Definition: The cost of a PRAM computation is the product of the parallel time complexity and the number of processors used.

Various PRAM models differ in how they handle the read or write conflicts;

- **EREW (Exclusive Read Exclusive Write):** Read and write conflicts are not allowed
- **CREW (Concurrent Read Exclusive Write):** Concurrent read allowed (i.e. multiple processors are allowed to read from the same memory location), but concurrent write is not allowed (*Default PRAM*)
- **CRCW:** Concurrent read and concurrent write is allowed (W-RAM). There are different policies to handle the concurrent write operation.

- **COMMON:** All processors writing to the same memory location must write same value.

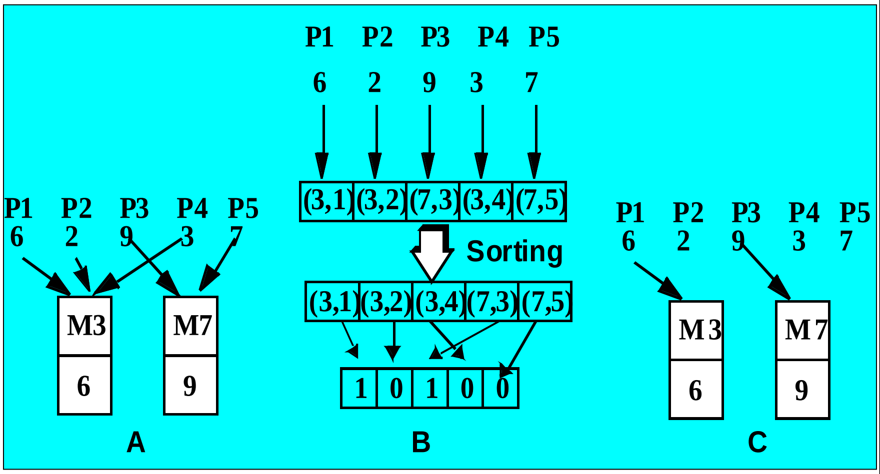
- **ARBITRARY:** If multiple processors concurrently write to the same global address, one of the competing processors is arbitrarily chosen and its value is written into the register.

- **PRIORITY:** If multiple processors concurrently write to the same global address, the processor with the lowest index succeeds its value into the memory location.

Relative strength of the models

Lemma: (Cole [88]) A p-processor EREW PRAM can sort a p-element array stored in global memory in (log n) time.

Theorem: A p-processors PRIORITY PRAM can be simulated by a p-processor EREW PRAM with the time complexity increased by a factor of (log n).



Two statements are used in the algorithm description

1. spawn (< processor names>) (log p) time needed

2. for all <processor list> do <{statement name} endfor

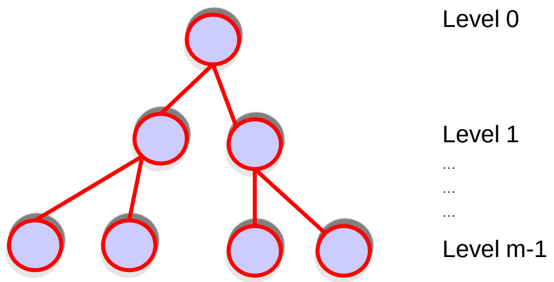
Binary tree is one of the most important data structure which can be exploited for the parallel algorithm design.

- **top down**
  1. Broadcast
  2. Divide and Conquer

- bottom up

fan-in or reduction

Balanced Binary tree method



Balanced Binary tree method structure

for levels m-1, m-2,…, 0 do

for each vertex **v** at internal node in parallel do

value[v] = value[LeftChild[**v**]] ***op*** value[RightChild[**v**]]

output = value[root];

Maximum of n = 2^m numbers stored in an array A of dimension (2n-1) from A(n), A(n+1),...,A(2n-1). At the end A(1) stores the result.

for k = m-1 step -1 to 0 do

for all j, $2^k \le j \le 2^{k+1} - 1$, in parallel do

A(j) = max{A(2j), A(2j+1)}

A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13	A14	A15
							4	3	5	1	7	9	4	0

A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13	A14	A15
			4	5	9	4	4	3	5	1	7	9	4	0

Max of

A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13	A14	A15
	5	9	4	5	9	4	4	3	5	1	7	9	4	0

A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13	A14	A15
9	5	9	4	5	9	4	4	3	5	1	7	9	4	0

- No of processors used, Parallel time required ?

SUM (EREW PRAM)

Global n, A[0,...,(n-1)],j

begin

  spawn (P0,P1,....,P $\lfloor n/2 \rfloor - 1$)

  for all Pi where $0 \le i \le \lfloor n/2 \rfloor - 1$ do

    for j = 0 to $\lceil \log n - 1 \rceil$ do

      if (i mod 2^j)= 0 and (2i + 2^j)< n then

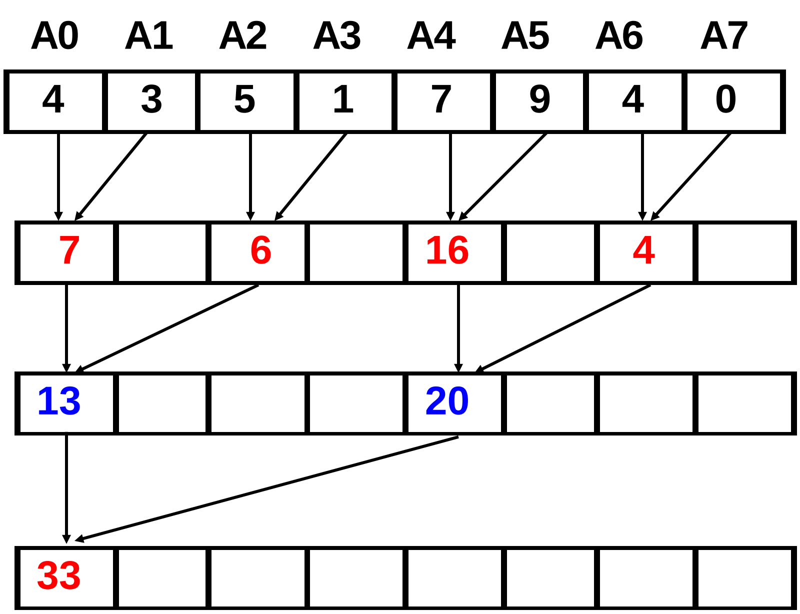
        A[2i] = A[2i] + A[2i + 2^j]

      endif

    endfor

  endfor

end



Applications of Prefix Computation

- Knapsack Problem
- Job Sequencing with deadline
- Compiler Design
- Computational Biology
- Evaluation of Polynomials
- Solving System of Linear Equations
- Polynomial Interpolation

PREFIX.SUMS (CREW PRAM)

Global n, A(0), A(1),...,A(n-1), j

begin

spawn(P_1, P_2, ..., P_{n-1})

for all P_i where $0 \le i \le n-1$ do

for j = 0 to $\lceil \log n \rceil - 1$ do

if (i -2^j) >= 0 then

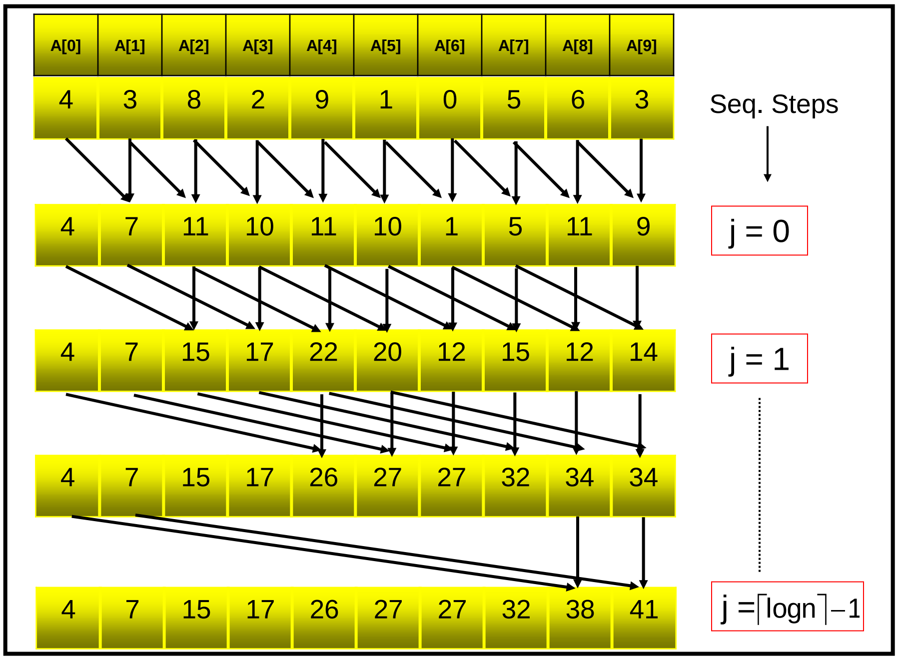
A[i] = A[i] + A[i -2^j]

endif

endfor

endfor

end



Exercise: n = 2^m numbers stored in an array A of dimension (2n-1) from A(n), A(n+1),...,A(2n-1). Write an algorithm for obtaining the prefix sum of these numbers, at the end A(i), $1 \leq i \leq n$ stores the result.

Doubling techniques

Normally applied to an array or to a list of elements. The computation proceeds by a recursive application of the computation in hand to all the elements.

Linked List Ranking

- Given a linked list, stored in an array, compute the distance of each element from the end (either end) of the list.
- Called Pointer Jumping when using pointers.
- Don't destroy original list!

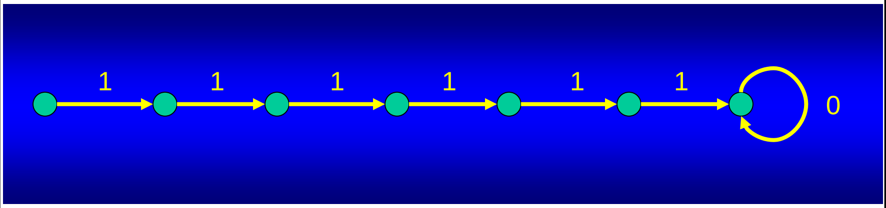
The distance doubles in successive steps. Thus after **k** iterations computation to all elements at distance 2^k is performed.

Value in an array ***next*** represents linked list

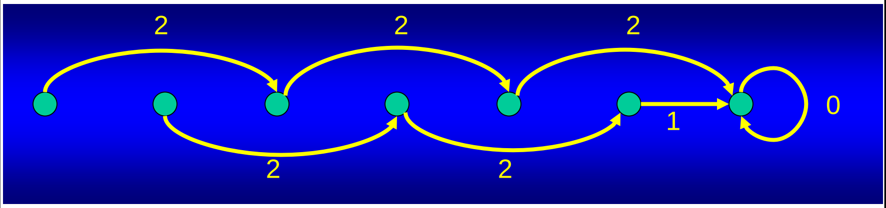
Value in an array ***position*** contain original distance of each element from end of the list.

```
Global: n, position[0..(n-1)], next[0..(n-1)], j
LIST.RANKING(CREW PRAM)
begin
        spawn(P_0, P_1, ...., P_{n-1})
        for all P_i where 0 ≤ i ≤ n−1 do
                if next[i] = i then position[i] := 0
                else position[i] := 1
                endif
                for j = 1 to ⌈logn⌉−1 do
                        position[i] := position[i] + position[next[i]]
                        next[i] = next[next[i]]
                endfor
        endfor
end
```

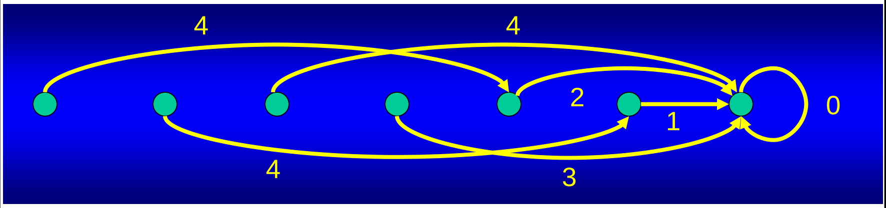

Step 1:



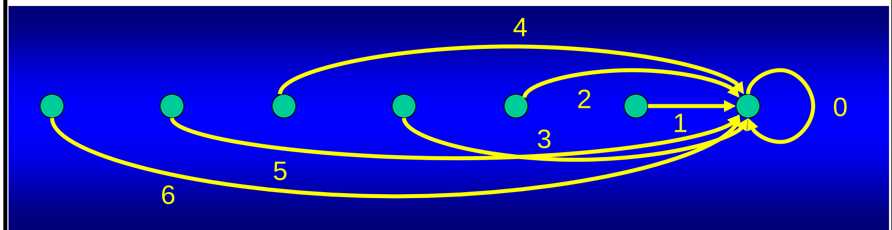
Step 2:



Step 3:



Step 4:



Parallel time complexity, Processors

Work Analysis

- Number of Steps: Tp = O(Log N)
- Number of Processors: N
- Work = O(N log N)
- Sequential = O(N)
- Optimal??

Applications of List Ranking

- Expression Tree Evaluation
- Parentheses Matching
- Tree Traversals
- Ear–Decomposition of Graphs
- Euler tour of trees
- - - - many others

Merging two sorted lists

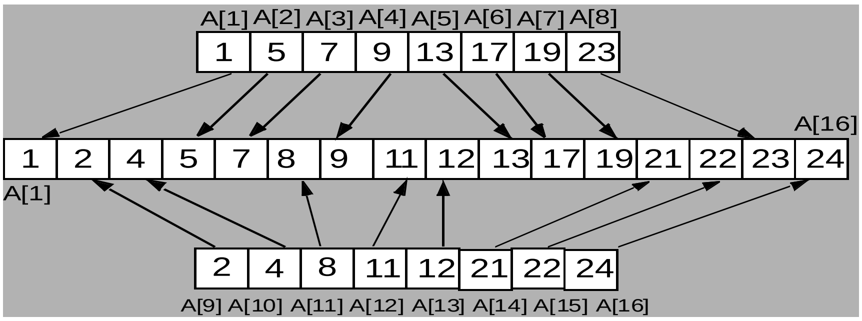
Best known sequential algorithm needs O(n) time. Every processor finds the position of ***its own*** element on the other list using binary search, making an algorithm that takes *O*(log n) parallel time.

Assumption: Two lists and their unions have disjoint values.

```
Global A[1..n]
MERGE.LISTS(CREW PRAM):
Local x, low, high, index
begin
  spawn(P_1, P_2, ...., P_n)
  for all P_i where 1 ≤ i ≤ n do
    if (i <= n/2) then
      low := (n/2)+1
      high := n
    else
      low := 1
      high := n/2
    endif
```



```
{Each processor performs binary search}
    x := A[i]
    repeat
      index:= ⌊(low+high)/2⌋
      if x < A[index] then
        high := index-1
      else
        low := index + 1
      endif
    until (low > high)
    {put values in correct position on merged list}
    A[high+i-n/2] := x
  endfor
end
```



Parallel time = ?, No. of processors = ?

Do we need so many processors

(Cost) Optimal parallel algorithm: One in which the product of number of processor *p* used and parallel time *t* is linear in problem size **S**, i.e. **pt = O(S)**

Reducing the number of processors

Suppose we have designed an algorithm working in parallel time t with p processors, here we assume that p is the maximum number of operations executed in the same parallel step.

Maximum finding algorithm takes O(log n) time with the $p \geq n/2$ processors, in fact n/2 processors are required only at the beginning of the procedure. Most of the processors are sitting idle.

suppose we have p<n/2 processors. Partition **n** elements in **p** groups. **p-1** such group will be having $\lceil n/p \rceil$ elements and remaining group contains

$(n-(p-1)\lceil n/p \rceil$ <= ) elements.

suppose we have $p<n/2$ processors. Partition **n** elements in **p** groups. **p-1** such group will be having $\lceil n/p \rceil$ elements and remaining group contains

$$(n-(p-1)\lceil n/p \rceil \;\; <= \lceil n/p \rceil\;) \text{ elements.}$$

Assign a processor to each group which finds maximum in $\lceil n/p \rceil - 1$ time each, in parallel, later log p time using balanced binary tree method.

Thus overall time is $\lceil n/p \rceil - 1 + \log p$ with $p < n/2$ processors.

What if $p = n / \log n$

Brent's theorem: Let **A** be a given parallel algorithm with computation time ***t***, if parallel algorithm performs ***m*** computational operations then ***p*** processors can execute algorithm **A** in time 0(**m/p + t)**.

Definition: The set $(\log n)^{0(n)}$ is called the set of polylogarithmic function.

Theorem(Parallel computation thesis): The class of problems solvable in time $T(n)^{0(n)}$ by a PRAM is equal to the class of problems solvable in work space $T(n)^{0(n)}$ by a RAM, if $T(n) >= \log n$.