Parallel Programming on the Connection Machine -- Stephen M. Omohundro

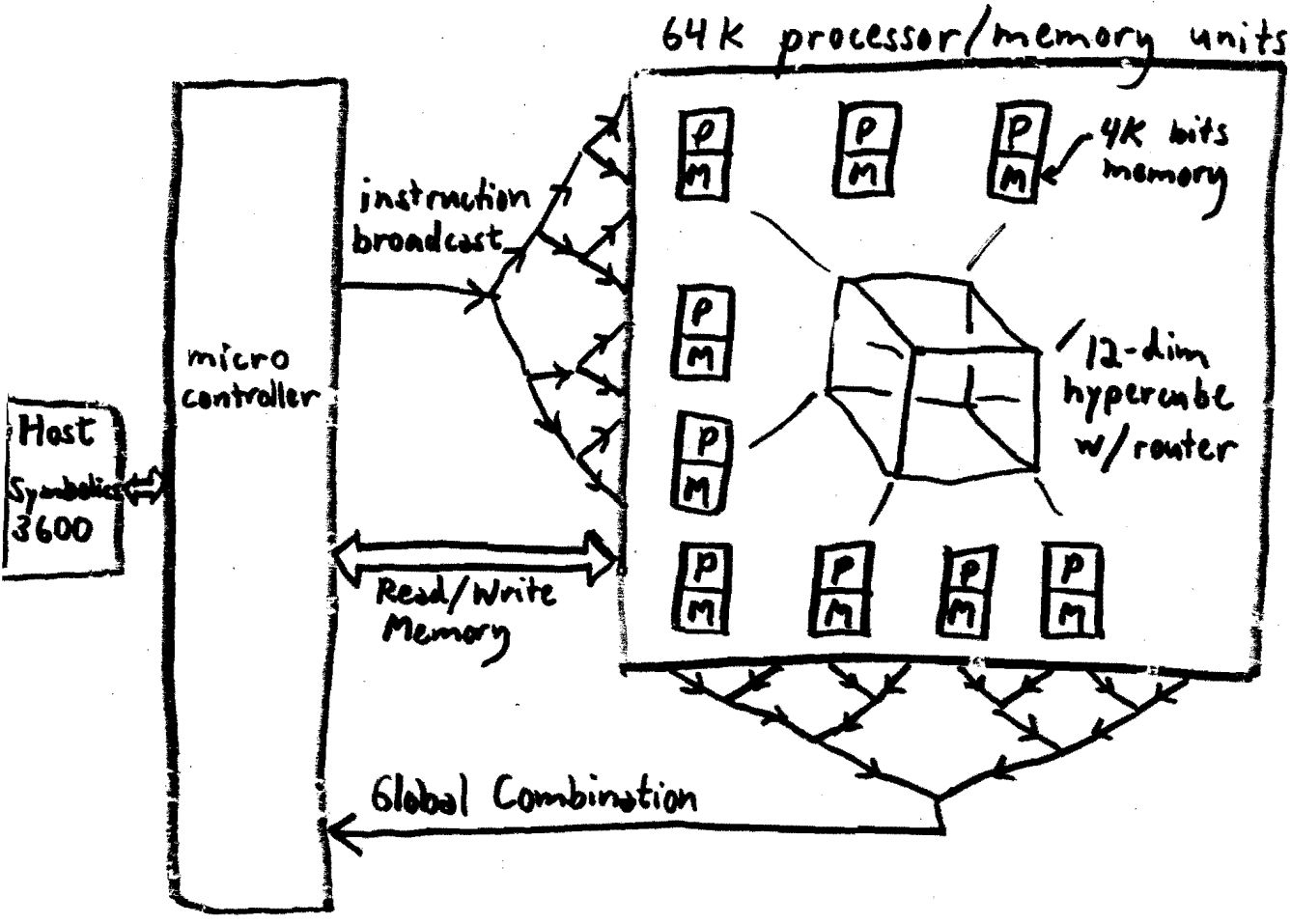
This talk will present a number of basic parallel algorithms for the Connection Machine and discuss the implementation of several applications. We will begin with a brief description of the basic architecture of the machine, some programming language constructs, and some issues of programming style. 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

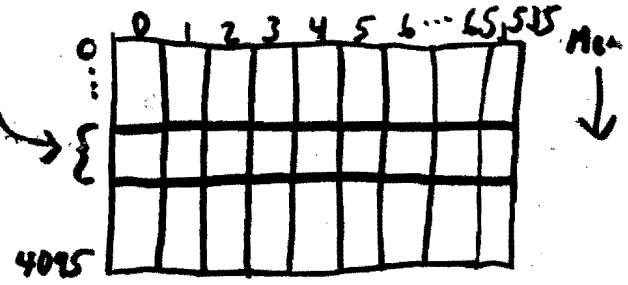
Each processor can send a message to any other processor.

w/ C. Lasser

*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 (1+!! self-pointer!!) b))
```

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

```
(*when (=!! a (!! 0))
       (*max b))
```

Parallel Programming Issues

Data Abstraction

serial programming abstractions: stacks, queues, deques, priority queues, dictionaries, union-find,
implementation data structures: linked lists, heaps, hash tables, 3-2 trees, finite state machine

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 O.D.
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, Batcher's sort

Irregular Global Communication what we're concerned with here

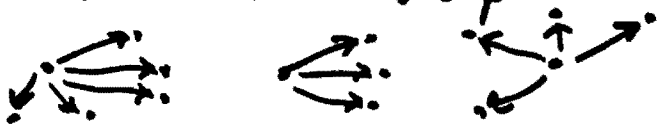
Types of Data Structure

Abstract Structures - only use pointers
dynamically alterable, 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 $\lceil 2 \lg N \rceil$ cycles
3. each processor only sends twice or combines once
4. have to maintain the balanced tree structure

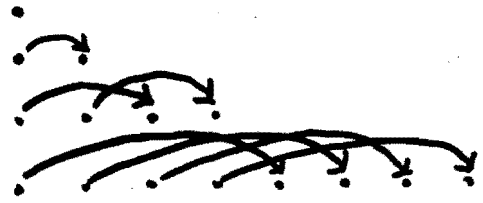
Solutions to problem #1:

Use nodes as well as leaves: eg. spread to first N procs
i is father of 2i and 2i+1

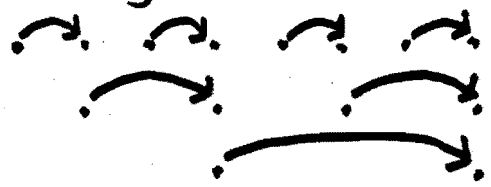
or: Store nodes with leaves: (always N-1 nodes in N leaf tree)
eg. store a node with the rightmost leaf of its left son:

Logarithmically Linked Lists

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

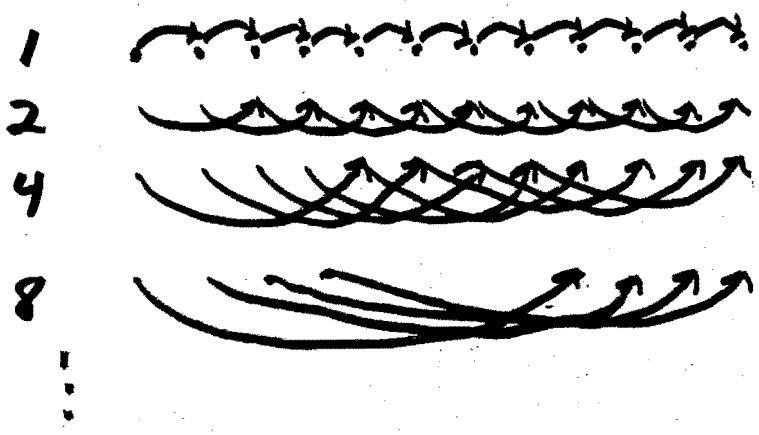


Similarly, in combining would like to halve the operands on each iteration:



These pointers all lie in a structure which extends a linked list with pointers 2 ahead, 4 ahead, 8 ahead, etc.

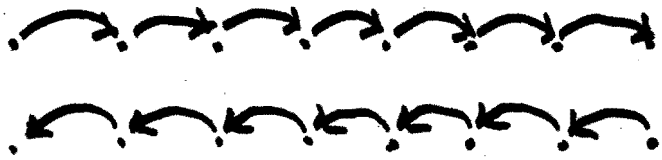
Let us call this a <u>logarithmically linked list.</u>



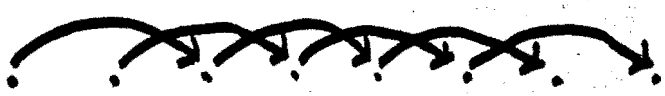
requires $\lceil \lg N \rceil$ pointers per processor

Log Linking Lists Using Pointer Hopping

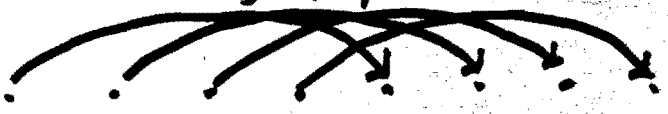
Can <u>reverse</u> a linked list by sending forward self-addresses:



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 $\lceil 2 \lg N \rceil - 2$ 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".

$$\oplus + (A \quad B \quad C \quad D \quad E \quad F \quad \ldots)$$

$$\Downarrow$$

$$(A \quad A\oplus B \quad A\oplus B\oplus C \quad A\oplus B\oplus C\oplus D \quad A\oplus B\oplus C\oplus D\oplus E \quad A\oplus B\oplus C\oplus D\oplus E\oplus F \ldots$$

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:

$$(0 \; 1 \; 1 \; 0 \; 0 \; 0 \; 1 \; 0 \; 1 \; 1 \; 1 \; 0 \; 1 \ldots)$$

$$\Downarrow$$

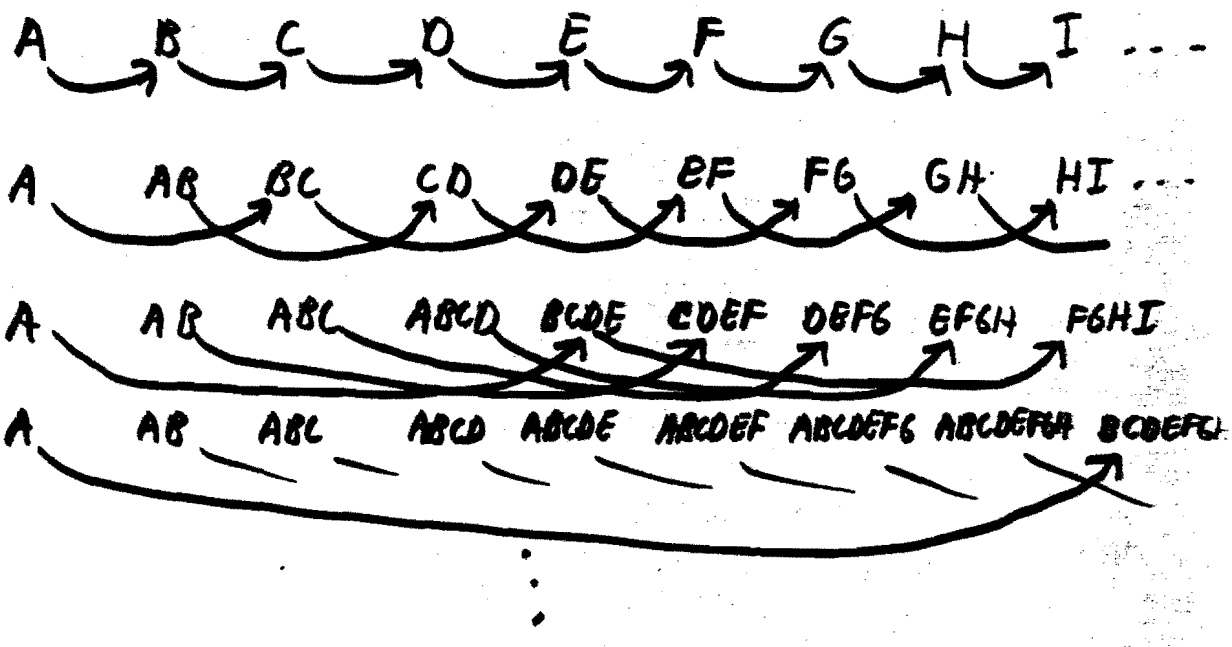
$$(0 \; 1 \; 2 \; 2 \; 2 \; 2 \; 3 \; 3 \; 4 \; 5 \; 6 \; 6 \; 7 \ldots)$$

$$\Downarrow$$

$$(0 \; 1 \; 2 \; 0 \; 0 \; 0 \; 3 \; 0 \; 4 \; 5 \; 6 \; 0 \; 7 \ldots)$$

Log Linked Lists Naturally Implement Prefix

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



First 2^N are correct after N stages, so takes $\lceil \lg N \rceil$ 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 want = 1
set of free processors free = 1

goal: assign a free processor to each one that wants one.

rendezvous mechanism:

want:	0	0	1	0	1	1	0	0	0	0	1	0
free:	0	0	0	1	0	0	0	1	1	1	0	1
address:	0	1	2	3	4	5	6	7	8	9	10	11

1. Enumerate those with want = 1.

$enum_w$	0	0	1	0	2	3	0	0	0	0	4	0

2. Wanters send self-address to $enum_w - 1$.

$addr_w$	2	4	5	10	-	-	-	-				

3. Enumerate the free processors.

$enum_f$	0	0	0	1	0	0	0	2	3	4	0	5

4. Free procs send self-address to $enum_f - 1$.

$addr_f$	3	7	8	9	11	-	-					

5. Procs which received both $addr_w$ and $addr_f$ send $addr_f$ to $addr$

		3		7	8					9		

One-dimensional Region Labelling with Prefix

Input: colored regions

r r r g g r r b b b b b g g g b b r r r r r r r r r g g g g b b . . .

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

1 0 0 1 0 1 0 1 0 0 0 0 1 0 0 1 0 1 0 1 0 0 0 0 0 0 0 1 0 0 0 1 0 0 1 0 . . .

Prefix + to get labelled regions:

1 1 1 2 2 3 3 4 4 4 4 4 5 5 5 6 6 7 7 7 7 7 7 7 8 8 8 8 9 9 . . .

Spreading Values in Regions with Prefix

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

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

Binary Operation:

$$I_1 \oplus I_2 = \text{if } I_2 = \text{nil then } I_1 \text{ else } I_2$$

Associativity:

$$I_1 \oplus (I_2 \oplus I_3) = I_1 \oplus (\text{If } I_3 = \text{nil then } I_2 \text{ else } I_3)$$

$$= \text{If } I_3 = \text{nil then } I_1 \oplus I_2 \text{ else } I_1 \oplus I_3$$

$$= \text{If } I_3 = \text{nil then (if } I_2 = \text{nil then } I_1 \text{ else } I_2) \text{ else } I_3$$

$$(I_1 \oplus I_2) \oplus I_3 = (\text{If } I_2 = \text{nil then } I_1 \text{ else } I_2) \oplus I_3$$

$$= \text{If } I_3 = \text{nil then (if } I_2 = \text{nil then } I_1 \text{ else } I_2) \text{ else } I_3$$

Regular Expression Recognition using Prefix

input: finite state machine & string

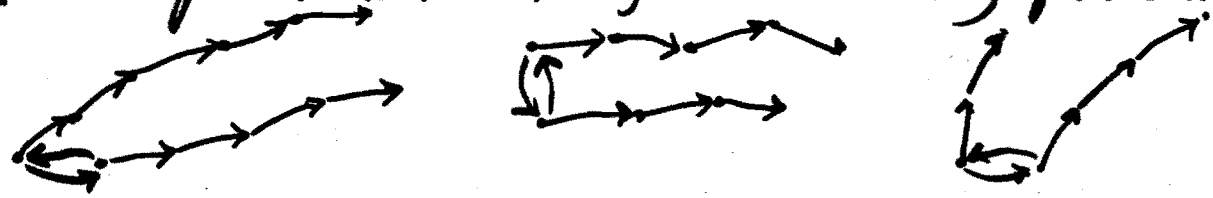
goal: determine if string is in language defined by DFA

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

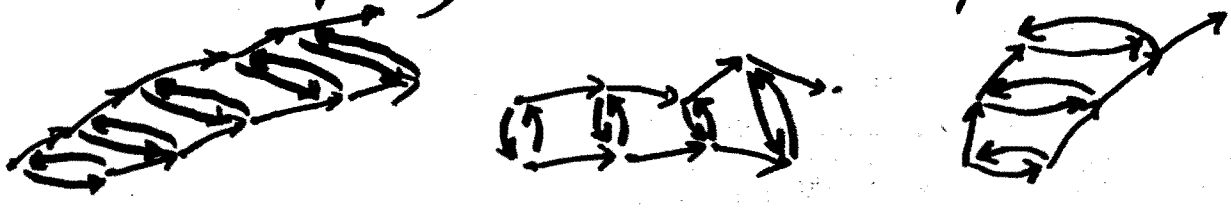
idea: elements of list are mappings from DFA states to themselves corresponding to the characters of the string. Prefix composition of mappings and evaluate on starting state to get state at each character.

Pairing Linked Lists with Pointer Hopping

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

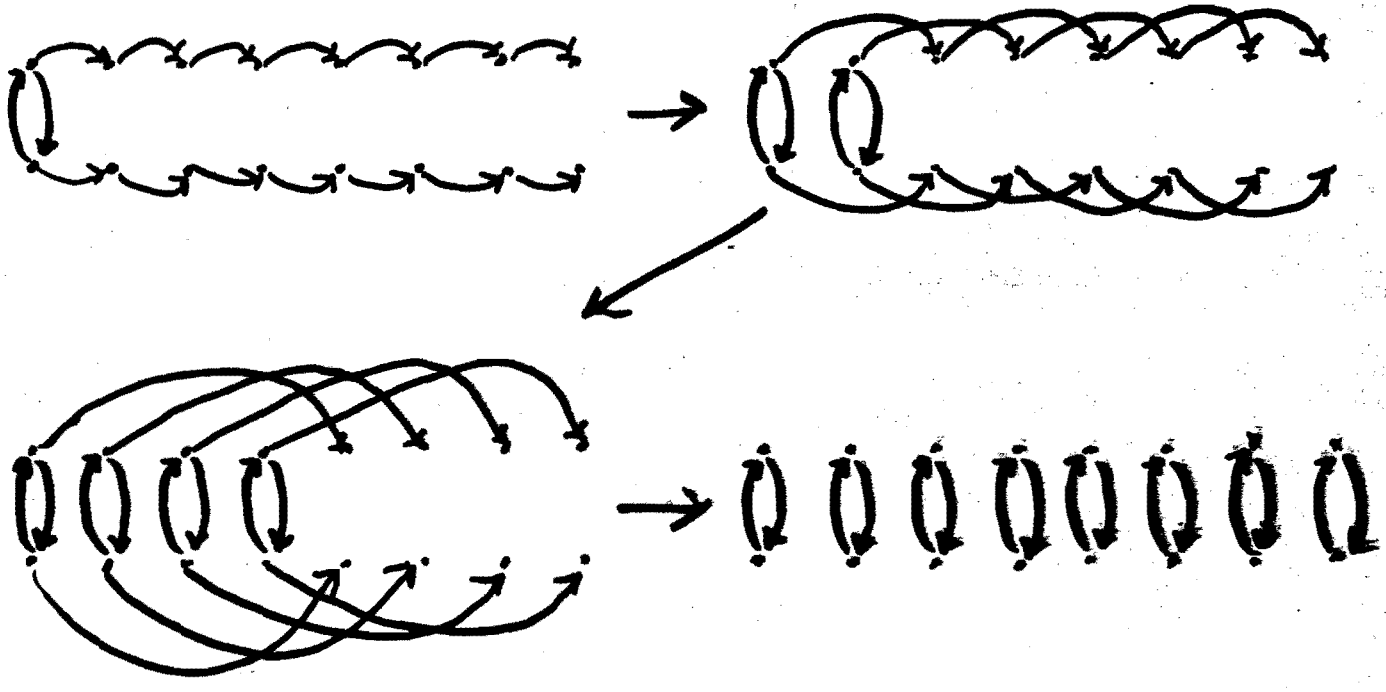


goal: to link corresponding elements in the list pairs.



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	1	0	1	0	1	1	0	= 53
S_2	1	1	0	0	1	0	0	= 19
LSB →	1	2	4	8	16	32	64	→ MSB

2. Make generate & propagate bits using: $P = S_1 \vee S_2$ $\quad G = S_1 \wedge S_2$

P	1	1	1	0	1	1	0
G	1	0	0	0	1	0	0

3. Prefix the operator: $\begin{pmatrix} P_1 \\ G_1 \end{pmatrix} \oplus \begin{pmatrix} P_2 \\ G_2 \end{pmatrix} = \begin{pmatrix} P_1 \wedge P_2 \\ G_2 \vee (G_1 \wedge P_2) \end{pmatrix}$

P	1	1	1	0	0	0	0
G	1	1	1	0	1	1	0

4. Let carry C be G in the next lower processor.

C	0	1	1	1	0	1	1

5. Sum bit is: $(C \wedge \bar{S_1} \wedge \bar{S_2}) \vee (\bar{C} \wedge \bar{S_1} \wedge S_2) \vee (\bar{C} \wedge S_1 \wedge \bar{S_2}) \vee (C \wedge S_1 \wedge S$

sum	0	0	0	1	0	0	1	= 72

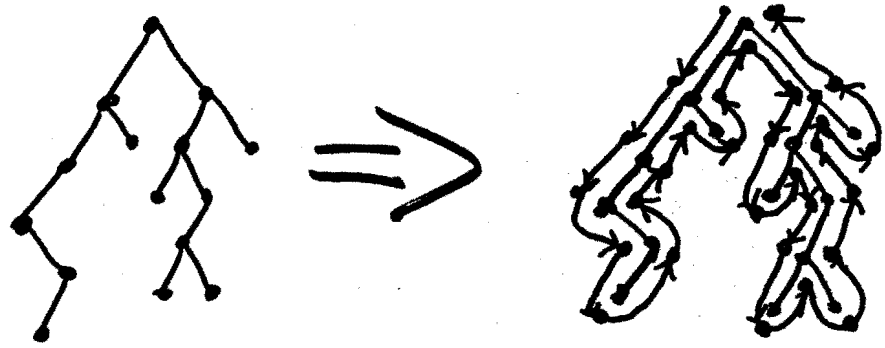
associativity:

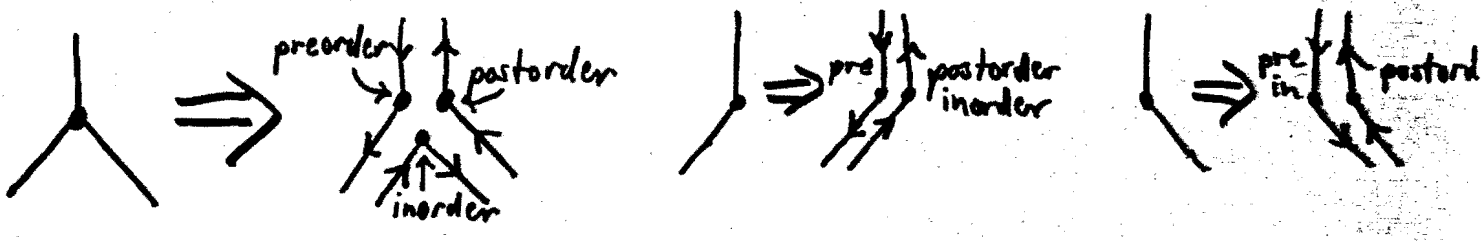
$$\left(\begin{pmatrix} P_1 \\ G_1 \end{pmatrix} \oplus \begin{pmatrix} P_2 \\ G_2 \end{pmatrix}\right) \oplus \begin{pmatrix} P_3 \\ G_3 \end{pmatrix} = \begin{pmatrix} P_1 \wedge P_2 \\ G_2 \vee (G_1 \wedge P_2) \end{pmatrix} \oplus \begin{pmatrix} P_3 \\ G_3 \end{pmatrix} = \begin{pmatrix} P_1 \wedge P_2 \wedge P_3 \\ G_3 \vee (P_3 \wedge (G_2 \vee (P_2 \wedge G_1))) \end{pmatrix}$$

$$\begin{pmatrix} P_1 \\ G_1 \end{pmatrix} \oplus \left(\begin{pmatrix} P_2 \\ G_2 \end{pmatrix} \oplus \begin{pmatrix} P_3 \\ G_3 \end{pmatrix}\right) = \begin{pmatrix} P_1 \\ G_1 \end{pmatrix} \oplus \begin{pmatrix} P_2 \wedge P_3 \\ G_2 \vee (P_3 \wedge G_2) \end{pmatrix} = \begin{pmatrix} P_1 \wedge P_2 \wedge P_3 \\ G_3 \vee (P_3 \wedge G_2) \vee ((P_2 \wedge P_3) \wedge G_1) \end{pmatrix}$$

Spreading, Combining, & Pre-, Post-, & In- order numbering of binary trees using prefix

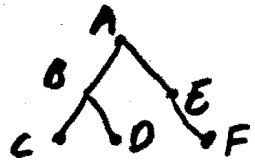
Basic idea:





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

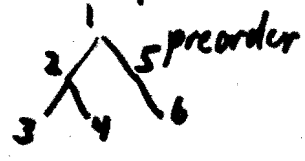
Combining:

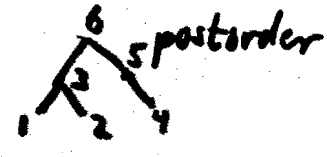


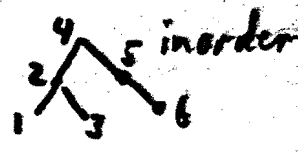
pre: ABCDEF
post: CDBFEA
in: CBDAEF

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

Higher 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 2 3 4

2. $\Sigma \leftarrow$ broadcast of (max E_0 in set with bit = 0)

   Σ: 4 4 4 4 4 4 4 4

3. $E_1 \leftarrow$ enumeration of processors with bit = 1

   E_1: 1 2 3 4

4. If bit = 0 send data to $E_0 - 1$ else to $\Sigma + E_1 - 1$:

   0 4 5 1 2 3 6 7

Stable sort on n bit key:

Make n passes, begin by sorting on LSB of key and by sorting on MSB.

Other sorts: Batcher's sort uses hypercube $(\lg n)^2$

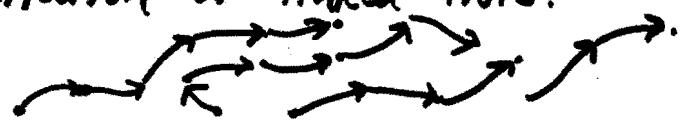
Flashsort - like quicksort - probabilistic $\lg n$

Selection

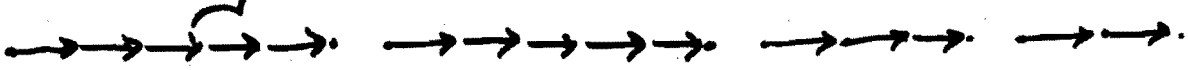
Radix sort is good for small keys $\sim \lg n$

Linearly ordering Linked Lists using prefix

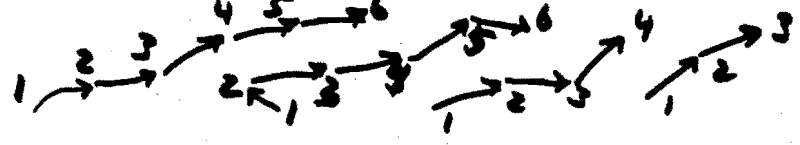
Given: A collection of linked lists.



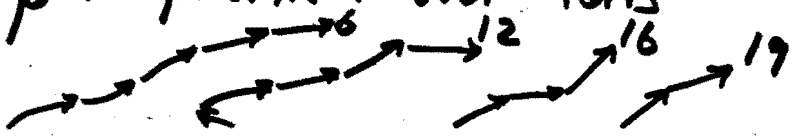
Goal: The lists layed out in order



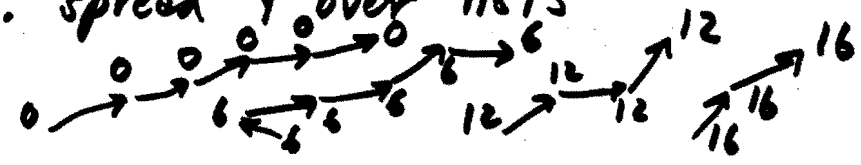
Method: 1. e ← enumeration of lists



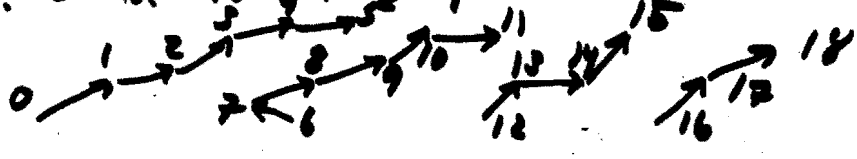
2. p ← prefix + over tails

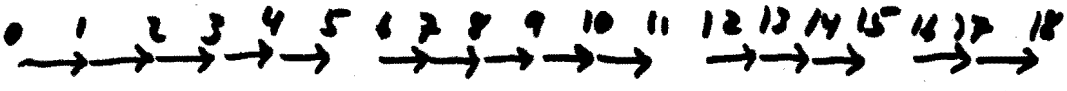


3. t ← p-e in tails
4. spread t over lists



5. send to t+e-1



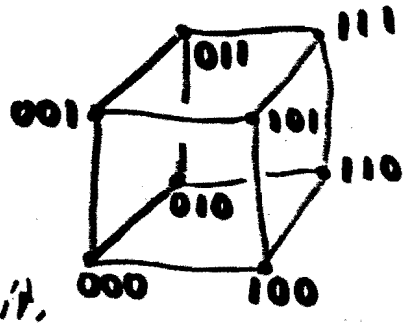


Example uses: 1. To simultaneously sort many linked lists, linearly order them, append t to front of key + sort.

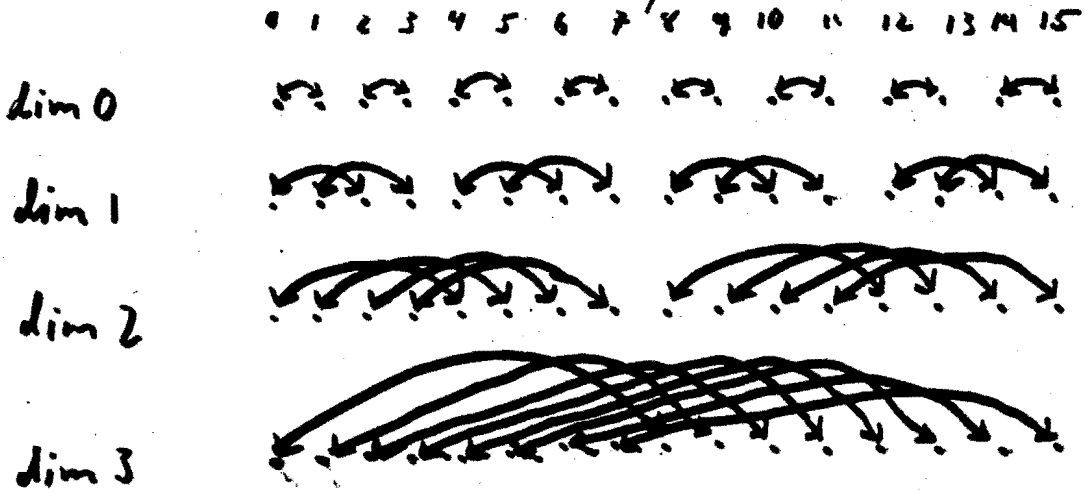
2. Grow balanced binary trees over many lists.

3. Use hypercube prefix on linked lists.

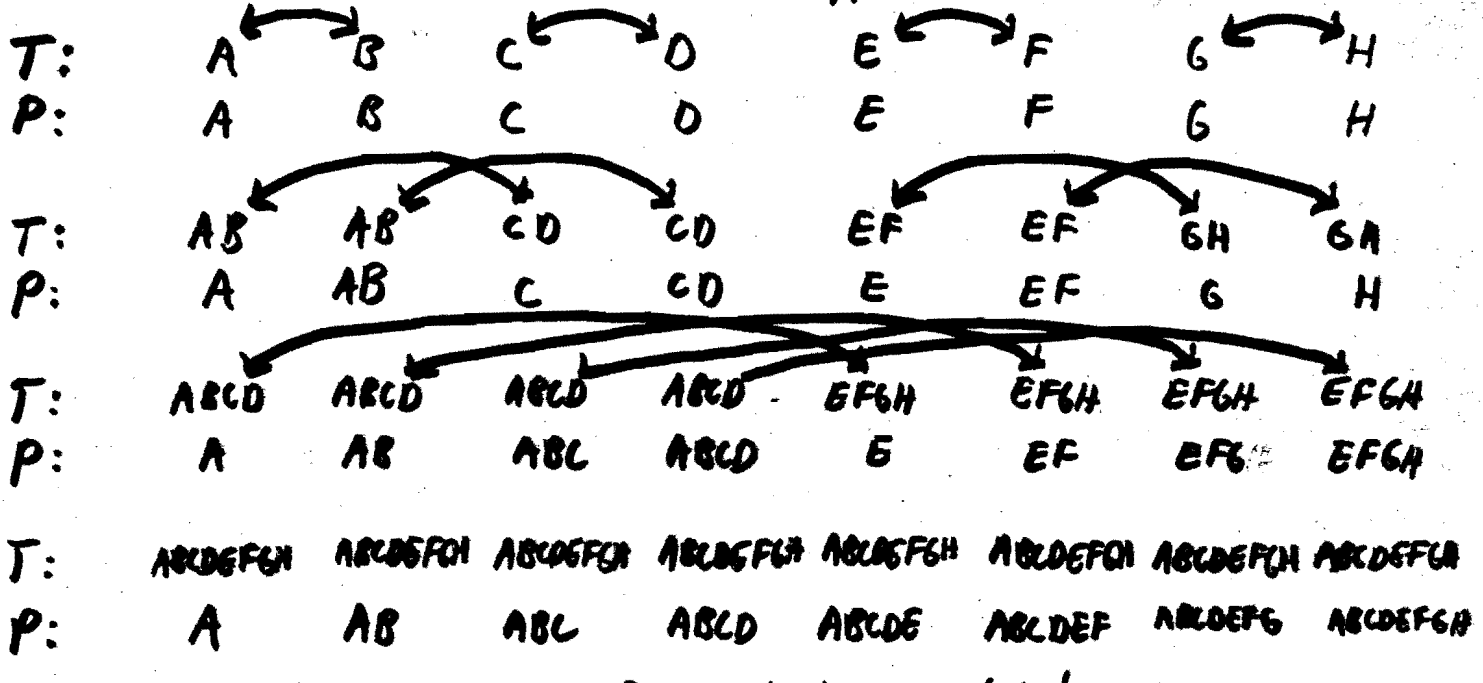
Prefix on a Hypercube



The n^{th} hypercube dimension connects processors whose addresses differ only in the nth bit.



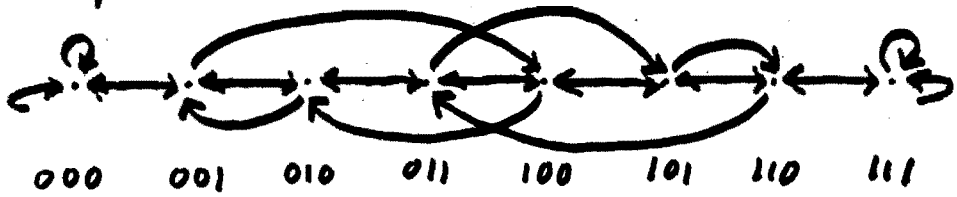
Prefix: 2 variables in each proc: T (for total) + P (for prefix) initialized to inp
Iterate over hypercube dimensions, swapping T and updating T in bot upper + lower subcubes and P in upper subcube.



Since both T and P must be updated, there are $\lceil 2 \lg N \rceil$ 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

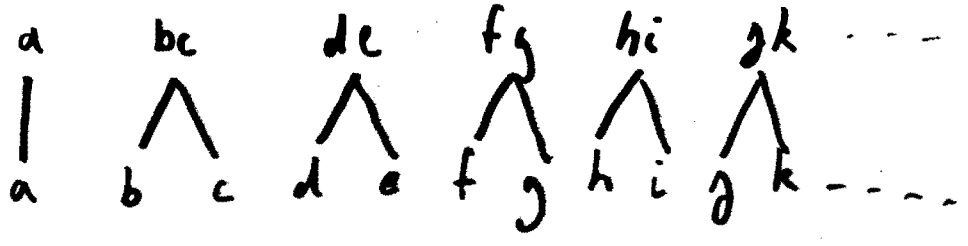
odds x evens: a bc de fg

shuffle: a bc de fg - - -

recursively prefix: first half: a abc abcde abcdefg - - -

unshuffle: a b abc d abcde f abcdefg h

evens x odds: a ab abc abcd abcde abcdef abcdefg abcdefgh



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

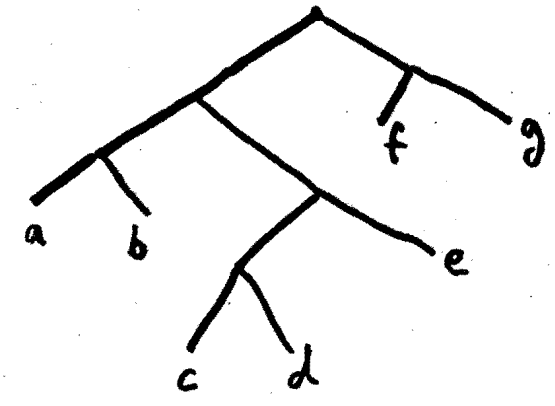
each node has 2 variables: prev + tree

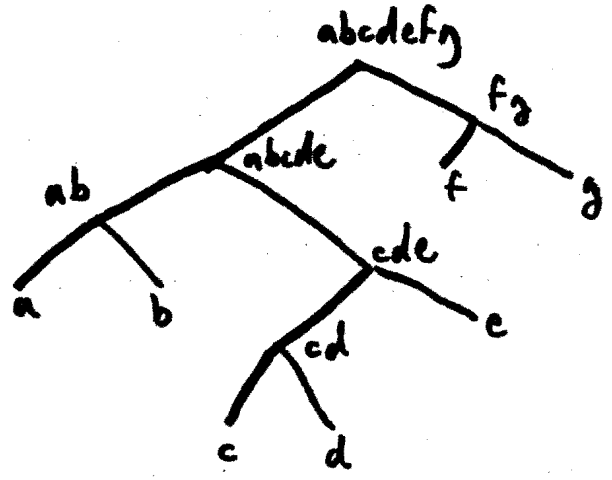
final state: prev holds sum of all leaves to the left of node
tree holds sum of all leaves below node

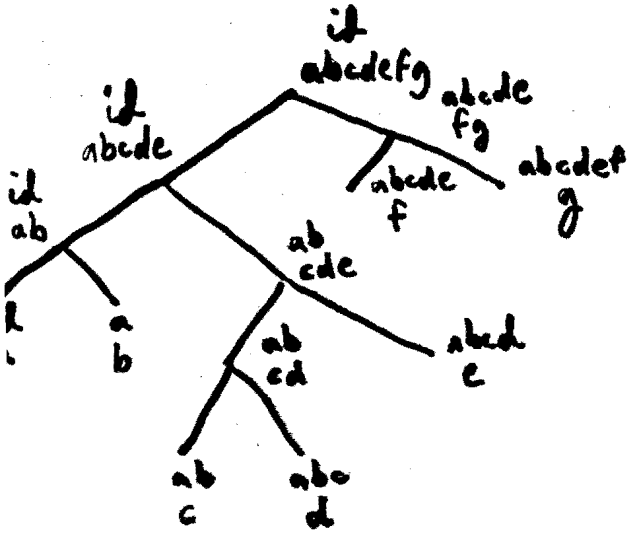
2 phases: 1. Moving up the tree, tree gets sum of 2 children's tree
2. Moving down the tree, root's prev gets id (identity).

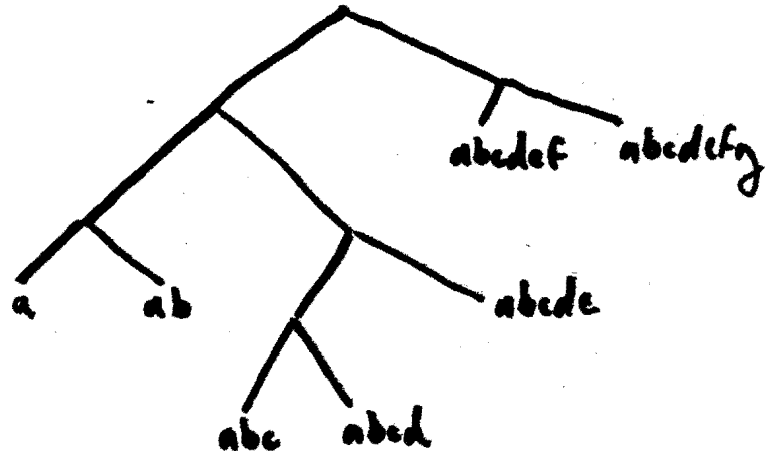
left child's prev ← father's prev
right child's prev ← father's prev ⊕ brother's tree

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.

—• • •— • •— $p = .25$ **depth = $\lceil 2.4 \lg N \rceil$**

Parallel Random Pairing with Neighbor checking

Most common unused link: —• •—

Let chosen links check their right 3 neighbors and make —•—•—•—

A link is added in the cases:

rllll, rlllr, rllrr, rlrrr

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

So $p = \frac{1}{4} + \frac{1}{8} = \frac{3}{8} = .37$ **depth = $1.5 \lg N$**

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.

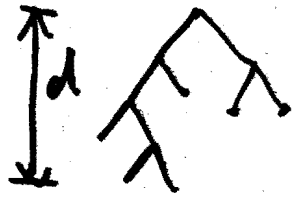
Growing Binary Trees over Linked Lists

Given: N leaves to repeatedly pair into tree →→→→→

If on each stage we pair the nodes of a fraction p of the links,

then depth d bounded by: $(1-p)^d N \leq 1$

$$\Rightarrow d \geq \frac{1}{\lg\left(\frac{1}{1-p}\right)} \lg N$$



Balanced tree: $p = .5$ $\quad \boxed{\text{depth} = \lceil \lg N \rceil}$ — • — • — • — • —

Sequential Random Pairing:

Worst case: $p = .33$ $\quad \boxed{\text{depth} = \lceil 1.66 \lg N \rceil}$ — • — • — • —

Average case: — • — • — • — • — • —

# of pairings beginning w/ node $n(N) = \ell(N-1)$

# of pairings beginning w/ link $\ell(N) = \ell(N-2) + n(N-2) = \ell(N-2) + \ell(N-3)$

$$\text{so} \quad \begin{pmatrix} \ell(N-2) \\ \ell(N-1) \\ \ell(N) \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} \ell(N-3) \\ \ell(N-2) \\ \ell(N-1) \end{pmatrix}$$

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

asymptotically: $\ell(N) = .41\,(1.325)^N$

probability of a link after a link is $p = \frac{1.3}{2.3} = .56$

average run has: $1 \times .56 + 2 \times (.56)^2 + 3 \times (.56)^3 + \ldots = \frac{(1/.56)}{(\frac{1}{.56} - 1)^2} = 2.9$

so $p = \frac{2.9}{2 \times 2.9 + 1} = .42$ $\quad \boxed{\text{depth} = \lceil 1.27 \lg N \rceil}$

C. Leiserson & B. Maggs

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 $\lg N$ communication tree using random pairing:

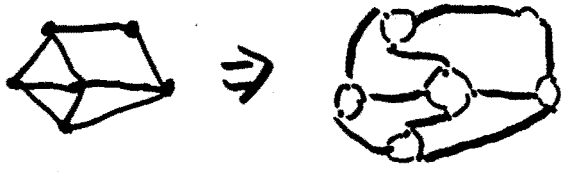
leaves: pick parent

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

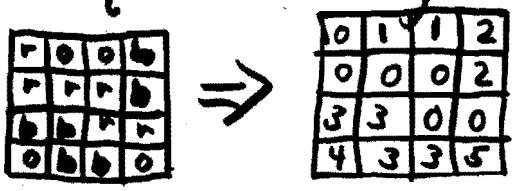
distinct weights, $2|E|$ processors



- 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 the 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 colo

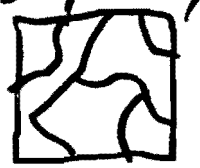


Client's Suggested Method:

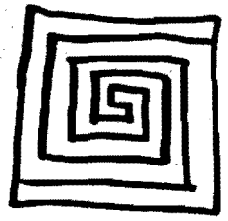
Initially, each processor is labelled with its address.
Then repeat until labels don't change:
Each processor looks at neighbors of the same color and sets its label to the smaller of it's or its neighbor's.

Minimum label in a region spreads 1 pixel per step.

For N x N image, typically takes about O(N) steps.



Bad cases can take $O(N^2)$ eg. spiral

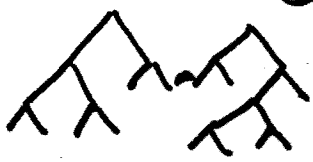


Serial Implementation: using union-find with path compression.
Tarjan showed N^2 unions take $O(N^2)$ times inverse of the Ackermann function.

Tree Growing Region Labelling

Grow binary trees over regions.

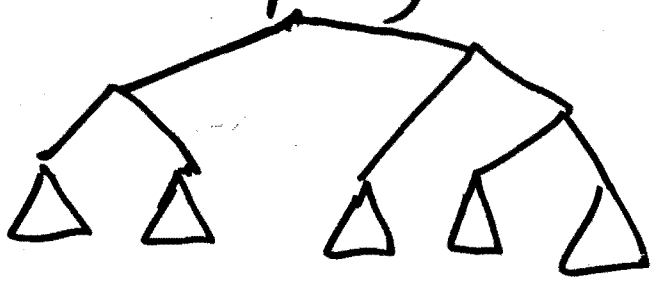
Randomly pair neighboring trees of the same color.



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.

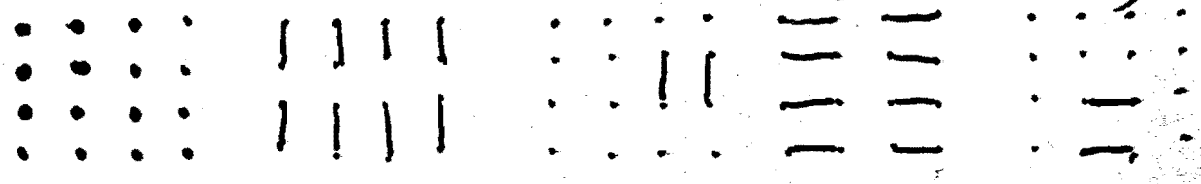
Until no links survive.

w/ C. Feynman

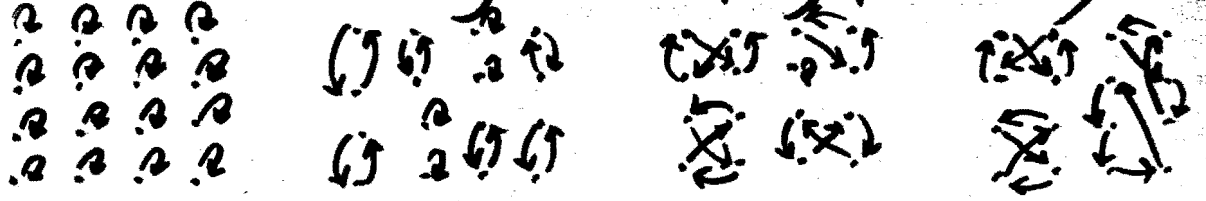
Random Pointer Hopping Region Labelling

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 grid



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

w/ G. Sabot

Application: Dictionary Lookup

Goal: Part 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 cm address space and look themselves 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)

aardwork	aardwork	artful	artful	artful	asked	asked	asked	asked	...
noun	nil	adj	nil	nil	verb	nil	nil	nil	

Use prefix to spread the definitions.

aardwork	aardwork	artful	artful	artful	asked	asked	asked	asked	...
noun	noun	adj	adj	adj	verb	verb	verb	verb	

Send definitions back to original processors.

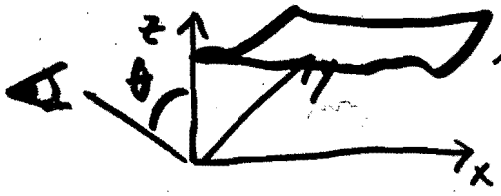
w/ M. Drumheller

Application: 3-d Surface Display

Problem: Output of stereo matching program is a graph of height, one value in each processor.

Previous Solution: Color map representing height.

New Solution: 3-d lit surface seen from an angle.



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

1. Calculate slopes h_x, h_y; broadcast $\sin\theta$ & $\cos\theta$
2. Calculate intensity (Lambertian) $= \dfrac{l_z - h_x l_x - h_y l_y}{\sqrt{1 + h_x^2 + h_y^2}}$
3. Calculate destination pixel: $x_{dest} = x$, $y_{dest} = \cos\theta\, y + \sin\theta\, h$
4. Send intensity to destination with max pending priority $y_{max} - y$

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 w/ Gaussian.

w/ C. Leiserson, J. Messina, L. Nathades, J. Reif, W. Taylor

Solving Sparse Linear Systems with Parallel Nested Dissection

Goal: Solve $Ax=b$, where A is an $n \times n$ symmetric, positive definite, sparse matrix

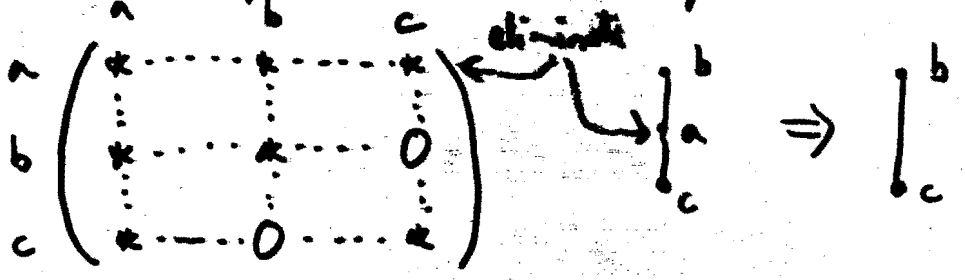
Previous techniques: sequential, dense $A \Rightarrow$ time $O(n^3)$

sequential, sparse A, nested dissection $\Rightarrow$ time $O(n^{3/2})$
Pan, Reif $\Rightarrow$ $\theta(n^{3/2})$ processors, time $O(\log^2 n)$
Reif, Taylor $\Rightarrow$ simulated rigid implementation for grid graph on hypercube

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

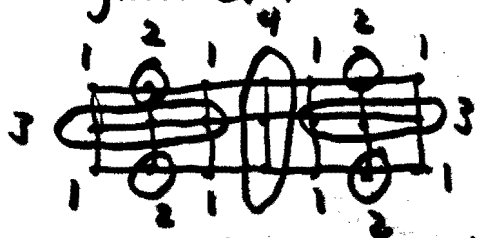
Nested Dissection: Inverse of a sparse matrix isn't sparse.

Gaussian elimination & Adjacency graph:

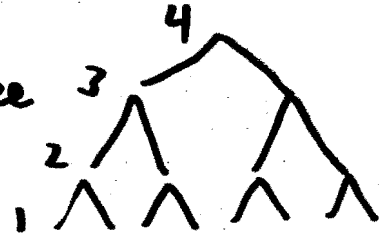


In general:

Separator tree – Kernighan-Lin



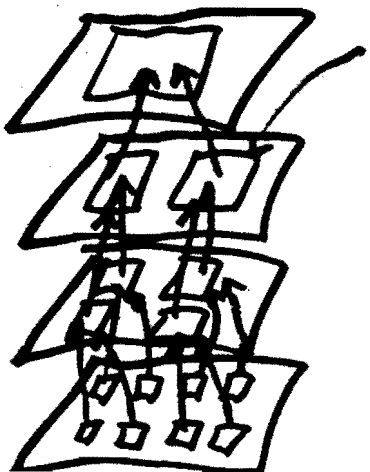
tree



eliminate variables in the order 1, 2, 3, 4, each level in parallel

matrices of vertices affected by eliminating each level

CM:



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