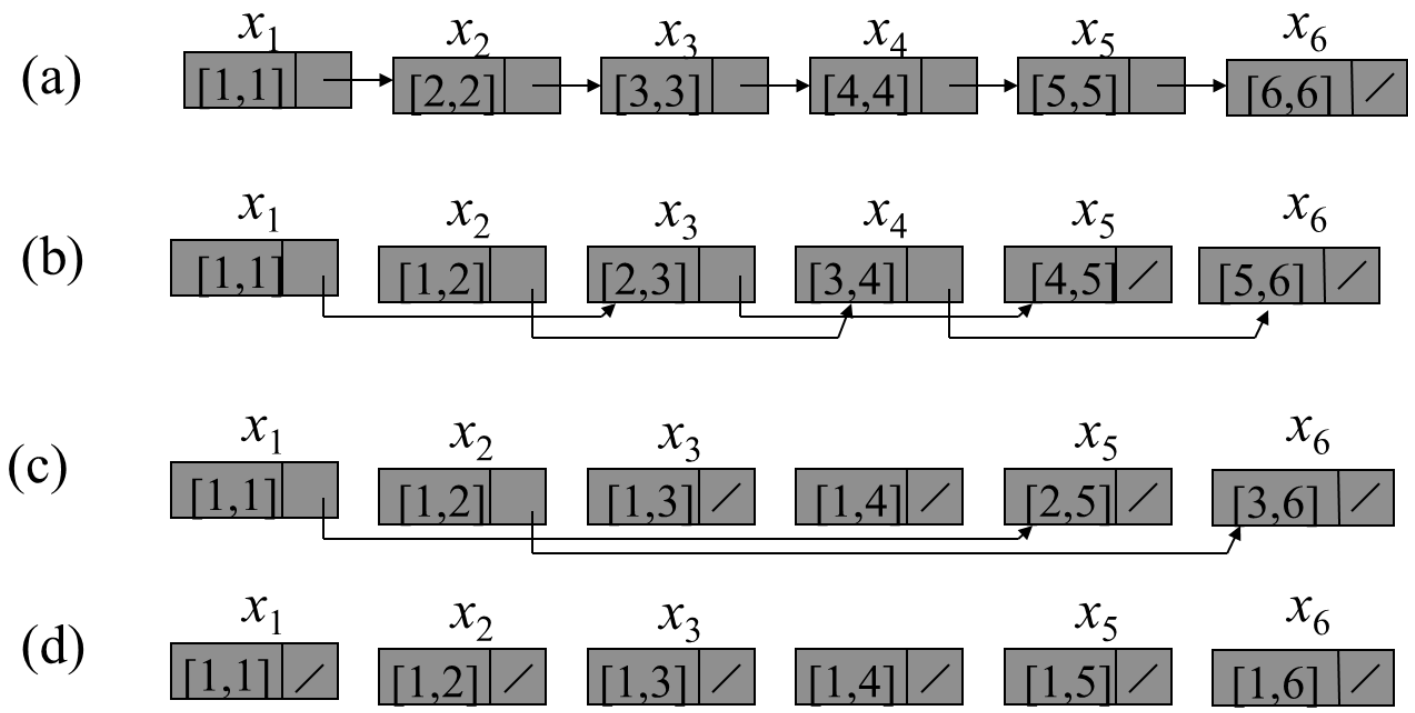
Parallel prefix on a list

- A prefix computation is defined as:
  - Input: $<x_1, x_2, \ldots, x_n>$
  - Binary associative operation $\otimes$
  - Output: $<y_1, y_2, \ldots, y_n>$
  - Such that:
    - $y_1 = x_1$
    - $y_k = y_{k-1} \otimes x_k$ for $k=2,3, \ldots, n$, i.e, $y_k = \otimes x_1 \otimes x_2 \ldots \otimes x_k$.
  - Suppose $<x_1, x_2, \ldots, x_n>$ are stored orderly in a list.
  - Define notation: $[i,j] = x_i \otimes x_{i+1} \ldots \otimes x_j$

Prefix computation

```
LIST-PREFIX(L)
1.  for each processor $i$, in parallel
2.      do $y[i] \leftarrow x[i]$
3.  while there exists an object $i$ such that $next[i] \neq nil$
4.      do for each processor $i$, in parallel
5.          do if $next[i] \neq nil$
6.              then $y[next[i]] \leftarrow y[i] \otimes y[next[i]]$
7.                  $next[i] \leftarrow next[next[i]]$
```

Prefix computation –EREW algorithm



Find root –CREW algorithm

- Suppose a forest of binary trees, each node i has a pointer parent[i].
- Find the identity of the tree of each node.
- Assume that each node is associated a processor.
- Assume that each node i has a field root[i].

CREW algorithm

```
FIND-ROOTS(F)
1.  for each processor i, in parallel
2.    do if parent[i] = nil
3.        then root[i] ← i
4.  while there exist a node i such that parent[i] ≠ nil
5.      do for each processor i, in parallel
6.          do if parent[i] ≠ nil
7.              then root[i] ← root[parent[i]]
8.                   parent[i] ← parent[parent[i]]
```

CREW algorithm

- Running time: O(log d), where d is the height of maximum-depth tree in the forest.
- All the writes are exclusive
- But the read in line 7 is concurrent, since several nodes may have same node as parent.

CREW v/s EREW

Q: How fast can *n* nodes in a forest determine their roots using only exclusive read?

A: $\Omega(\log n)$

Argument: when exclusive read, a given peace of information can only be copied to one other memory location in each step, thus the number of locations containing a given piece of information at most doubles at each step. Looking at a forest with one tree of n nodes, the root identity is stored in one place initially. After the first step, it is stored in at most two places; after the second step, it is stored in at most four places, …, so need lg n steps for it to be stored at n places.

Find maximum – CRCW algorithm

FAST-MAX(A)

```
1.  n ← length[A]
2.  for i ← 0 to n-1, in parallel
3.      do m[i] ← true
4.  for i ← 0 to n-1 and j ← 0 to n-1, in parallel
5.      do if A[i] < A[j]
6.          then m[i] ← false
7.  for i ← 0 to n-1, in parallel
8.      do if m[i] = true
9.          then max ← A[i]
10. return max
```

The running time is $O(1)$.

$$A[j]$$

$$A[i] \quad \begin{array}{c|ccccc|c} & 5 & 6 & 9 & 2 & 9 & m \\ \hline 5 & F & T & T & F & T & F \\ 6 & F & F & T & F & T & F \\ 9 & F & F & F & F & F & T \\ 2 & T & T & T & F & T & F \\ 9 & F & F & F & F & F & T \\ \hline \end{array}$$

max=9

CRCW v/s EREW

- If find maximum using EREW, then $\Omega(\lg n)$.
- Argument: consider how many elements "think" that they might be the maximum.
  - First, n,
  - After first step, n/2,
  - After second step n/4. …, each step, halve.
- Moreover, CREW takes $\Omega(\log n)$.

CRCW v/s EREW

CRCW:

- Some say : easier to program and more faster.
- Others say: The hardware to CRCW is slower than EREW. And one can not find maximum in $O(1)$.
- Still others say: either EREW or CRCW is wrong. Processors must be connected by a network, and only be able to communicate with other via the network, so network should be part of the model.

CRCW algorithms can solve some problems quickly than can EREW algorithm

- The problem of finding MAX element can be solved in O(1) time using CRCW algorithm with n^2 processors
- EREW algorithm for this problem takes Ω(log n) time and that no CREW algorithm does any better. Why?

Any EREW algorithm can be executed on a CRCW PRAM

- Thus, the CRCW model is strictly more powerful than the EREW model.
- But how much more powerful is it?
- Now we provide a theoretical bound on the power of a CRCW PRAM over an EREW PRAM

THEOREM

Theorem. A *p*-processor CRCW algorithm can be no more than O(*log p*) time faster than the best *p*-processor EREW algorithm for the same problem.

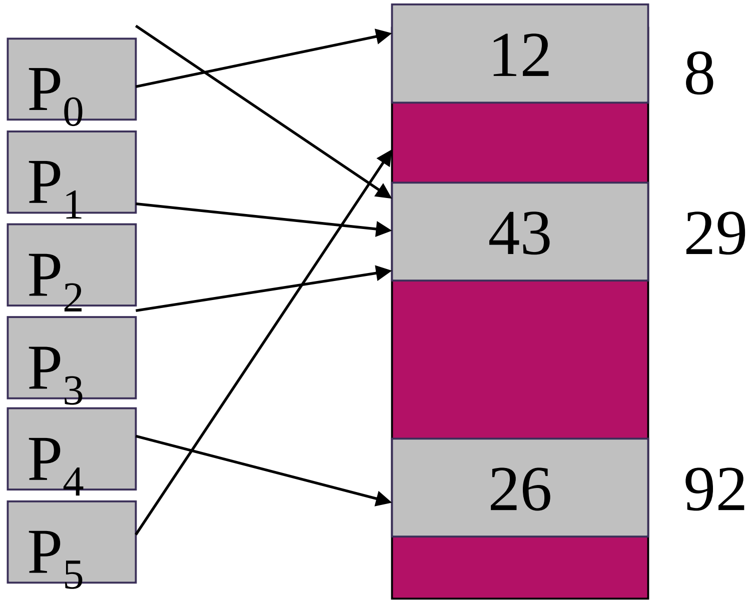
Proof.

The proof is a simulation argument. We simulate each step of the CRCW algorithm with an O(log p)-time EREW computation.

Because the processing power of both machines is the same, we need only focus on memory accessing.

Let’s present the proof for simulating concurrent writes here. Implementation of concurrent reading is left as an exercise.

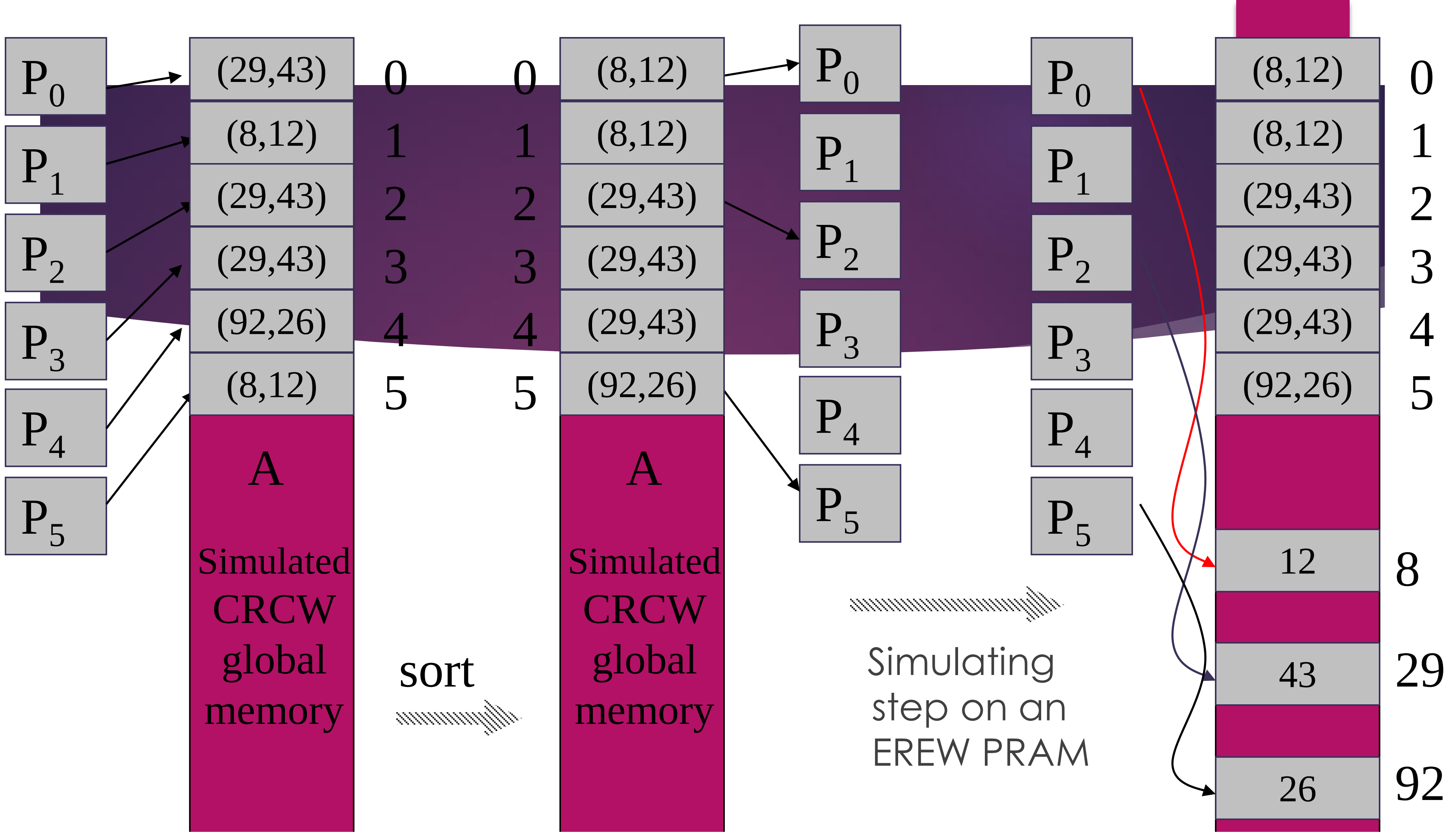
The p processors in the EREW PRAM simulate a concurrent write of the CRCW algorithm using an auxiliary array A of length p.



1.When CRCW processor P_i, for $i=0,1,\ldots,p-1$, desires to write a datum x_i to location l_i, each corresponding EREW processor P_i instead writes the ordered pair (l_i,x_i) to location A[*i*].

2. This writes are exclusive, since each processor writes to a distinct memory location.

3. Then, the array A is sorted by the first coordinate of the ordered pairs in O(log p) time, which causes all data written to the same location to be brought together in the output



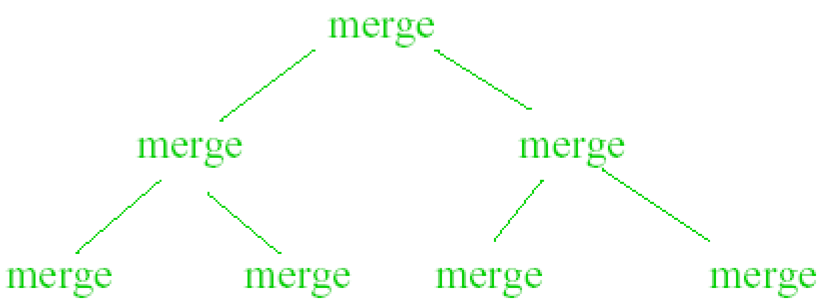
4. Each EREW processor P_i now inspects A[i]=(l_j,x_j) and A[i-1]=(l_k,x_k), where j and k are values in the range $0 \le j,k \le p-1$. If $l_j \neq l_k$ *or* $i=0$ then P_i writes the datum x_j to location l_j in the global memory. Otherwise, the processor does nothing.

End of the proof

- Since the array A is sorted by first coordinate, only one of the processors writing to any given location actually succeeds, and thus the write is exclusive.
- This process thus implements each step of concurrent writing in the common CRCW model in O(log p) time

Optimal sorting in log(n) steps
Cole's algorithm

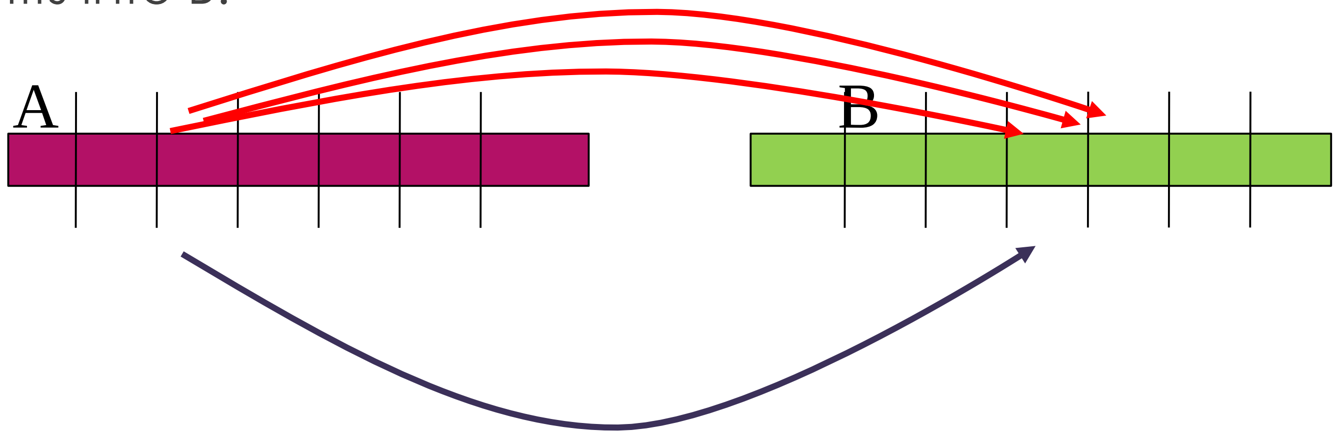
- Suppose we know how to merge two increasing sequences in *log(log(n))* steps
- Then we can climb up the merging tree and spend only *log(log(n))* per level, thus getting a parallel sorting technique in *log(n) log(log(n))*



- Merges at the same level are performing in parallel

How to merge in log(log(n)) time with n processors

- Let A and B are to sorted sequences of size n
- Divide A,B into $\sqrt{n}$ blocks of length $\sqrt{n}$
- Compare first elements of each block in A with first elements of each block in B
- Then compare first elements of each block in A with each element in a “suitable” block of B
- At this point we know where all first elements of each block in A fits into B.



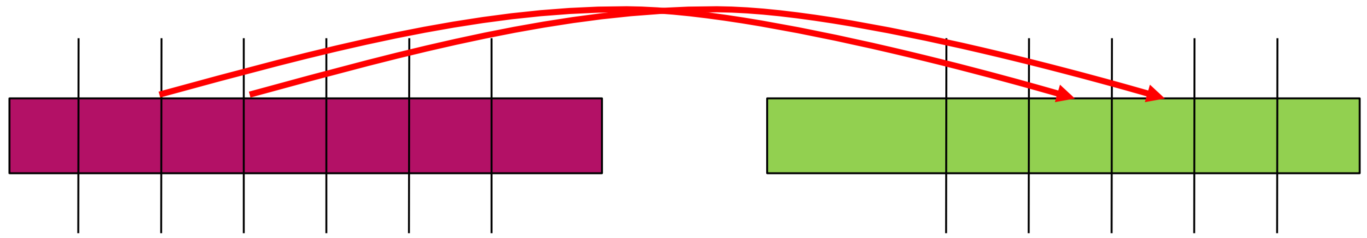
How to merge in log(log(n)) time with n processors

- Thus the problem has been reduced to a set of disjoint problems each of which involves merging of block of   elements of A with some consecutive piece of B.
- Recursively we solve these problems
- The parallel time t(n) satisfies to

$\sqrt{n}$

t(n)≤2+ t(   ) implying t(n)=O(log(log(n)))

$\sqrt{n}$



The issue arises, therefore, of which model is preferable – CRCW or EREW

- Advocates of the CRCW models point out that they are easier to program than EREW model and that their algorithms run faster
- Critics contend that hardware to implement concurrent memory operations is slower than hardware to exclusive memory operations, and thus the faster running time of CRCW algorithm is fictitious.
  - In reality, they say, one cannot find the maximum of n values in O(1) time
- Others say that PRAM is the wrong model entirely. Processors must be interconnected by a communication network, and the communication network should be part of the model

It is quite clear that the issue of the "right" parallel model is not going to be easily settled in favour of any one model. The important think to realize, however, is that these models are just that: models!