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 = $n \times \log n = O(n \log n)$
- Optimal parallel algorithm:
  - Total number of steps in parallel algorithm is equal to the number of steps in a sequential algorithm
  - Use $n/\log n$ 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 \( 0 \leq i, j \leq n-1 \).

**FAST-MAX(A):**

1. \( n \leftarrow \text{length}[A] \)
2. for \( i \leftarrow 0 \) to \( n-1 \), in parallel
   3. \( m[i] \leftarrow \text{true} \)
4. for \( i \leftarrow 0 \) to \( n-1 \) and \( j \leftarrow 0 \) to \( n-1 \), in parallel
   6. then \( m[i] \leftarrow \text{false} \)
7. for \( i \leftarrow 0 \) to \( n-1 \), in parallel
   8. if \( m[i] = \text{true} \)
   9. then \( \text{max} \leftarrow A[i] \)
10. return \( \text{max} \)

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](tree_diagram)

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

![Add-reduction](tree_diagram)
Scan (or Parallel prefix)

- What if you want to compute partial sums
- Definition: the parallel prefix operation takes a binary associative operator \( \oplus \), and an array of \( n \) elements
  \[ a_0, a_1, a_2, \ldots a_{n-1} \]
  and produces the array
  \[ a_0, (a_0 \oplus a_1), (a_0 \oplus a_1 \oplus a_2), \ldots (a_0 \oplus a_1 \oplus \cdots \oplus a_{n-1}) \]
- 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

![Diagram of Prefix Sum in Parallel](image)
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

Up sweep:
mine = left
tmp = left + right

Down sweep:
tmp = parent (root is 0)
right = tmp + mine

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
  \[1,0,1,1,0,0,1,\ldots\]
  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
  \[0,1,1,2,3,3,4,\ldots\]
  Gives the index of the i\textsuperscript{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

\[
F_{n+1} = F_n + F_{n-1}
\]

\[
\begin{pmatrix}
F_{n+1} \\
F_n
\end{pmatrix} =
\begin{pmatrix}
1 & 1 \\
1 & 0
\end{pmatrix}
\begin{pmatrix}
F_n \\
F_{n-1}
\end{pmatrix}
\]

Can compute all \( F_n \) by matmul_prefix on

\[
\begin{bmatrix}
(1 \ 1), (1 \ 1), (1 \ 1), (1 \ 1), (1 \ 1), (1 \ 1), (1 \ 1), (1 \ 1), (1 \ 1)
\end{bmatrix}
\]

then select the upper left entry

---

**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

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

- Serial algorithm: \( \Theta(n) \)
List ranking – EREW algorithm

- LIST-RANK(L) (in $O(\lg n)$ time)
  1. for each processor $i$, in parallel
  2. \hspace{1em} do if $\text{next}[i] = \text{nil}$
  3. \hspace{2em} then $d[i] \leftarrow 0$
  4. \hspace{2em} else $d[i] \leftarrow 1$
  5. \hspace{1em} while there exists an object $i$ such that $\text{next}[i] \neq \text{nil}$
  6. \hspace{1em} do for each processor $i$, in parallel
  7. \hspace{2em} do if $\text{next}[i] \neq \text{nil}$
  8. \hspace{3em} then $d[i] \leftarrow d[i] + d[\text{next}[i]]$
  9. \hspace{3em} next[i] $\leftarrow \text{next}[\text{next}[i]]$

List-ranking – EREW algorithm

(a) 3 4 6 1 0 5
(b) 2 4 6 1 0 5
(c) 4 6 2 1 0 5
(d) 5 4 3 2 1 0
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 $\oplus$, and an array of $n$ elements $[a_0, a_1, a_2, \ldots, a_{n-1}]$
  - and produces the array $[a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_2 \oplus \ldots \oplus a_{n-1})]$

```plaintext
Scan(a, n):
    if (n == 1) {  s[0] = a[0]; return s; }
    for (j = 0, x[0], y[0]; j < n/2, j++)
        x[j] = a[2*j] \oplus a[2*j+1];
        y[j] = Scan(x, n/2);
    for odd j in {0, 1, \ldots, n-1}
        s[j] = y[j/2];
    for even j in {0, 2, \ldots, n-1}
        s[j] = y[j/2] \oplus 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
  - Wj(n): number of operations the algorithm performs in step j
  - W(n) = Σ Wj(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 $S(n) = \text{poly-log}(n)$
- 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 = $n/\log n$

- Scalable up to $n/\log(n)$ processors
Segmented Operations

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

<table>
<thead>
<tr>
<th></th>
<th>(y, T)</th>
<th>(y, F)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x, T)</td>
<td>(x+y, T)</td>
<td>(y, F)</td>
</tr>
<tr>
<td>(x, F)</td>
<td>(y, T)</td>
<td>(x⊕y, F)</td>
</tr>
</tbody>
</table>

e.g. 1 2 3 4 5 6 7 8

<table>
<thead>
<tr>
<th>T</th>
<th>T</th>
<th>F</th>
<th>F</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result 1</td>
<td>3</td>
<td>3</td>
<td>7</td>
<td>12</td>
</tr>
</tbody>
</table>

Change of segment indicated by switching T/F

Parallel prefix on a list

- A prefix computation is defined as:
  - Input: <x₁, x₂, ..., xₙ>
  - Binary associative operation ⊗
  - Output: <y₁, y₂, ..., yₙ>
  - Such that:
    - y₁ = x₁
    - yₖ = yₖ₋₁ ⊗ xₖ for k = 2, 3, ..., n, i.e., yₖ = ⊗ x₁ ⊗ x₂ ... ⊗ xₖ.
- Suppose <x₁, x₂, ..., xₙ> are stored orderly in a list.
- Define notation: [i,j] = xᵢ ⊗ xᵢ₊₁ ... ⊗ xⱼ
Prefix computation

- \textsc{list-prefix}(L)
  1. \textbf{for} each processor \(i\), in parallel
  2. \textbf{do} \(y[i] \leftarrow x[i]\)
  3. \textbf{while} there exists an object \(i\) such that \(\text{prev}[i] \neq \text{nil}\)
     4. \textbf{do for} each processor \(i\), in parallel
     5. \textbf{do if} \(\text{prev}[i] \neq \text{nil}\)
     6. \textbf{then} \(y[\text{prev}[i]] \leftarrow y[i] \otimes y[\text{prev}[i]]\)
     7. \(\text{prev}[i] \leftarrow \text{prev}[\text{prev}[i]]\)

List Prefix Operations

- What is \(S(n)\)?
- What is \(W(n)\)?
- What is speedup on \(n/\log n\) 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

```
4 ← 3 ← 6 ← 7 ← 4 ← 3
```

```
4  7  9  13  11  7
```

```
4  7  13  20  20  20
```

```
4  7  13  20  24  27
```
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

Eliminate #1:

Eliminate #2:

Eliminate #3:

Randomized List Ranking

- Elimination step:
  - Each processor is assigned $O(\log n)$ elements
  - Processor $j$ is assigned elements $j \cdot \log n \ldots (j+1) \cdot \log n - 1$
  - 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 $\text{parent}[i]$.
- 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 $\text{root}[i]$.

Find-roots – CREW algorithm

- $\text{FIND-ROOTS}(F)$
  1. for each processor $i$, in parallel
  2. do if $\text{parent}[i] = \text{nil}$
  3. then $\text{root}[i] \leftarrow i$
  4. while there exist a node $i$ such that $\text{parent}[i] \neq \text{nil}$
  5. do for each processor $i$, in parallel
  6. do if $\text{parent}[i] \neq \text{nil}$
  7. then $\text{root}[i] \leftarrow \text{root}[\text{parent}[i]]$
  8. $\text{parent}[i] \leftarrow \text{parent}[\text{parent}[i]]$
**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 piece 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

Euler Tour Diagram
Using Euler Tours

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

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

Depth of tree vertices

- Input:
  - $L[i] = \text{position of incoming edge into } i \text{ in euler tour}$
  - $R[i] = \text{position of outgoing edge from } i \text{ in euler tour}$

```plaintext
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:
  - $W(n) = 2W(n/2) + \text{merge}_\text{cost}$
  - $S(n) = S(n/2) + \text{merge}_\text{cost}$
  - If $\text{merge}_\text{cost}$ 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
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)