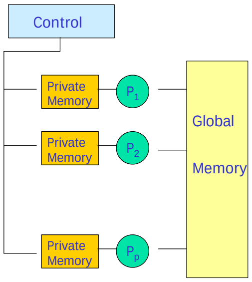
PRAM Algorithms

Arvind Krishnamurthy
Fall 2004

Parallel Random Access Machine (PRAM)

- Collection of numbered processors
- Accessing shared memory cells
- Each processor could have local memory (registers)
- Each processor can access any shared memory cell in unit time
- Input stored in shared memory cells, output also needs to be stored in shared memory
- PRAM instructions execute in 3-phase cycles
  - Read (if any) from a shared memory cell
  - Local computation (if any)
  - Write (if any) to a shared memory cell
- Processors execute these 3-phase PRAM instructions synchronously

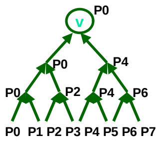


Shared Memory Access Conflicts

- Different variations:
  - Exclusive Read Exclusive Write (EREW) PRAM: no two processors are allowed to read or write the same shared memory cell simultaneously
  - Concurrent Read Exclusive Write (CREW): simultaneous read allowed, but only one processor can write
  - Concurrent Read Concurrent Write (CRCW)
- Concurrent writes:
  - Priority CRCW: processors assigned fixed distinct priorities, highest priority wins
  - Arbitrary CRCW: one randomly chosen write wins
  - Common CRCW: all processors are allowed to complete write if and only if all the values to be written are equal

A Basic PRAM Algorithm

- Let there be “n” processors and “2n” inputs
- PRAM model: EREW
- Construct a tournament where values are compared



Processor k is active in step j
if (k % 2^j) == 0
At each step:
Compare two inputs,
Take max of inputs,
Write result into shared memory

Details:
Need to know who is the “parent” and whether you are left or right child
Write to appropriate input field

PRAM Model Issues

- Complexity issues:
  - Time complexity = O(log n)
  - Total number of steps = n * log n = O(n log n)
- Optimal parallel algorithm:
  - Total number of steps in parallel algorithm is equal to the number of steps in a sequential algorithm
- Use n/logn processors instead of n
- Have a local phase followed by the global phase
- Local phase: compute maximum over log n values
  - Simple sequential algorithm
  - Time for local phase = O(log n)
- Global phase: take (n/log n) local maximums and compute global maximum using the tournament algorithm
  - Time for global phase = O(log (n/log n)) = O(log n)

Time Optimality

- Example: n = 16
- Number of processors, p = n/log n = 4
- Divide 16 elements into four groups of four each
- Local phase: each processor computes the maximum of its four local elements
- Global phase: performed amongst the maximums computed by the four processors

Finding Maximum: CRCW Algorithm

Given n elements A[0, n-1], find the maximum.
With n^2 processors, each processor (i,j) compare A[i] and A[j], for 0≤ i, j ≤n-1.

FAST-MAX(A):

```
n←length[A]
for i ←0 to n-1, in parallel
      do m[i] ←true
for i ←0 to n-1 and j ←0 to n-1, in parallel
      do if A[i] < A[j]
          then m[i] ←false
for i ←0 to n-1, in parallel
      do if m[i] =true
          then max ← A[i]
return max
```

		$A[j]$					
		5	6	9	2	9	m
$A[i]$	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

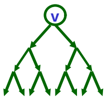
max=9

The running time is $O(1)$.

Note: there may be multiple maximum values, so their processors Will write to max concurrently. Its *work* = $n^2 \times O(1) = O(n^2)$.

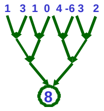
Broadcast and reduction

- Broadcast of 1 value to p processors in log p time



Broadcast

- Reduction of p values to 1 in log p time
- Takes advantage of associativity in +,*, min, max, etc.



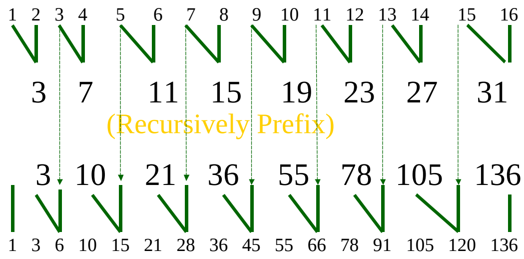
Add-reduction

Scan (or Parallel prefix)

- What if you want to compute partial sums
- Definition: the parallel prefix operation take a binary associative operator ⊖, and an array of n elements
  $[a_0, a_1, a_2, \ldots a_{n-1}]$
  and produces the array
  $[a_0, (a_0 ⊖ a_1), \ldots (a_0 ⊖ a_1 ⊖ \ldots ⊖ a_{n-1})]$
- Example: add scan of
  [1, 2, 0, 4, 2, 1, 1, 3] is [1, 3, 3, 7, 9, 10, 11, 14]
- Can be implemented in O(n) time by a serial algorithm
  - Obvious n-1 applications of operator will work

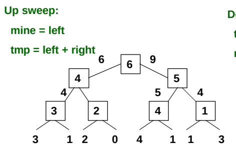
Prefix Sum in Parallel

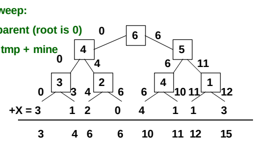
Algorithm: 1. Pairwise sum 2. Recursively Prefix 3. Pairwise Sum



Implementing Scans

- **Tree summation 2 phases**
  - **up sweep**
    - **get values L and R from left and right child**
    - **save L in local variable Mine**
    - **compute Tmp = L + R and pass to parent**
  - **down sweep**
    - **get value Tmp from parent**
    - **send Tmp to left child**
    - **send Tmp+Mine to right child**





E.g., Using Scans for Array Compression

- Given an array of n elements
  $[a_0, a_1, a_2, \ldots a_{n-1}]$
  and an array of flags
  [1,0,1,1,0,0,1,...]
  compress the flagged elements
  $[a_0, a_2, a_3, a_6, \ldots]$
- Compute a "prescan" i.e., a scan that doesn't include the element at position i in the sum
  [0,1,1,2,3,3,4,...]
- Gives the index of the i^{th} element in the compressed array
  - If the flag for this element is 1, write it into the result array at the given position

E.g., Fibonacci via Matrix Multiply Prefix

$$\boxed{F_{n+1} = F_n + F_{n-1}}$$

$$\boxed{\begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix}}$$

Can compute all F_n by matmul_prefix on

$$\left[\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \right]$$

then select the upper left entry

Slide source: Alan Edelman, MIT

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

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

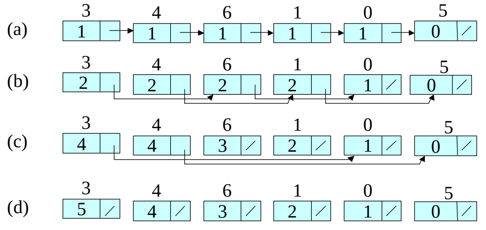
- Serial algorithm: $\Theta(n)$

List ranking –EREW algorithm

- LIST-RANK(L) (in O(lg n) time)

```
for each processor i, in parallel
    do if next[i]=nil
        then d[i]←0
        else d[i]←1
while there exists an object i such that next[i]≠nil
    do for each processor i, in parallel
        do if next[i]≠nil
            then d[i]← d[i]+ d[next[i]]
                 next[i] ←next[next[i]]
```

List-ranking –EREW algorithm



Recap

- PRAM algorithms covered so far:
  - Finding max on EREW and CRCW models
  - Time optimal algorithms: number of steps in parallel program is equal to the number of steps in the best sequential algorithm
    - Always qualified with the maximum number of processors that can be used to achieve the parallelism
  - Reduction operation:
    - Takes a sequence of values and applies an associative operator on the sequence to distill a single value
    - Associative operator can be: +, max, min, etc.
    - Can be performed in O(log n) time with up to O(n/log n) procs
  - Broadcast operation: send a single value to all processors
    - Also can be performed in O(log n) time with up to O(n/log n) procs

Scan Operation

- Used to compute partial sums
- Definition: the parallel prefix operation take a binary associative operator $\ominus$, and an array of n elements

$$[a_0, a_1, a_2, \ldots a_{n-1}]$$

and produces the array

$$[a_0, (a_0 \ominus a_1), \ldots (a_0 \ominus a_1 \ominus \ldots \ominus a_{n-1})]$$

```
Scan(a, n):
        if (n == 1) {  s[0] = a[0]; return s; }
        for (j = 0 … n/2-1)
                x[j] = a[2*j] ⊖ a[2*j+1];
        y = Scan(x, n/2);
        for odd j in {0 … n-1}
                s[j] = y[j/2];
        for even j in {0 … n-1}
                s[j] = y[j/2] ⊖ a[j];
        return s;
```

Work-Time Paradigm

- Associate two complexity measures with a parallel algorithm
- S(n): time complexity of a parallel algorithm
  - Total number of steps taken by an algorithm
- W(n): work complexity of the algorithm
  - Total number of operations the algorithm performs
  - $W_j(n)$: number of operations the algorithm performs in step j
  - $W(n) = \Sigma W_j(n)$ where j = 1…S(n)
- Can use recurrences to compute W(n) and S(n)

Recurrences for Scan

```
Scan(a, n):
        if (n == 1) {  s[0] = a[0]; return s; }
        for (j = 0 … n/2-1)
                x[j] = a[2*j] ⊖ a[2*j+1];
        y = Scan(x, n/2);
        for odd j in {0 … n-1}
                s[j] = y[j/2];
        for even j in {0 … n-1}
                s[j] = y[j/2] ⊖ a[j];
        return s;
```

$$W(n) = 1 + n/2 + W(n/2) + n/2 + n/2 + 1$$
$$= 2 + 3n/2 + W(n/2)$$
$$S(n) = 1 + 1 + S(n/2) + 1 + 1 = S(n/2) + 4$$

Solving, W(n) = O(n); S(n) = O(log n)

Brent's Scheduling Principle

- A parallel algorithm with step complexity S(n) and work complexity W(n) can be simulated on a p-processor PRAM in no more than $T_C(n,p) = W(n)/p + S(n)$ parallel steps
  - S(n) could be thought of as the length of the "critical path"
- Some schedule exists; need some online algorithm for dynamically allocating different numbers of processors at different steps of the program
- No need to give the actual schedule; just design a parallel algorithm and give its W(n) and S(n) complexity measures
- Goals:
  - Design algorithms with $W(n) = T_S(n)$, running time of sequential algorithm
    - Such algorithms are called work-efficient algorithms
  - Also make sure that S(n) = poly-log(n)
  - Speedup = $T_S(n) / T_C(n,p)$

Application of Brent's Schedule to Scan

- Scan complexity measures:
  - W(n) = O(n)
  - S(n) = O(log n)
- $T_C(n,p) = W(n)/p + S(n)$
- If p equals 1:
  - $T_C(n,p) = O(n) + O(\log n) = O(n)$
  - Speedup = $T_S(n) / T_C(n,p) = 1$
- If p equals n/log(n):
  - $T_C(n,p) = O(\log n)$
  - Speedup = $T_S(n) / T_C(n,p) = n/\log n$
- If p equals n:
  - $T_C(n,p) = O(\log n)$
  - Speedup = n/logn
- Scalable up to n/log(n) processors

Segmented Operations

Inputs = Ordered Pairs
(operand, boolean)
e.g. (x, T) or (x, F)

Change of segment indicated by switching T/F

+ 2	(y, T)	(y, F)
(x, T)	(x+ y, T)	(y, F)
(x, F)	(y, T)	(x⊕y, F)

e. g.	1	2	3	4	5	6	7	8
	T	T	F	F	F	T	F	T
Result	1	3	3	7	12	6	7	8

Parallel prefix on a list

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

Prefix computation

- LIST-PREFIX(L)

```
for each processor i, in parallel
    do y[i]← x[i]
while there exists an object i such that prev[i]≠nil
    do for each processor i, in parallel
        do if prev[i]≠nil
            then y[prev[i]]← y[i] ⊗ y[prev[i]]
                prev[i] ← prev[prev[i]]
```

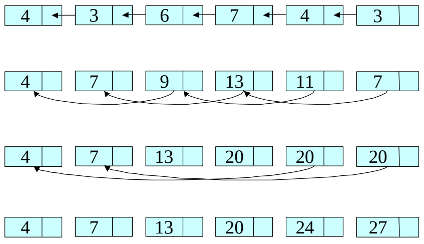
List Prefix Operations

- What is S(n)?
- What is W(n)?
- What is speedup on n/logn processors?

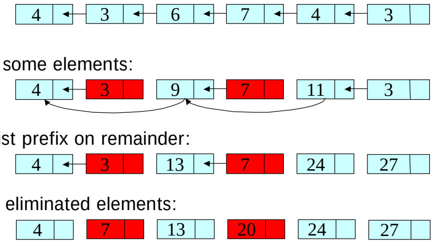
Announcements

- Readings:
  - Lecture notes from Sid Chatterjee and Jans Prins
  - Prefix scan applications paper by Guy Blelloch
  - Lecture notes from Ranade (for list ranking algorithms)
- Homework:
  - First theory homework will be on website tonight
  - To be done individually
- TA office hours will be posted on the website soon

List Prefix



Optimizing List Prefix



Eliminate some elements:

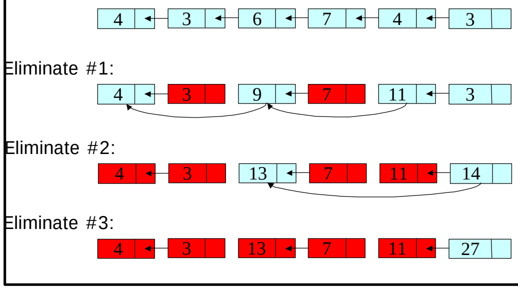
Perform list prefix on remainder:

Integrate eliminated elements:

Optimizing List Prefix

- Randomized algorithm:
  - Goal: achieve W(n) = O(n)
- Sketch of algorithm:
  1. Select a set of list elements that are non adjacent
  2. Eliminate the selected elements from the list
  3. Repeat steps 1 and 2 until only one element remains
  4. Fill in values for the elements eliminated in preceding steps in the reverse order of their elimination

Optimizing List Prefix



Randomized List Ranking

- Elimination step:
  - Each processor is assigned O(log n) elements
  - Processor j is assigned elements j*logn … (j+1)*logn –1
  - Each processor marks the head of its queue as a candidate
  - Each processor flips a coin and stores the result along with the candidate
  - A candidate is eliminated if its coin is a HEAD and if it so happens that the previous element is not a TAIL or was not a candidate

Find root –CREW algorithm

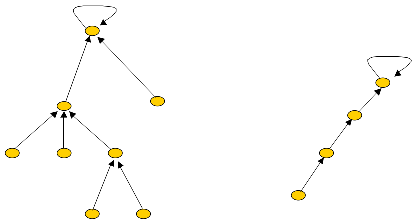
- Suppose a forest of binary trees, each node *i* has a pointer *parent*[*i*].
- Find the identity of the tree of each node.
- Assume that each node is associated a processor.
- Assume that each node *i* has a field *root*[*i*].

Find-roots –CREW algorithm

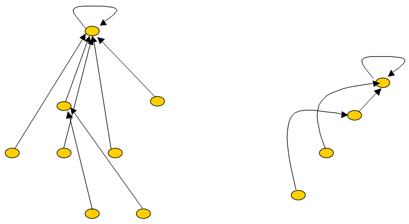
- FIND-ROOTS(F)

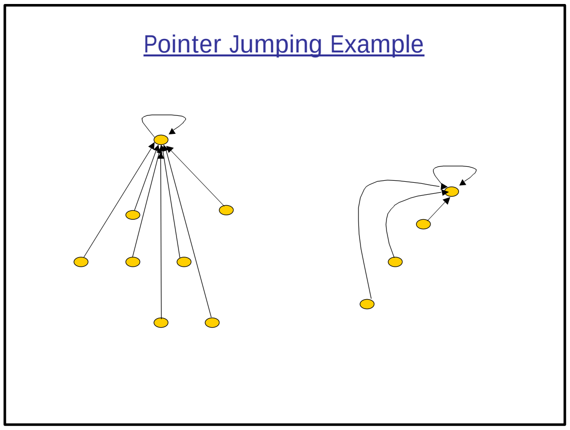
```
for each processor i, in parallel
  do if parent[i] = nil
       then root[i]←i
while there exist a node i such that parent[i] ≠ nil
   do for each processor i, in parallel
        do if parent[i] ≠ nil
             then root[i] ← root[parent[i]]
                  parent[i] ← parent[parent[i]]
```

Pointer Jumping Example



Pointer Jumping Example





Pointer Jumping Example

Analysis

- Complexity measures:
  - What is W(n)?
  - What is S(n)?
- Termination detection: When do we stop?
- All the writes are exclusive
- But the read in line 7 is concurrent, since several nodes may have same node as parent.

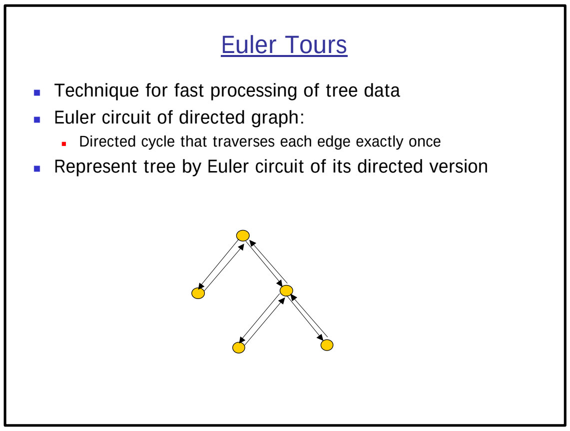
Find roots –CREW vs. EREW

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

$\Omega(\lg n)$

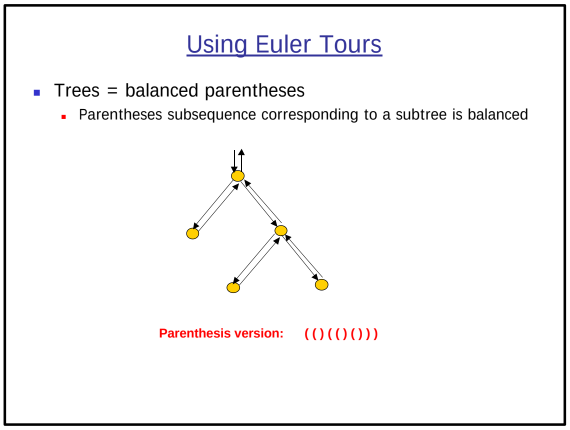
Argument: when exclusive read, a given peace of information can only be copied to one other memory location in each step, thus the number of locations containing a given piece of information at most doubles at each step. Looking at a forest with one tree of *n* nodes, the root identity is stored in one 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.

So CREW: $O(\lg d)$ and EREW: $\Omega(\lg n)$.
If $d=2^{o(\lg n)}$, CREW outperforms any EREW algorithm.
If $d=\Theta(\lg n)$, then CREW runs in $O(\lg \lg n)$, and EREW is much slower.



Euler Tours

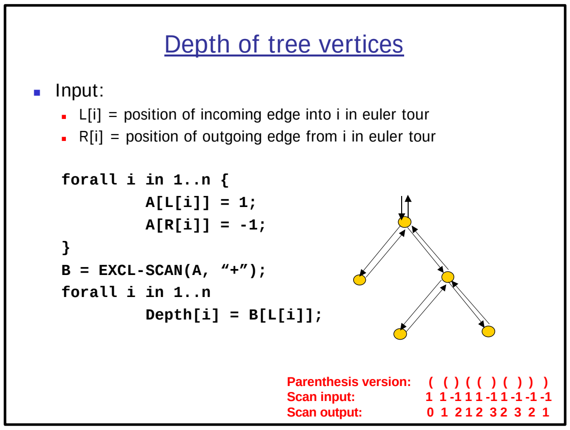
- Technique for fast processing of tree data
- Euler circuit of directed graph:
  - Directed cycle that traverses each edge exactly once
- Represent tree by Euler circuit of its directed version



Using Euler Tours

- Trees = balanced parentheses
  - Parentheses subsequence corresponding to a subtree is balanced

Parenthesis version: (()(()()))



Depth of tree vertices

- Input:
  - L[i] = position of incoming edge into i in euler tour
  - R[i] = position of outgoing edge from i in euler tour

```
forall i in 1..n {
        A[L[i]] = 1;
        A[R[i]] = -1;
}
B = EXCL-SCAN(A, “+”);
forall i in 1..n
        Depth[i] = B[L[i]];
```

Parenthesis version: (( ) (( ) ( )))
Scan input: 1 1 -1 1 1 -1 1 -1 -1 -1
Scan output: 0 1 2 1 2 3 2 3 2 1

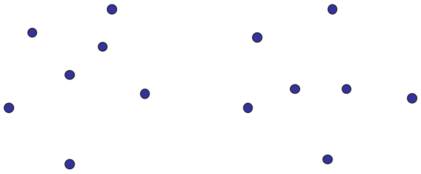
Divide and Conquer

- Just as in sequential algorithms
  - Divide problems into sub-problems
  - Solve sub-problems recursively
  - Combine sub-solutions to produce solution
- Example: planar convex hull
  - Give set of points sorted by x-coord
  - Find the smallest convex polygon that contains the points

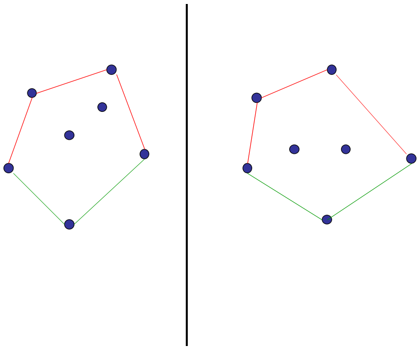
Convex Hull

- Overall approach:
  - Take the set of points and divide the set into two halves
  - Assume that recursive call computes the convex hull of the two halves
  - Conquer stage: take the two convex hulls and merge it to obtain the convex hull for the entire set
- Complexity:
  - W(n) = 2*W(n/2) + merge_cost
  - S(n) = S(n/2) + merge_cost
  - If merge_cost is O(log n), then S(n) is O(log^{2}n)
  - Merge can be sequential, parallelism comes from the recursive subtasks

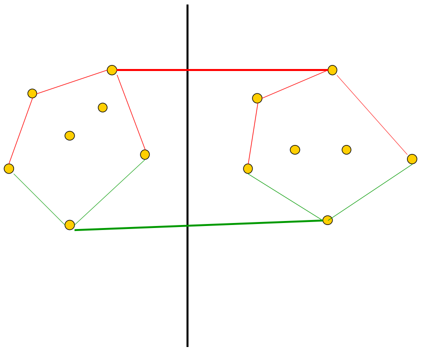
Complex Hull Example



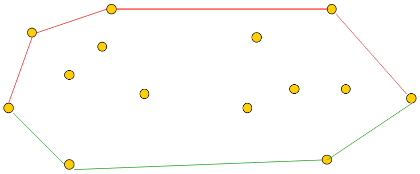
Complex Hull Example



Complex Hull Example



Complex Hull Example



Merge Operation

- Challenge:
  - Finding the upper and lower common tangents
  - Simple algorithm takes O(n)
  - We need a better algorithm

- Insight:
  - Resort to binary search
  - Consider the simpler problem of finding a tangent from a point to a polygon
  - Extend this to tangents from a polygon to another polygon
  - More details in Preparata and Shamos book on Computational Geometry (Lemma 3.1)