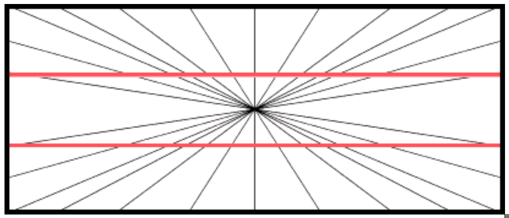
PRAM Algorithms



"Computer science is no more about computers than astronomy is about telescopes."

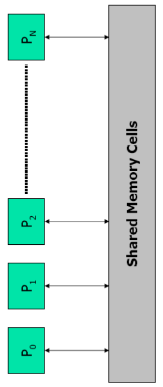
Edsger Dijkstra

(11/05/1930-6/9/2002)

Slides adapted from R. Libeskind-Hadas, I. Potapov

One more time about PRAM model

- N synchronized processors
- Shared memory
  - EREW, ERCW,
  - CREW, CRCW
- Constant time
  - access to the memory
  - standard multiplication/addition
  - Communication
    (implemented via access to shared memory)



Metrics

A measure of relative performance between a multiprocessor system and a single processor system is the *speed-up* $S(p)$, defined as follows:

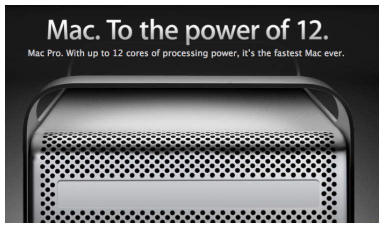
$$S(p) = \frac{\text{Execution time using a single processor system}}{\text{Execution time using a multiprocessor with } p \text{ processors}}$$

$$S(p) = \frac{T_1}{T_p} \qquad Efficiency = \frac{S_p}{p}$$

$$Work = p \times T_p$$

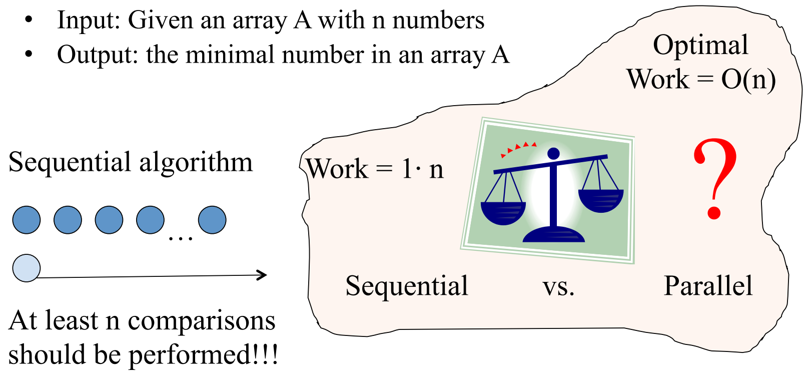
Parallelism at all levels

- Parallel circuits
- Parallel computers
- Distributed systems


Mac. To the power of 12.
Mac Pro. With up to 12 cores of processing power, it's the fastest Mac ever.

Min of n numbers

- Input: Given an array A with n numbers
- Output: the minimal number in an array A



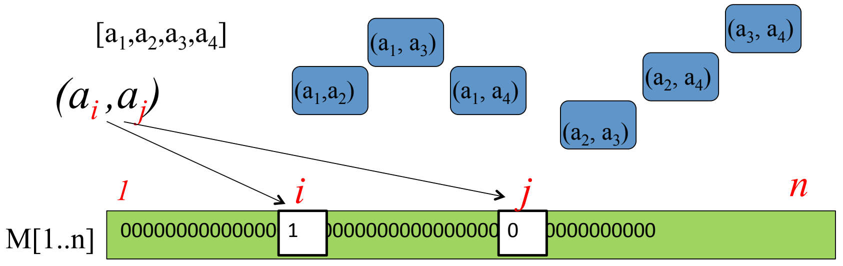
Work = (num. of processors) · (time)

Mission: Impossible …
computing in a constant time

- Archimedes: *Give me a lever long enough and a place to stand and I will move the earth*
- NOWDAYS….
  Give me a parallel machine with enough processors and I will find the smallest number in any giant set in a constant time!

Parallel solution 1
Min of n numbers

- Comparisons between numbers can be done independently
- The second part is to find the result using concurrent write mode
- For n numbers ----> we have ~ n^2 pairs



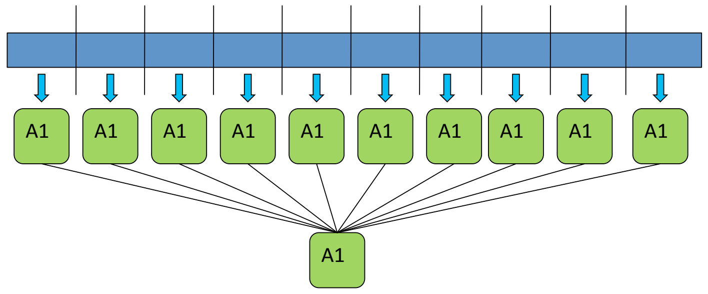
If $a_i > a_j$ then a_i cannot be the minimal number

The following program computes MIN of n numbers stored in the array C[1..n] in O(1) time with n^2 processors.

<u>Algorithm A1</u>

```
for each $1 \le i \le n$ do in parallel
    $M[i] := 0$
for each $1 \le i,j \le n$ do in parallel
    if $i \ne j$ $C[i] \le C[j]$ then $M[j] := 1$
for each $1 \le i \le n$ do in parallel
    if $M[i] = 0$ then output:=i
```

From n^2 processors to $n^{1+1/2}$

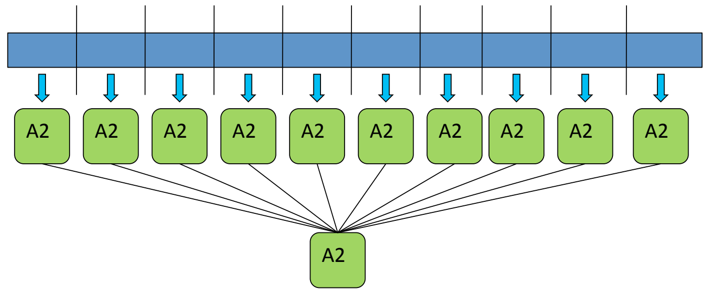


Step 1: Partition into disjoint blocks of size $\sqrt{n}$

Step 2: Apply A1 to each block $n\sqrt{n}$

Step 3: Apply A1 to the results from the step 2 $\sqrt{n}$

From $n^{1+1/2}$ processors to $n^{1+1/4}$



Step 1: Partition into disjoint blocks of size $\sqrt{n}$

Step 2: Apply A2 to each block

Step 3: Apply A2 to the results from the step 2

$n^2 \to n^{1+1/2} \to n^{1+1/4} \to n^{1+1/8} \to n^{1+1/16} \to \ldots \to n^{1+1/k} \sim n^1$

- Assume that we have an algorithm A_k working in O(1) time with $n^{1+\varepsilon_k}$ processors

Algorithm A_{k+1}

1. Let $\alpha=1/2$
2. Partition the input array C of size n into disjoint blocks of size n^α each
3. Apply in parallel algorithm A_k to each of these blocks
4. Apply algorithm A_k to the array C' consisting of n/n^α minima in the blocks.

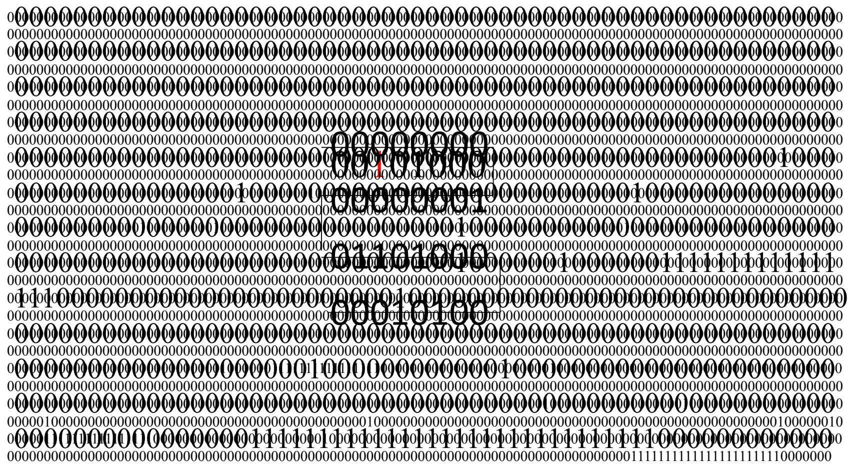
Complexity

- We can compute minimum of n numbers using CRCW PRAM model in O(log log n) with n processors by applying a strategy of partitioning the input

$$\text{Work} = n \cdot \log \log n$$

Mission: Impossible (Part 2)

Computing a position of the first one in the sequence of 0's and 1's in a constant time.



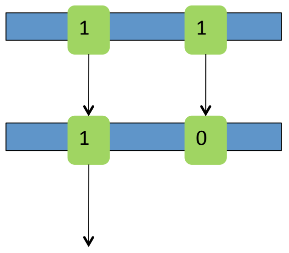
Worksheet:

- Write a parallel algorithm for finding the first 1 in a sequence of *n* 0's & 1's in *constant* time.
- How many processors do you need?

Problem 2.
Computing a position of the first one in the sequence of 0's and 1's.

Algorithm A
(2 parallel steps and n^2 processors)

```
for each 1≤i<j ≤n do in parallel
   if C[i] =1 and C[j]=1 then C[j]:=0
for each 1≤ i ≤n do in parallel
   if C[i] =1 then FIRST-ONE-POSITION:=i
```



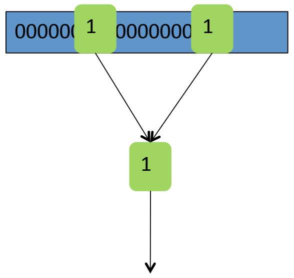
FIRST-ONE-POSITION(C)=4
for the input array
C=[0,0,0,1,0,0,0,1,1,1,0,0,0,1]

After the first parallel step C will contain a single element 1

Reducing number of processors

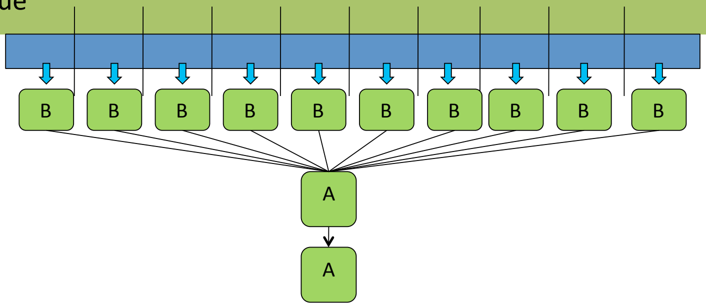
Algorithm B –
it reports if there is any one in the table.

```
There-is-one:=0
for each 1≤ i ≤n do in parallel
   if C[i] =1 then There-is-one:=1
```



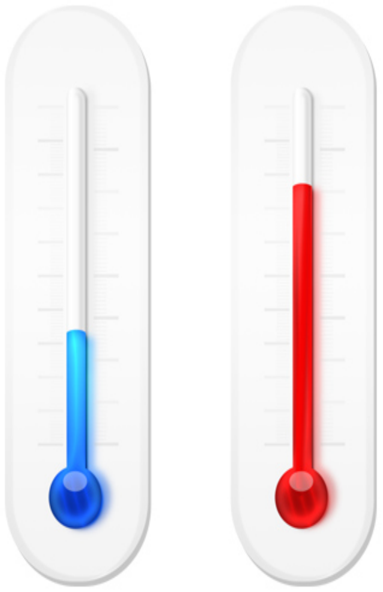
Now we can merge two algorithms A and B

1. Partition table C into segments of size $\sqrt{n}$
2. In each segment apply the algorithm B
3. Find position of the first one in these sequence by applying algorithm A
4. Apply algorithm A to this single segment and compute the final value



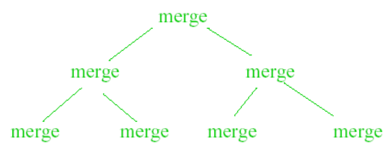
Complexity

- We apply an algorithm A twice and each time to the array of length $\sqrt{n}$ which need only $(\sqrt{n})^2 = n$ processors
- The time is O(1) and number of processors is n.



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

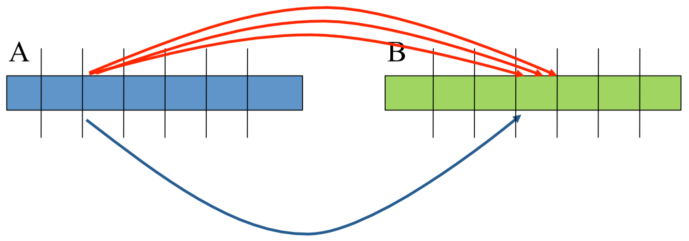
- Suppose we know how to merge two increasing sequences in *log(log(n))* steps
- Then we can climb up the merging tree and spend only *log(log(n))* per level, thus getting a parallel sorting technique in *log(n) log(log(n))*

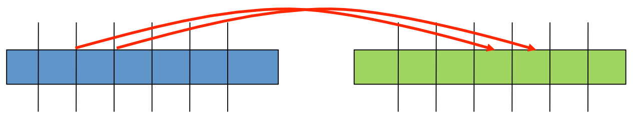


- Merges at the same level are performing in parallel

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

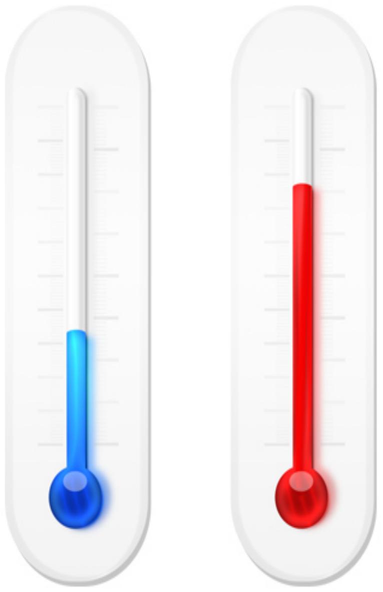
- Let A and B are to sorted sequences of size n
- Divide A,B into $\sqrt{n}$ blocks of length $\sqrt{n}$
- Compare first elements of each block in A with first elements of each block in B
- Then compare first elements of each block in A with each element in a "suitable" block of B
- At this point we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





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

$t(n) \leq 2 + t(\sqrt{n})$ implying $t(n)=O(\log(\log(n)))$



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 $\Omega(\log n)$ 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. 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.

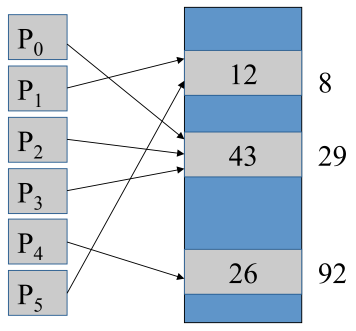
Proof.

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.

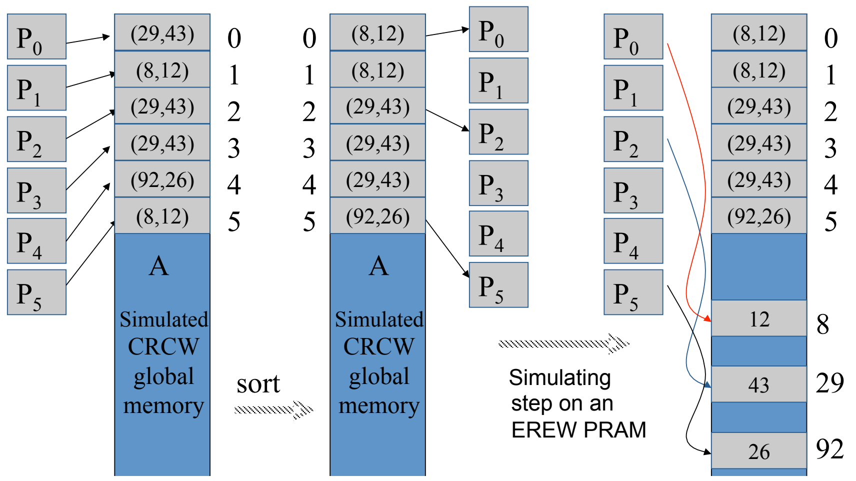
- The p processors in the EREW PRAM simulate a concurrent write of the CRCW algorithm using an auxiliary array A of length p.



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. These 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≤j,k≤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.

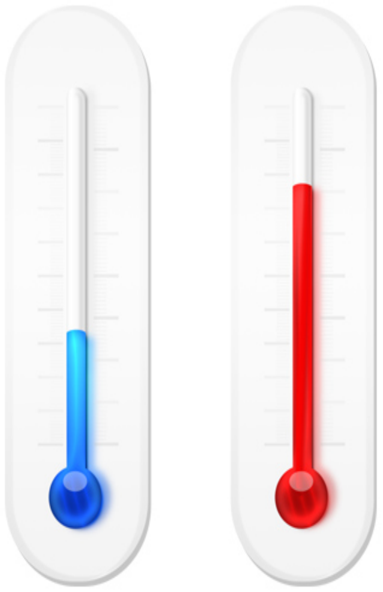
End of the proof

- 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

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

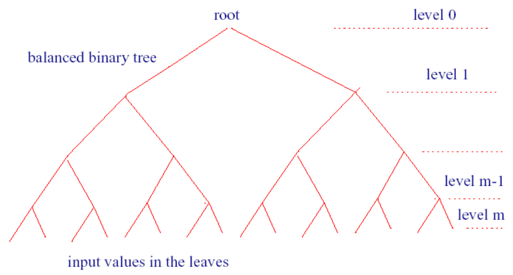
It is quite clear that the issue of the "right" parallel model is not going to be easily settled in favor of any one model. The important think to realize, however, is that these models are just that: models!



Basic techniques for PRAM

- Balanced binary tree technique
- Parallel divide and conquer
- Pointer Jumping (e.g., doubling)

Balanced binary tree technique



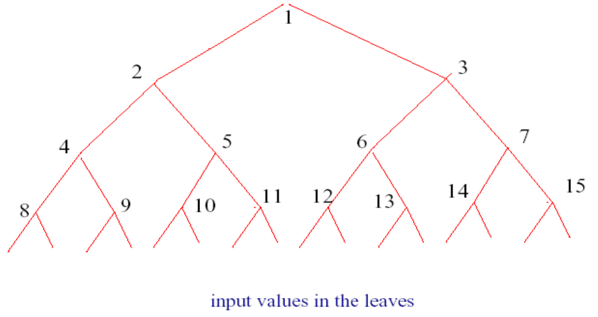
Structure of the algorithm

```
for level $i = m-1, m-2, \ldots, 0$ do
  for each vertex $v$ at level $i$ do in parallel
     value[v]:=value[LeftChild(v)] + value[RightChild(v)]
output:=value[root]
```

A possible way to store vertices in an array

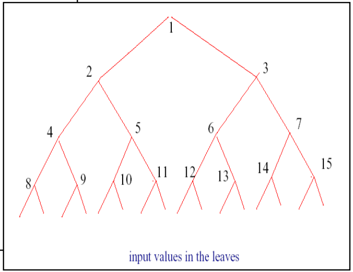
Input values stored in the array A[n … 2n-1]

LeftChild(i) = 2i; RightChild(i)=2i+1

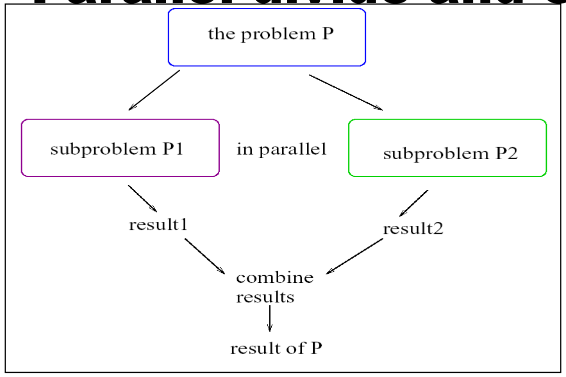


Sum

```
Initialize:
   for each $1 \le i \le 2n-1$ do in parallel
     if $i < n$ then T[i] :=0 else T[i]:=A[i-n+1]
   (* i-th input value stored in the i-th leaf*)

for level:=m-1 downto 0 do
   for all $2^{level} \le i < 2^{level+1}$ do in parallel
     T[i] := T[2i] + T[2i+1]
output := T[1]
```



Parallel divide and conquer

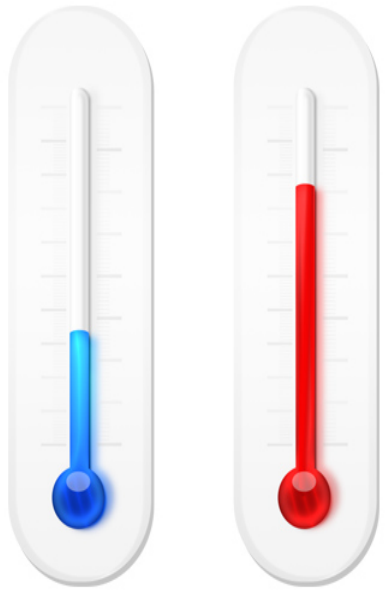


What about DP? Memoization?

for example result1 = sum of elements A[1 .. n/2]
result2 = sum of elements A[n/2+1 .. n]

result = result1 + result2

recursive parallel computation of the sum of n elements



Pointer Jumping

- This technique is normally applied to an array or to a list of elements
- The computation proceeds by recursive application of the calculation in hand to all elements over a certain distance (in the data structure) from each individual element
- This distance doubles in successive steps.
- Thus after k stages the computation has performed (for each element) over all elements within a distance of 2^k.

List ranking problem

- We introduce an O(log n) time algorithm that computes the distance to the end of the list for each object in an n-object list.
- One solution to the list ranking problem is simply to propagate distances back from the end of the list.
- This takes $\Theta(n)$ time, since k-th object from the end must wait for the k-1 objects following it to determine their distances from the end before it can determine its own.
- This solution is essentially a serial algorithm.

The propose of the following computation is to rank the elements of the list

Algorithm: RANK LIST ELEMENTS

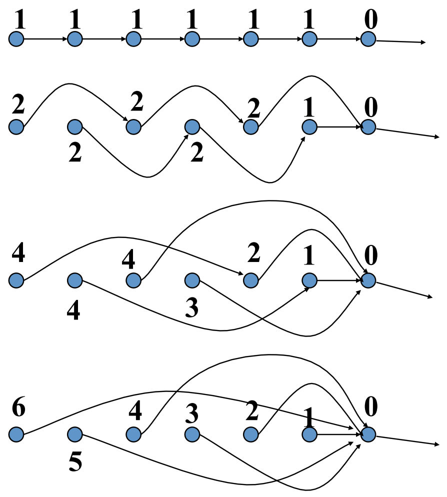
- Let L denote a list of n elements and let us associate processor with each element
- d(*i*) is the order number of *i* on the list
- We can take this to be the distance of element i from the end of the list
- The pointer for element i is next(i).

```
for each processor i do
    if next[i]=NIL then d[i]←0
    else d[i]←1
while exist i | next[i]≠NIL do
    for each processor i do
        if next[i] ≠ NIL then
            d[i]← d[i] + d[next[i]]
            next[i] ← next[next[i]]
```

$T_p = O(\log n)$

Work = $\Theta(n \log n)$

```
for each processor i do
    if next[i]=NIL then d[i]←0
    else d[i]←1
while exist i | next[i]≠NIL do
    for each processor i do
        if next[i] ≠ NIL then
            d[i]← d[i] + d[next[i]]
            next[i] ← next[next[i]]
```



Prefix Computation

A ***prefix computation*** is defined in terms of a binary associative operator ❖. It takes as input a sequence <x_1, x_2, …, x_n> and produces as output a sequence <y_1, y_2, …, y_n>, where $y_1 = x_1$ and

$$y_k = y_{k-1} ❖, x_k$$
$$= x_1 ❖ x_2 ❖ \dots ❖ \quad x_k$$

Prefix sums computation:

input $a_1\ a_2 \dots a_n$

output: $a_1, a_1 + a_2, a_1 + a_2 + a_3, \dots$

Prefix Notation

A ***prefix computation*** is defined in terms of a binary associative operator ❖. It takes as input a sequence <x_1, x_2, …, x_n> and produces as output a sequence <y_1, y_2, …, y_n>, where $y_1 = x_1$ and

$$y_k = y_{k-1} ❖, x_k$$
$$= x_1 ❖ x_2 ❖ \dots ❖ \quad x_k$$

$$[i,j]= x_i ❖ x_{i+1} ❖ \dots ❖ \quad x_j$$

$$[k,k]= x_k$$

$$[i,k]= [i,j] ❖ [j+1,k]$$

Goal: compute y_k=[1,k] for k=1,2,…,n

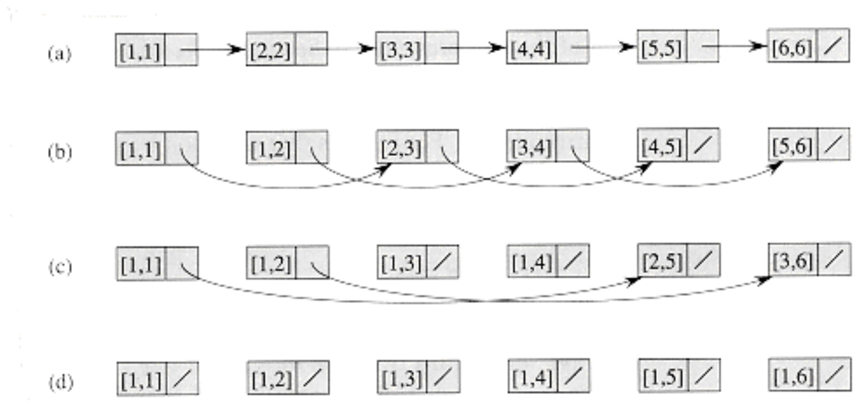
WORKSHEET:List prefix

```
for each processor i do

while exist i | next[i]≠NIL do
    for each processor i do
        if next[i] ≠ NIL then

```

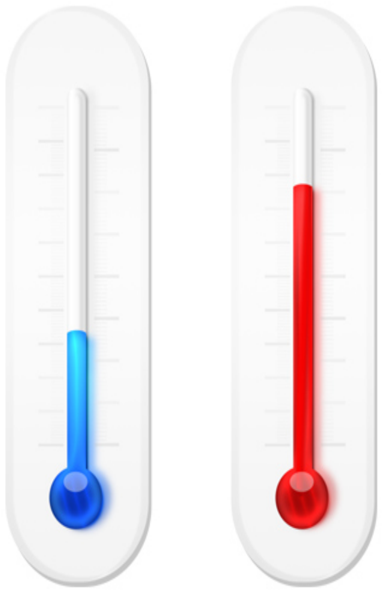
Fill these in!

y[i]= x[1]+ x[2]+…+ x[i]

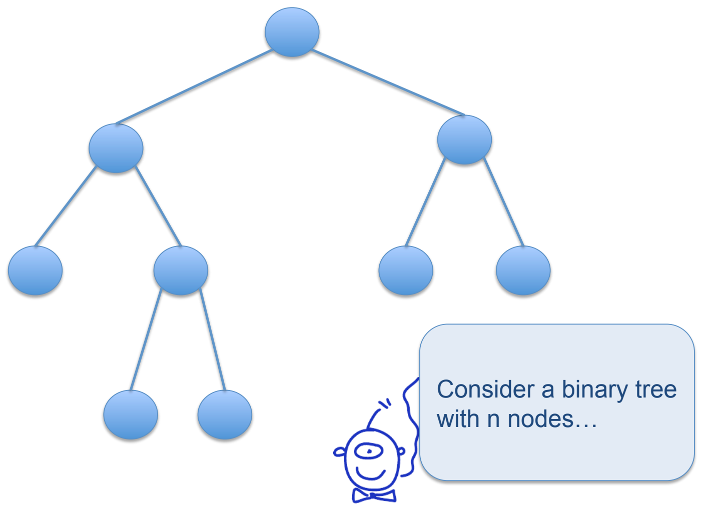
List Prefix



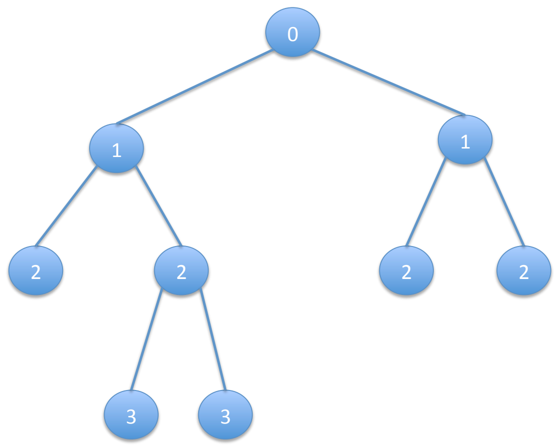
(a) [1,1] [2,2] [3,3] [4,4] [5,5] [6,6]
(b) [1,1] [1,2] [2,3] [3,4] [4,5] [5,6]
(c) [1,1] [1,2] [1,3] [1,4] [2,5] [3,6]
(d) [1,1] [1,2] [1,3] [1,4] [1,5] [1,6]



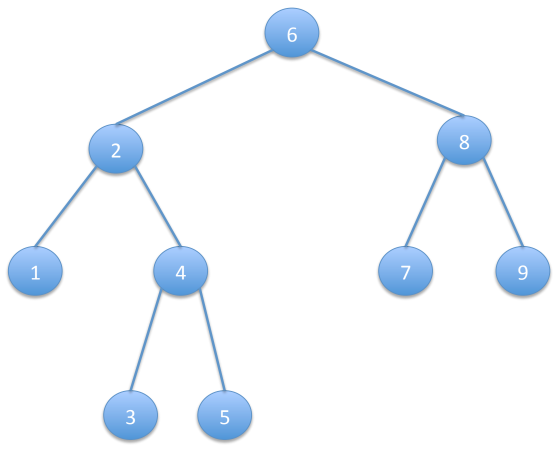
Fun with Trees (in Parallel!)



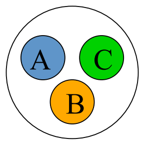
Depth from root…



Inorder traversal number…



Binary tree



- Let T be a binary tree stored in a PRAM
- Each node *i* has fields parent[i], left[i] and right[i], which point to node i's parent, left child and right child respectively
- Let's assume that each node is identified by a non-negative integer
- Also we associate not one but 3 processes with each node; we call these node's A,B and C processors
- Mapping between each node *i* and its 3 processors A,B and C: 3i, 3i+1, 3i+2

Computing depth of each node in an n node tree takes O(n) time on a serial RAM

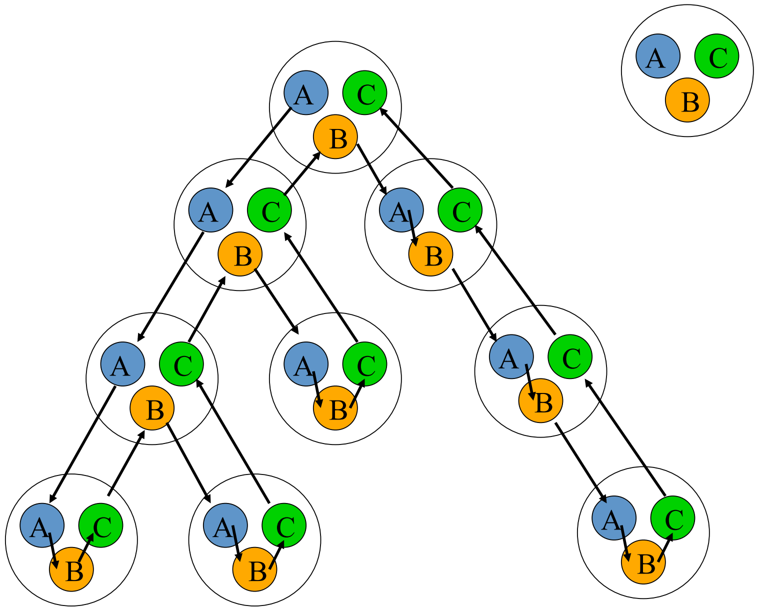
- A simple parallel algorithm to compute depths propagates a "wave" downward from the root of the tree.
  - The wave reaches all nodes at the same depth simultaneously, and thus by incrementing a counter carried along with the wave, we can compute the depth of each node.
- This parallel algorithm works well on a complete binary tree, since it runs in time proportional to the tree's height.
- But the height of the tree could be as large as n-1

Using the Euler-tour technique we can compute node depths in O(log n) time on an EREW PRAM

- An Euler-tour of a graph is a cycle that traverses each edge exactly once, although it may visit a vertex more than ones
  - A connected, directed graph has an Euler tour if and only if for all vertices v, the in-degree of v equals the out degree of v
  - Since each undirected edge (u,v) in an undirected graph maps to two directed edges (u,v) and (v,u) in the directed version, the directed of any connected, undirected graph (and therefore of any undirected tree) has an Euler tour

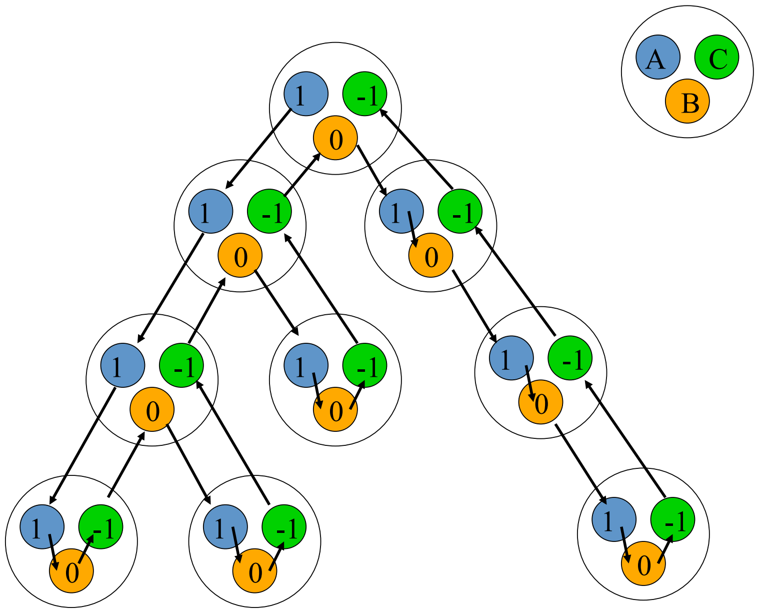
Depth of nodes computation

- First we form an Euler tour of the directed version of T.
- The tour corresponds to walk of the tree with the following structure:
  - A node's A processor points to the A processor of its left child, if it exist, and otherwise to its own B processor
  - A node's B processor points to the A processor of its right child, if it exist, and otherwise to its own C processor
  - A node's C processor points to the B processor of its parent, if it is a left child and to the C processor of its parent if it is a right child. The root's C processor points to NIL.



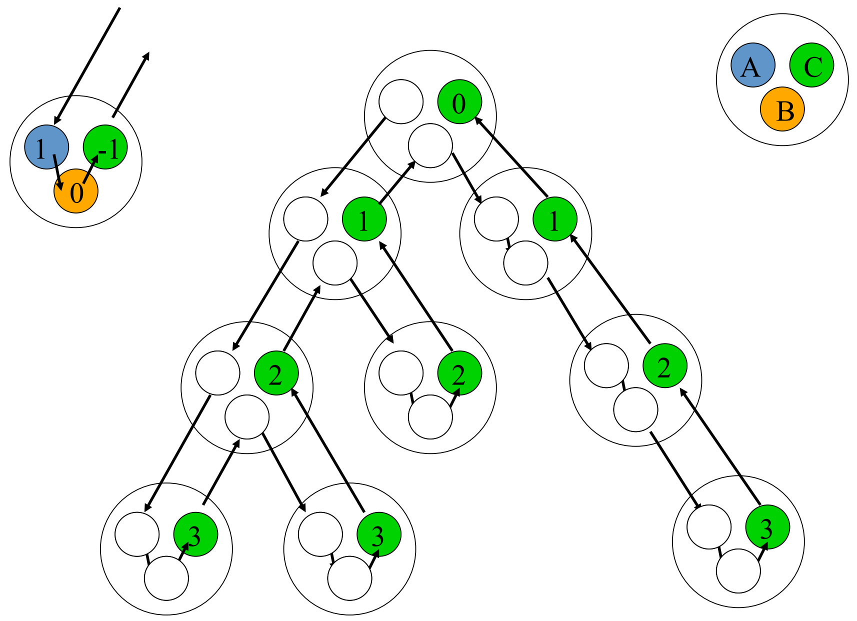
First step

- Thus, the head of the linked list formed by the Euler tour is the root's A processor, and the tail is the root's C processor.
- Given the pointers composing the original tree, an Euler tour can be constructed in O(1) time.
- Once we have linked list representing the Euler tour of T, we place
  - a 1 in each A processor,
  - a 0 in each B processor and
  - a –1 in each C processor



Second step

- We then perform a parallel prefix computation using ordinary addition as the associative operation
- We claim that after performing the parallel prefix computation, the depth of each node resides in the node's C processor. Why?

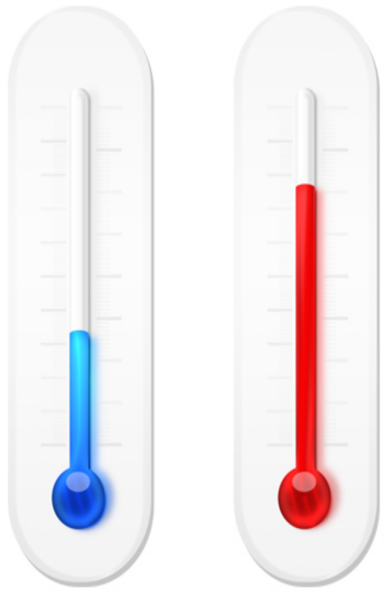


WHY ???

- The numbers are placed into the A,B and C processors in such a way that the net effect of visiting a subtree is to add 0 to the running sum
- The A processor of each node *i* contributes 1 to running sum
- The B processor of each node *i* contributes 0 because the depth of the node *i's* left child equals the depth of the node *i's* right child
- The C processor contributes –1, so the entire visit to the subtree rooted at node *i* has no effect on the running sum.

Conclusion

- The list representing Euler-tour can be computed in O(1) time.
- It has 3n objects, and thus the parallel prefix computation takes only O(log n) time
- Thus the total amount of time to compute all node depths is O(log n).
- Because no concurrent memory accesses are needed, the algorithm is an EREW algorithm.



Transitive Closure

- **TC problem** has numerous applications in many areas of computer science.
- **Lack of course-grained algorithms** for distributed environments with slow communication.
- **Decreasing the number of dependences** in a solution could improve a performance of the algorithm.

What is transitive closure?

GENERIC TRANSITIVE CLOSURE PROBLEM (TC)

Input: a matrix A with elements from a semiring S= < ⊕,⊗ >

Output: the matrix A*, A*(i,j) is the sum of all simple paths from i to j

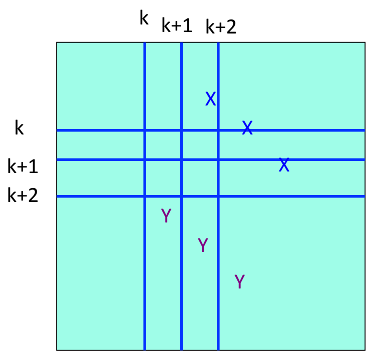
< ⊕ , ⊗ > TC

< or , and > boolean closure - TC of a directed graph

< MIN, + > all pairs shortest path

<MIN, MAX> minimum spanning tree {all(i,j): A(i,j)=A*(i,j)}

Floyd-Warshall algorithm

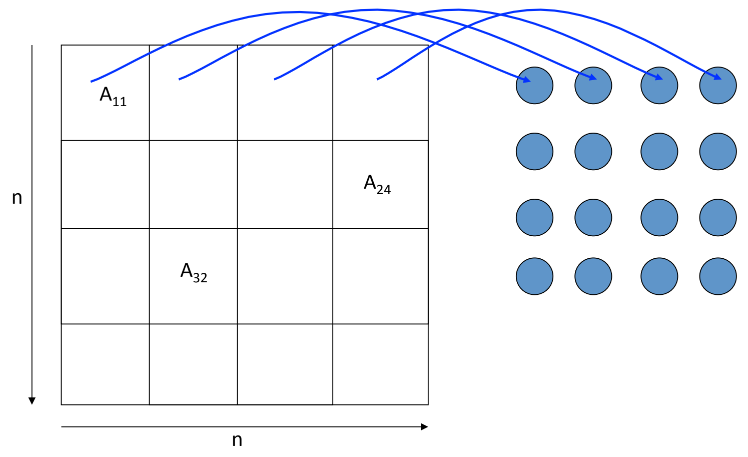


```
for k:=1 to n
  for all 1≤i,j≤n parallel do
     Operation(i, k, j)
```

Operation(i, k, j): a(i,j):=a(i,j) ⊕ a(i,k) ⊗ a(k,j)

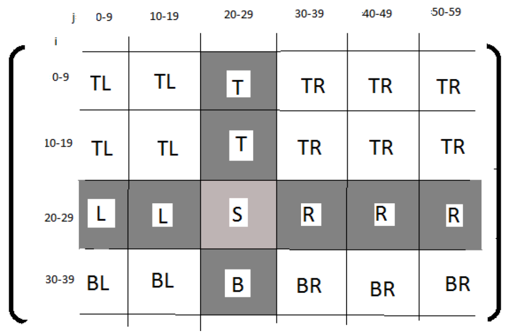
Floyd-Warshall algorithm

Coarse-Grained computations



Blocked Floyd Warshall

Blocked Floyd Warshall



i \ j	0-9	10-19	20-29	30-39	40-49	50-59
0-9	TL	TL	T	TR	TR	TR
10-19	TL	TL	T	TR	TR	TR
20-29	L	L	S	R	R	R
30-39	BL	BL	B	BR	BR	BR

When we relax from k=20-29 for this matrix the block dependencies are shown

Dependencies (for k=20-29) shown in the figure

S - Self Dependent Block

L,R,T,B - Blocks dependent on themselves and self dependent blocks

TR -Blocks dependent on themselves and some T,R blocks

BL - blocks dependent on themselves and some L,B blocks

BR - Blocks dependent on themselves and some B,R blocks

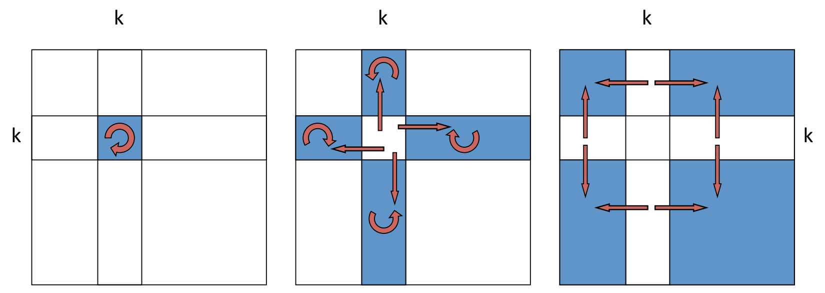
TL - Blocks dependent on themselves and some T,L blocks

Course-grained Floyd-Warshall algorithm

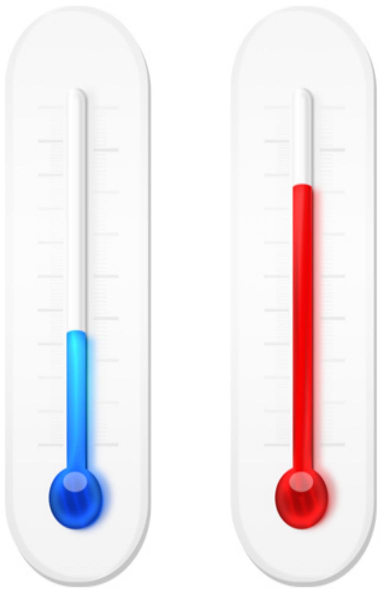
```
Algorithm Blocks-Warshall
for k :=1 to N do
  A(K,K):=A*(K,K)
    for all 1 ≤ I,J ≤ N, I ≠ K ≠ J  parallel do
        Block-Operation(K,K,J) and Block-Operation(I,K,K)
  for all 1 ≤ I,J ≤ N  parallel do
    Block-Operation(I,K,J)
```

Block-Operation(I, K, J): A(I,J):=A(I,J) ⊕ A(I,K) ⊗ A(K,K) ⊗ A(K,J)

Implementation of Warshall TC Algorithm



The implementation in terms of multiplication of submatrices



Synchronization and Tada!