This talk will present a number of basic parallel algorithms for the Connection Machine and discuss the implementation of several applications. We will begin with a brief description of the basic architecture of the machine, some programming language constructs, and some issues of programming style. We then present a number parallel data structures and the algorithms for their manipulation. While the discussion will be in the context of the Connection Machine, the basic principles should be applicable to any parallel architecture. We introduce logarithmically linked lists and discuss using them for basic spreading and combining operations as well as the fundamental parallel prefix operation. This is useful for sorting, parallel parsing, carry propagation, enumeration, and parallel dynamic processor allocation. Communication patterns for the prefix operation based on hypercubes, shuffle exchange networks, and binary trees will also be discussed. Recent extensions to trees which simplify many parallel graph algorithms will be introduced. Finally, application of these concepts to four applications will be presented. Region labelling of an image is accomplished by both random tree growth and random pointer hopping. Parallel dictionary lookup of free text may be done by sorting followed by a prefix operation. Three-dimensional surface representation with hidden surface removal may be accomplished using a single routing operation. Finally, we discuss the solution of large sparse linear systems using parallel nested dissection.
Parallel Programming on the Connection Machine
- Stephen Omohundro

1. The Connection Machine
2. *Lisp
3. Parallel Algorithms
4. Applications
   a. Region Labelling
   b. Dictionary Lookup
   c. 3-d graphics
   d. Large sparse linear systems
The Connection Machine Hardware

3 Communication Paths
1. Instruction stream - broadcast to all processors
2. Read/Write - individual memory locations
3. Global Combination - or bits together from each processor

Context Flag
Instructions are executed only in those processors with this flag set.

Routing
Each processor can send a message to any other processor.
**Lisp**

- structured language
- extension of Common Lisp, simple macros
- automatic stack maintenance & type coercion
- virtual processors

**Pvars:** "parallel variables"

- Lisp struct w/ location, length, type

**Operations:** `+!!`, `-!!`, `*!!`, `/!!`, `1-!!`, `!!`, `sqrt!!`, ...

(*set a (+!! b (+!! c d)))

**Predicates:** `=!!`, `<!!`, `>!!`, `<=!!`, ...

**Processor Selection:**

(*when (=!! a b)
 (*set c d))

**Temporary Pvar Allocation:**

(*let ((a (!! 5)))
 (*set b a))

**Routing:** `send-with-or`, `send-with-min`, `send-with-max`, `get!!`, ...

(*when test
 (*send a (!! self-pointer!!) b))

**Global Combination:** `or`, `and`, `min`, `max`, ...

(*when (=!! a (!! 0))
 (*max b))
Parallel Programming Issues

Data Abstraction
Serialization: stacks, queues, deques, priority queues, dictionaries, union-find
Implementation: data structures: linked lists, heaps, hash tables, 3-2 trees, finite state machines

Extra Parallel Issues:
- Independence of size and structure of machine
- Utilization of processors
- Ability to do many instances at once
- Randomness

Types of Parallel Problems: w/ N processors

No Communication: N trials in Monte Carlo, N initial conditions of S.O.
N rays in image, analysis of N experiment runs, database lookup

Local Communication: P.D.E. simulation, image processing, convolution,
circuit simulation, cellular automata

Regular Global Communication: FFT, Batchelor sort

Irregular Global Communication: what we're concerned with here

Types of Data Structure

Abstract Structures: only use pointers
dynamically allocatable, machine independent, fault tolerant
but: need processor allocation, garbage collection, complex routing

Machine Based Structures: fast but rigid, machine dependent
Combining & Spreading with Binary Trees

Spreading: copy information from one processor to several others

Combining: combine sets of data using a given binary operation

Binary fan-out and fan-in trees:

Disadvantages:
1. Uses $N-1$ extra processors
2. Takes $\lceil 2 \log N \rceil$ cycles
3. Each processor only sends twice or combines once
4. Have to maintain the balanced tree structure

Solutions to problem #1:
Use nodes as well as leaves: e.g., spread to first $N$ processes
or: Store nodes with leaves: (always $N-1$ nodes in $N$ leaf tree)
  e.g., store a node with the rightmost leaf of its left son:
Logarithmically Linked Lists

Really would like to double the number of processors containing the data on each iteration:

Similarly, in combining would like to halve the operands on each iteration:

These pointers all lie in a structure which extends a linked list with pointers 2 ahead, 4 ahead, 8 ahead, etc. Let us call this a logarithmically linked list.

requires $\log N$ pointers per processor
Log Linking Lists Using Pointer Hopping

Can reverse a linked list by sending forward self-addresses:

Send forward link along back link to get doubled list:

Gives 2 interleaved lists, repeat the same 2 steps:

Forms log linked list from linked list in $2gN^2$ sends.

Often don't store log links, rather generate them as needed.
Prefix - A Generalization of Spreading & Combining

Prefix multiplication: Takes a vector of elements and an associative binary operation and returns a vector of partial products. Known in APL as "scan".

\[ \oplus + (A \ B \ C \ D \ E \ F \ldots) \]

\[ \downarrow \]

\[ (A \ A \ B \ B \ C \ C \ D \ D \ C \ C \ D \ D \ E \ E \ \ldots) \]

Spreading: take \( A \oplus B = B \)

Combining: Get result from last element of prefix.

Enumeration: Often useful to number the elements of a set. Mark elements of set with 1, others with 0 and prefix \( + \), using result in set elements:

\[ (0 \ 1 \ 1 \ 0 \ 0 \ 0 \ 1 \ 0 \ 1 \ 1 \ 1 \ 0 \ 1 \ldots) \]

\[ \downarrow \]

\[ (0 \ 1 \ 2 \ 2 \ 2 \ 2 \ 3 \ 3 \ 4 \ 5 \ 6 \ 6 \ 7 \ldots) \]

\[ \downarrow \]

\[ (0 \ 1 \ 2 \ 0 \ 0 \ 0 \ 3 \ 0 \ 4 \ 5 \ 6 \ 0 \ 7 \ldots) \]
Log Linked Lists Naturally Implement Prefix

Iterate over links, sending over $2^{n-1}$ on the $n^{th}$ stage and combining:

$A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow F \rightarrow G \rightarrow H \rightarrow I \ldots$

$A \rightarrow AB \rightarrow BC \rightarrow CD \rightarrow DE \rightarrow EF \rightarrow FG \rightarrow GH \rightarrow HI \ldots$

$A \rightarrow AB \rightarrow ABC \rightarrow ABCD \rightarrow BCDE \rightarrow CDEF \rightarrow DBFG \rightarrow EFHI \rightarrow FGHI \ldots$

First $2^n$ are correct after $N$ stages, so takes $\log N$ iterations.

Can do over whole machine without storing pointers. Just send to the address $2^{n-1}$ greater than yours.
Parallel Processor Allocation using Prefix

**Input:** set of processors who want a proc \( \text{want} = 1 \)
set of free processors \( \text{free} = 1 \)

**Goal:** assign a free processor to each one that wants one.

**Rendezvous mechanism:**

1. Enumerate those with \( \text{want} = 1 \).
   \( \text{enum}_w \) 0 0 1 0 2 3 0 0 0 0 0 0 0 0 0 0 0 0 0

2. Workers send self-address to \( \text{enum}_w - 1 \).
   \( \text{addr}_w \) 2 4 5 1 0

3. Enumerate the free processors.
   \( \text{enum}_f \) 0 0 0 1 0 0 0 2 3 4 0 5

4. Free procs send self-address to \( \text{enum}_f - 1 \).
   \( \text{addr}_f \) 3 7 8 9 1 1

5. Procs which received both \( \text{addr}_w \) and \( \text{addr}_f \) send \( \text{addr}_f \) to addr
   \(--3--7--8--9--\)
One-dimensional Region Labelling with Prefix

Input: colored regions

Mark elements whose color differs from their left neighbor with a 1, others with a 0:

Prefix + to get labelled regions:
Spreading Values in Regions with Prefix

Input: A nil nil C nil A nil nil nil B nil nil D nil nil

Output: A A A C C A A A A A A B B B B D D D D

Binary Operation:

\[ I_1 \oplus I_2 = \begin{cases} I_1 & \text{if } I_2 = \text{nil} \\ I_2 & \text{else} \end{cases} \]

Associativity:

\[ I_1 \oplus (I_2 \oplus I_3) = I_1 \oplus (\begin{cases} I_2 & \text{if } I_3 = \text{nil} \\ I_3 & \text{else} \end{cases}) \]

\[ = \begin{cases} I_1 \oplus I_2 & \text{if } I_3 = \text{nil} \\ I_1 \oplus I_3 & \text{else} \end{cases} \]

\[ = \begin{cases} I_1 \oplus (\begin{cases} I_2 & \text{if } I_3 = \text{nil} \\ I_3 & \text{else} \end{cases}) & \text{if } I_2 = \text{nil} \\ I_2 \oplus I_3 & \text{else} \end{cases} \]

\[ (I_1 \oplus I_2) \oplus I_3 = (\begin{cases} I_1 & \text{if } I_2 = \text{nil} \\ I_2 & \text{else} \end{cases}) \oplus I_3 \]

\[ = \begin{cases} I_1 \oplus I_3 & \text{if } I_2 = \text{nil} \\ I_2 \oplus I_3 & \text{else} \end{cases} \]
Regular Expression Recognition using Prefix

input: finite state machine & string

goal: determine if string is in language defined by DFA

even better: "lexing" - determine state of DFA reached at each character

idea: elements of list are mappings from DFA states to themselves corresponding to the characters of the string.

Prefix composition of mappings and evaluate on starting state to get state at each character.
Pairing Linked Lists with Pointer Hopping

**input**: set of pairs of linked lists, heads containing pointers to each other

**goal**: to link corresponding elements in the list pairs.

This is analogous to the Common Lisp `pairlis` function.

**pointer hopping algorithm**: Iterate over n. On n-th stage, paired list elements send pointer $2^n$ ahead in list to paired element. The received pointers are sent ahead $2^n$ in each list. The $2^{n+1}$ pointers are obtained by hopping.
Bignum Addition using Pairing + Prefix

Input: pairs of linked lists of 1's + 0's representing big binary numbers.
Goal: linked lists of 1's + 0's representing the sums.

Algorithm:
1. Use pointer hopping pairing to get summands into a single list.
   \[ S_1 = 1010110 \]
   \[ S_2 = 1100100 \]

2. Make generate & propagate bits using:
   \[ P = S_1 \lor S_2 \]
   \[ G = S_1 \land S_2 \]
   \[ P = 1110110 \]
   \[ G = 1000100 \]

3. Prefix the operators:
   \[ \left( \begin{array}{c} P_1 \\ G_1 \end{array} \right) \oplus \left( \begin{array}{c} P_2 \\ G_2 \end{array} \right) = \left( \begin{array}{c} P_1 \land P_2 \\ G_2 \lor (G_1 \land P_2) \end{array} \right) \]
   \[ P = 1110000 \]
   \[ G = 1110110 \]

4. Let carry \( C \) be 0 in the next lower processor.
   \[ C = 0111011 \]

5. Sum bit is:
   \[ (C \land \overline{S}_1 \land \overline{S}_2) \lor (C \land S_1 \land \overline{S}_2) \lor (C \land \overline{S}_1 \land S_2) \lor (C \land S_1 \land S_2) \]
   \[ \text{Sum} = 0001001 = 72 \]

Associativity:
\[ \left( \begin{array}{c} p_1 \\ g_1 \end{array} \right) \oplus \left( \begin{array}{c} p_2 \\ g_2 \end{array} \right) = \left( \begin{array}{c} p_1 \land p_2 \\ g_2 \lor (g_1 \land p_2) \end{array} \right) \]
\[ \left( \begin{array}{c} p_1 \\ g_1 \end{array} \right) \oplus \left( \begin{array}{c} p_2 \\ g_2 \end{array} \right) = \left( \begin{array}{c} p_1 \land p_3 \\ g_2 \lor (g_1 \land p_3) \end{array} \right) \]
\[ \left( \begin{array}{c} p_1 \\ g_1 \end{array} \right) \oplus \left( \begin{array}{c} p_2 \\ g_2 \end{array} \right) = \left( \begin{array}{c} p_1 \land p_3 \\ g_2 \lor (g_1 \land p_3) \end{array} \right) \]
\[ \left( \begin{array}{c} p_1 \\ g_1 \end{array} \right) \oplus \left( \begin{array}{c} p_2 \\ g_2 \end{array} \right) = \left( \begin{array}{c} p_1 \land p_3 \\ g_2 \lor (g_1 \land p_3) \end{array} \right) \]
\[ \left( \begin{array}{c} p_1 \\ g_1 \end{array} \right) \oplus \left( \begin{array}{c} p_2 \\ g_2 \end{array} \right) = \left( \begin{array}{c} p_1 \land p_3 \\ g_2 \lor (g_1 \land p_3) \end{array} \right) \]
\[ \left( \begin{array}{c} p_1 \\ g_1 \end{array} \right) \oplus \left( \begin{array}{c} p_2 \\ g_2 \end{array} \right) = \left( \begin{array}{c} p_1 \land p_3 \\ g_2 \lor (g_1 \land p_3) \end{array} \right) \]
preorder Combining, + Pre-, Post-, & In-order numbering of binary trees using prefix

asic idea:

Spreading: root initializes pre node, prefix A\oplus B = A
others irrelevant

Combining:

Each node initializes its pre, post, or in vertex to its value + the others to id.
Prefix of a binary operation will leave the total in the root's post vertex

Numbering:

Each node initializes its pre, post, or in vertex to 1 if the others to 0.
Prefix + over list and read result from pre, post or in vertex

tingier degree trees:
Radix Sort of the whole machine using prefix

Stable sort on 1 bit:

bit: 0 1 1 0 0 0 1 1

1. \( E_0 \leftarrow \) enumeration of processors with bit = 0
   \[ E_0: 1 \quad 2 \quad 3 \quad 4 \]

2. \( E \leftarrow \) broadcast of (max \( E_0 \) in set with bit = 0)
   \[ E: 4 \quad 4 \quad 4 \quad 4 \quad 4 \quad 4 \quad 4 \]

3. \( E_1 \leftarrow \) enumeration of processors with bit = 1
   \[ E_1: 1 \quad 2 \quad 3 \quad 4 \]

4. If bit = 0 send data to \( E_0 - 1 \) else to \( E + E_1 - 1 \):
   \[ 0 \quad 4 \quad 5 \quad 1 \quad 2 \quad 3 \quad 6 \quad 7 \]

Stable sort on n bit key:

Make \( n \) passes, begin by sorting on LSB of key
end by sorting on MSB.

Other sorts: Batcher's sort uses hypercube \((\log n)^2\)
Flashsort - like quicksort - probabilistic \( \log n \)
Selection
Radix sort is good for small keys \( \sim \log n \)
Linearly ordering Linked Lists using prefix

Given: A collection of linked lists.

Goal: The lists lined out in order.

Method: 1. \( e \leftarrow \text{enumeration of lists} \)
\[ 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 10 \rightarrow 11 \rightarrow 12 \rightarrow 13 \rightarrow 14 \rightarrow 15 \rightarrow 16 \rightarrow 17 \rightarrow 18 \]

2. \( p \leftarrow \text{prefix + over tails} \)
\[ 12 \rightarrow 16 \rightarrow 19 \]

3. \( t \leftarrow p - e \text{ in tails} \)

4. \( \text{spread } t \text{ over lists} \)
\[ 0 \rightarrow 6 \rightarrow 6 \rightarrow 6 \rightarrow 6 \rightarrow 12 \rightarrow 12 \rightarrow 12 \rightarrow 16 \]

5. \( \text{send to } y + e - 1 \)
\[ 0 \rightarrow 2 \rightarrow 6 \rightarrow 10 \rightarrow 14 \rightarrow 18 \rightarrow 18 \]

Example uses: 1. To simultaneously sort many linked lists, linearly order them, append \( t \) to front of key and sort.
2. Grow balanced binary trees over many lists.
3. Use hypercube prefix on linked lists.
Prefix on a Hypercube

The n'th hypercube dimension connects processors whose addresses differ only in the n'th bit.

```
   0  1  2  3  4  5  6  7  8  9  10  11  12  13  14  15

dim 0

dim 1

dim 2

dim 3
```

Prefix: 2 variables in each proc:  T (for total) + P (for prefix) initialized to 0. Iterate over hypercube dimensions, swapping T and updating T in both upper & lower subcubes and P in upper subcube.

```

T:  A  B  C  D  E  F  G  H
P:  A  B  C  D  E  F  G  H

T:  AB  AB  CD  CD  EF  EF  GH  GH
P:  A  AB  C  CD  E  EF  G  H

T:  ABCD  ABCD  ABCD  ABCD  EF  GH  EF  GH
P:  A  AB  ABC  ABCD  E  EF  G  H

T:  ABCDEF  ABCDEF  ABCDEF  ABCDEF  ABCDEF  ABCDEF  ABCDEF  ABCDEF
P:  A  AB  ABC  ABCD  ABCDE  ABCDEF  ABCDEFG
```

Since both T and P must be updated, there are \(2 \times \log_2 N\) evaluations of the binary function.
Prefix on a Shuffle-Exchange Graph

A shuffle connects a processor to one whose address bits are rotated 1 to the right.

Exchange links connect a processor to its predecessor and successor.

Prefix:

- **Start:** a b c d e f g h
- **Odds & Evens:** a b c d e f g h
- **Shuffle:** a b c d e f g

Recursively prefix:

- **First half:** a b c ab cd e f gh
- **Unshuffle:** a b c d ab cd e f gh
- **Odds & Evens:** a b c ab cd ed ab cd ef gh

Induction: if we can prefix the smaller list, we can prefix the larger one that formed it.

Gives binary tree:
Prefix of the Leaves of a Binary Tree

each node has 2 variables: prev and tree

final state: prev holds sum of all leaves to the left of node
tree holds sum of all leaves below node

2 phases: 1. Moving up the tree, tree gets sum of 2 children's tree
     2. Moving down the tree, root's prev gets id (identity).
     left child's prev ← father's prev
     right child's prev ← father's prev ⊕ brother's tree

Finally: Prefix is sum of prev and tree at the leaves.
**Parallel Random Pairing**

Each processor flips a coin to choose r or l. A link is chosen only if it is chosen by its two ends.

\[ p = 0.25 \] \[ \text{depth} = \lceil 2.4 \log N \rceil \]

**Parallel Random Pairing with Neighbor checking**

Most common unused link: \[ \ldots \ldots \] Let chosen links check their right 3 neighbors and make

A link is added in the cases:

- rllll, rlllr, rllrr, rllrn

This is 4 out of \( 2^5 = 32 \), so prob a link is added = \( \frac{4}{32} = \frac{1}{8} \)

So \[ p = \frac{1}{4} + \frac{1}{8} = \frac{3}{8} = 0.37 \] \[ \text{depth} = 1.5 \log N \]

**Deterministic Parallel Pairing**

Each processor gets a word made of its address followed by its complement. Given any two processors, there are bit positions where they have 0,1 and 1,0. Choose a pairing by going down the bits of this word as the coin flip, taking links when legal. Gives the serial selection depth if nodes sit randomly in the machine.
Growing Binary Trees over Linked Lists

Given: \( N \) leaves to repeatedly pair into tree

If on each stage we pair the nodes of fraction \( p \) of the links, then depth \( d \) bounded by:

\[
(1-p)^d N \leq 1
\]

\[
\Rightarrow d = \frac{\log(1-p)}{\log N}
\]

Balanced tree: \( p = 0.5 \)

\[
\text{depth} = \lceil \log N \rceil
\]

Sequential Random Pairing:

Worst case: \( p = 0.33 \)

\[
\text{depth} = \lceil 1.66 \log N \rceil
\]

Average case:

\[
\begin{align*}
\text{\# of pairings beginning at node} & : n(N) = 2(n(N-1)) \\
\text{\# of pairings beginning at link} & : l(N) = l(N-2) + n(N-2) = l(N-2) + l(N-3)
\end{align*}
\]

so

\[
\begin{pmatrix}
l(N) \\
l(N-1)
\end{pmatrix} = \begin{pmatrix}
0 & 1 \\
0 & 0 & 1
\end{pmatrix} \begin{pmatrix}
l(N-2) \\
l(N-3)
\end{pmatrix}
\]

The largest eigenvalue is the largest root of \( (-\lambda^3 + \lambda + 1 = 0) = 1.325 \)

asymptotically:

\[
l(N) = 0.41 (1.325)^N
\]

probability of a link after a link is \( p = \frac{\frac{3}{2} - 1}{(\frac{1}{3} - 1)^2} = 0.56 \)

average run has:

\[
1 \times 0.56 + 2 \times (0.56)^2 + 3 \times (0.56)^3 + \cdots = \frac{\frac{3}{2}}{(\frac{1}{3} - 1)^2} = 2.9
\]

so

\[
p = \frac{2.9}{2 \times 2.9 + 1} = 0.42
\]

\[
\text{depth} = \lceil 1.27 \log N \rceil
\]
Treefix and Graph Algorithms
Two generalizations of prefix apply to binary trees:

- **Rootfix**: product from root to element
- **Leaffix**: product of subtree beneath element

Grow a \( \log N \) communication tree using random pairing:
- leaves: pick parent
- nodes with 1 child: pick parent or child with probability \( \frac{1}{2} \)
- nodes with 2 children: pick each child with probability \( \frac{1}{2} \)

---

**Minimal Spanning Tree** - based on Borůvka algorithm

- distinct weights, \( 2 \log \log N \) processors

- edges marked selected or not
- initially no selected edges
- at each step, selected edges form a forest of trees
- grow communication trees over selected edges and node pointers
- use rootfix to broadcast index of root to all nodes
- use leaffix and then rootfix to determine for each tree
  - one incident edge of smallest weight whose other ends in another
- select these edges & repeat till there aren't any.
Application: Region Labelling of Satellite Images

Goal: Assign unique labels to regions of contiguous pixels of the same color

\[
\begin{array}{ccc}
1 & 1 & 2 \\
0 & 0 & 2 \\
3 & 3 & 0 \\
4 & 3 & 5
\end{array}
\]

Client's Suggested Method:

Initially, each processor is labelled with its address.
Then repeat until labels don't change:
Each processor looks at neighbors of the same color and sets its label to the smaller of its or its neighbors.
Minimum label in a region spreads 1 pixel per step.
For NxN image, typically takes about O(N) steps.

Read cases can take O(N^2) eg, spiral

Serial Implementation: using union-find with path compression.
Tarjan showed N^2 unions take O(N^(1.5)) times inverse of the Ackerman function.
Tree Growing Region Labelling

Grow binary trees over regions.
Randomly pair neighboring trees of the same color.

Each leaf keeps its current top-of-tree and a list of unused links from original graph.
Leaves obtain top-of-tree across links and send them with overwrite to their top-of-tree - only one gets there.

Use random pairing to make tree

Top of tree broadcasts its address to all leaves.
Links with the same top-of-tree on both sides turn themselves off.
Until no links survive.
Random Pointer Hopping Region Labelling

Idea: each pixel has a pointer into its region. Pointers diffuse around and walk along each other, hopefully getting exponentially longer, making the diameter small. Then use label propagation.

Diffusion: split links within regions into matchings of underlying graph.

Start with only self-pointers. Diffuse by choosing a random subset of one of the matchings and hopping pointers along links.

Hopping: Repeat with another set of pointers. Walk the second set along the first set a few times. Freeze these pointers and repeat.

Useful for parallel garbage collection.
Application: Dictionary Lookup

Goal: Port of a system for automatic text indexing requires the part of speech of each word of text.

Original Method: 3 phases

1. Broadcast the 50 most common words and their part of speech
   eg. the, of, and, I, ...

2. The remaining words hash code themselves into the can address space and look them selves up. Perform several passes.

3. Definitions of the remaining words are sequentially looked up by the host and broadcast.

New Method:

Sort the dictionary & text together, dictionary entry < text entry
(carrying address along)

aardvark aardvark artful artful artful asked asked asked asked ...

noun nil adj nil nil verb nil nil nil

Use prefix to spread the definitions

aardvark aardvark artful artful artful asked asked asked asked ...

noun noun adj adj adj verb verb verb verb

Send definitions back to original processors.
**Application: 3-d Surface Display**

**Problem:** Output of stereo matching program is a graph of heights one point in each processor.

**Previous Solution:** Color map representing height.

**New Solution:** 3-d isosurface seen from an angle.

![Image of light direction](image)

given height: $h$ in each processor $x, y$
observer angle: $\theta$
light direction: $l_x, l_y, l_z$

1. Calculate slopes $h_x, h_y$: broadcast $\sin \theta + \cos \theta$

2. Calculate intensity (Cemberion) = $\frac{l_z - h_x l_x - h_y l_y}{\sqrt{1 + h_x^2 + h_y^2}}$

3. Calculate destination pixel: $x_{\text{dest}} = x$, $y_{\text{dest}} = \cos \theta y + \sin \theta h$

4. Send intensity to destination with max
   preparing priority $y_{\text{max}} - y$

**Improvements:**
- Interpolate first by solving Poisson equation iteratively
- Calculate extent of square in image and spread this distance after sending
- Smooth final image by convolution of Gaussian.
Solving Sparse Linear Systems with Parallel Nested Dissection

**Goal:** Solve $Ax = b$, where $A$ is an non-symmetric, positive definite, sparse matrix.

**Previous techniques:**
- Sequential, dense $A$: time $O(n^3)$
- Sequential, sparse $A$, nested dissection: time $O(n^{3/2})$
- Pan, Reif: $O(n^{3/2})$ processors, time $O(n \log^2 n)$
- Reif, Taylor: simulated rigid implementation for grid graph on hypercube

This version: $O(n \sqrt{n})$ time, $O(n \log n)$ processors for planar graphs. Works with any graph that has a good separator.

**Nested Dissection:** Inverse of a sparse matrix isn't sparse.

**Gaussian elimination vs. Adjacency graph:**

```
  a          b
  |          |
  |          |
  |          |
  |          |
  |          |
  a          b
```

```
  a
  |          |
  |          |
  |          |
  |          |
  |          |
  a
```

In general:

```
  a
  |          |
  |          |
  |          |
  |          |
  |          |
  a
```

**Separator tree - Kernighan-Lin**

```
  1
  |  |
  2 3
  |  |
  4
```

```
  1
  |  |
  2
  |  |
```

Eliminate variables in the order $1, 2, 3, 4$, each level in parallel.

**CM:**

1. Proceed upward w/ LU decomposition using systolic algorithm
2. Plug in $b$ at bottom, proceed upward solving $L^{-1} b$
3. Proceed downward solving for $x$. 

```
  a
  |          |
  |          |
  |          |
  |          |
  |          |
  a
```

```
  a
  |          |
  |          |
  |          |
  |          |
  |          |
  a
```