Blelloch 3

Goals: breadth-first search in parallel
and purely functional

Goals:

1) sequential semantics but with parallelism
(functional)

2) Cost model + provable bound

3) Work efficiency → probably more important
on lower # cores

4) low depth (span)

5) as simple as sequential

6) sequential as a side effect

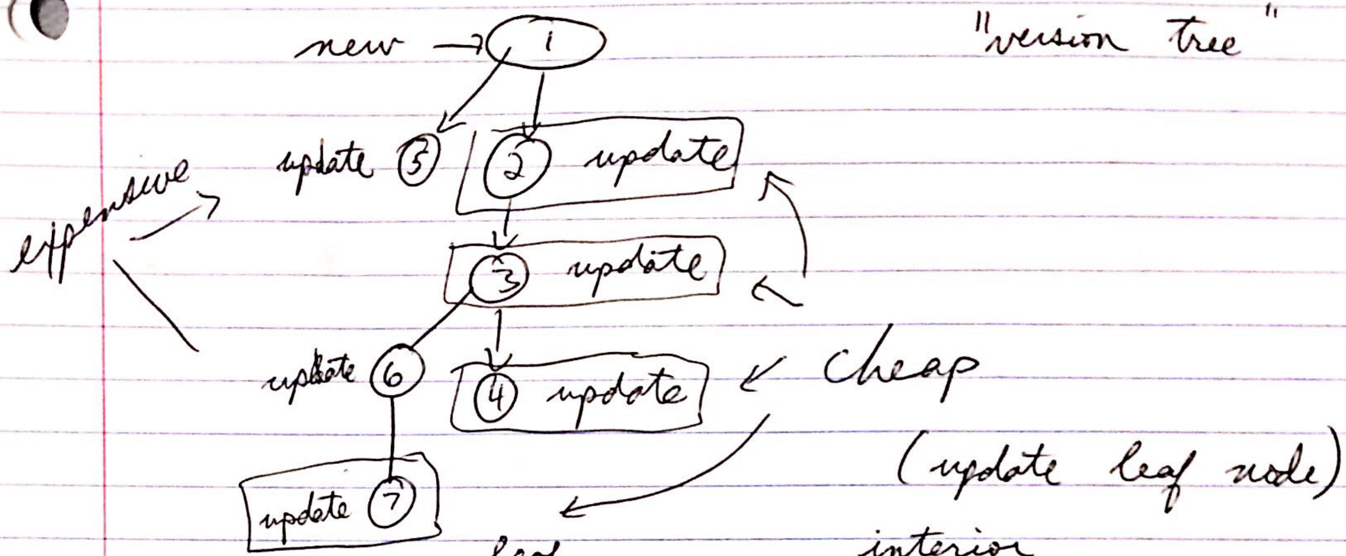
Problem with pure array:
have to copy array to make a change

1) persistence "requires" copying

except: there is only one instance in use
linear logic - single use
but doesn't work for parallelism
Popl 17 purely fn'l arrays
[language pure, cost model not pure, eg. Haskell]

2) keep a history of changes
disting. version leaf vs version interior node

new → ○



$n = \lvert A \rvert$	leaf W	leaf S	interior W	interior S
sub A	$O(1)$	$O(1)$	$O(\log n)$	$O(\log n)$
updateA	$O(1)$	$O(1)$	$O(n)$	$O(\log n)$
injectAU	$O(\lvert U \rvert)$	$O(1)$	$O(\lvert U \rvert + n)$	$O(\log n)$

Semantics

store $\quad \delta : \ell \to \{+, -\}$ — each version of array has label; + : leaf; − : interior node

big-step semantics (or small step)

$$\delta\ e \to \delta', v, w, d$$

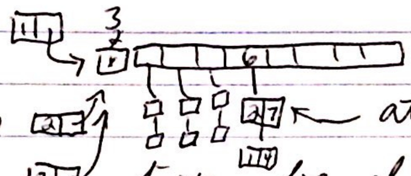
(v : return value, w : work, d : span/depth)

$\delta\ \ \text{update (set)}\ (A, u)$ — A : α seq., u : $\text{int} \times \alpha$ (update)

$$\frac{A = (a, \overset{\text{label}}{\ell}) \qquad \delta[\ell \to +] \qquad \text{new } \ell'}{\delta,\ \text{update}(A, v) \to \delta[\ell \to -,\ \ell' \to +],\ (\text{update}(a, v), \ell'), 1, 1}$$

$W: O(1)$
$S: O(1)$

ex.



update

at version 2, this was a 7

history log of what cell looked like previously

do update in place
store version history

if lookup with non-current version,
find cell ~~version~~ contents in history w/ matching version

if update old version, copy contents of array

— amount of space never more than double
if history is larger than array,
then copy

never waste more than factor of 2
constant space

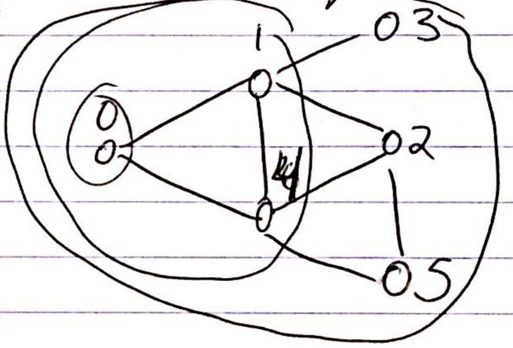
amortize cost of copy
by keeping history

can do same trick w/ inject

tricky: $e_1 \parallel e_2$

- both could update
- need to account for this, but it can be done

$\boxed{\text{BFS}}$ Breadth-first search



step 1: 0
step 2: 1, ~~4~~
step 3: 3, 2, 5

onion algorithm
1-away
2-away
3-away
etc.

common, eg. Facebook bounded BFS

use queue to do BFS sequentially

better to consider BFS as searching each level in parallel

iterative where each iteration covers whole level in parallel

$W = O(m+n)$
$S = O(d \cdot \log n)$

$m = |\text{Edges}|$
$n = |\text{vertices}|$
$d =$ diameter of graph
longest shortest path in graph
people ~ 7
web graph ~ 40

Repeat over levels:

keep track of:
F frontier - list from prev level
F , P

P: tree: every vertex point to parent
choose arbitrary when ambiguous

1) find all neighbors of F

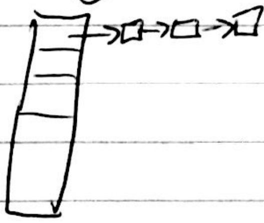
2) filter those not already visited

3) update P
3) remove duplicates

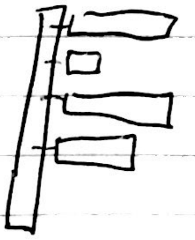
4)

Graph representation

adjacency list, horrible for parallelism ~~lists~~



use sequence of sequences
could use trees



ex graph

$G = \langle \langle 1,4 \rangle, \langle 0,2,3,4 \rangle, \langle 1,4,5 \rangle \ldots \rangle$

frontier as sequence

$F = \langle 1,6 \rangle$

(root points to itself)

$P = \langle 0,0,-1,-1,0,-1 \rangle$ ← -1 = not visited

one step

cost is proportional to sum of degree of frontier

after step:

$F = \langle 3,2,5 \rangle$

$P = \langle 0,0,1,1,0,4 \rangle$

(there are work-inefficient algs for logarithmic depth)

(unknown whether work-efficient w/ poly-log depth)

neighbors

$N = \text{flatten}\ (\text{map}\ \underline{(\lambda v.\ G[v])}\ F) = \langle 0,2,3,4,0,1,2,5 \rangle$

(can be done in parallel)

($G[i]$ = neighbors of i)

$\text{tag}(S, v) = \text{map}\ (\lambda x.(x,v))\ S$

$N = \text{flatten}\ (\text{map}\ (\lambda v.\ \text{tag}(G[v], v))\ F) = \langle (0,1)(2,1),(3,1)$
$(0,4),(1,4),(2,4),(5,4) \rangle$

Scanned with CamScanner

filter out ones already visited

$N' =$ filter $(\lambda(u,v).\ P[v] = -1)\ N$

-1 indicates has not been visited

$= \langle (2,1), (3,1), (2,4), (5,4) \rangle$

N, N' – both computed in linear work
– with log span

$P' =$ inject $P\ N'$ (inject takes later case when conflict)

final filter

note: inject – work proportional to # updates
requires sequence
↳ $w = O(d \cdot n)$
here, never look back at previous P ~~inject~~

every step is log span

$S = O(d \cdot \underline{\log n})$
filter, flatten