Concurrent Disjoint Set Union

Robert E. Tarjan

Princeton University & Intertrust Technologies

joint work with Siddhartha Jayanti, Princeton

Key messages

Ideas and results from sequential algorithms can carry over to concurrent algorithms, but new ideas are needed: the design space is different in interesting ways

Design and analysis of concurrent data structures and algorithms is still a rich area to explore

Concurrency creates new and subtle complications!

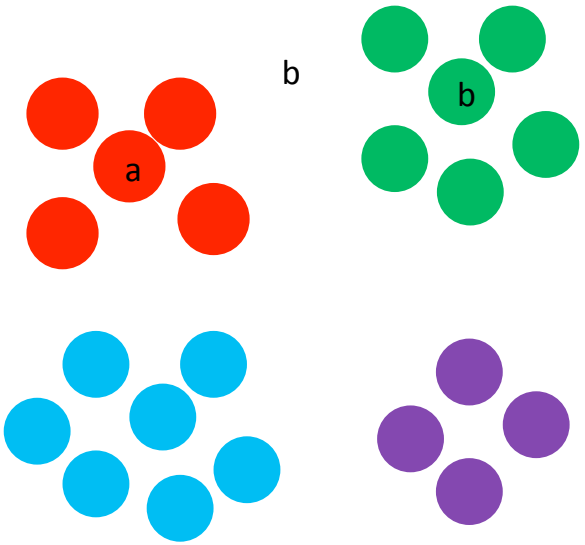
This Talk

- Disjoint set union problem
- Motivating application:

  Strong components for model checking
- Sequential set union
- Concurrency model
- Previous work
- Our results

Disjoint set union

Devise a data structure with the following operations:



***make-set*(*x*)**

***same-set*(*x*, *y*): true if *x*, *y* in same set, else false**

unite*(*x*, *y*): combine sets containing *x*, *y

Each element is in one set (sets are *disjoint*)

same-set(a,b)	False
unite(a,b)	
same-set(a,b)	True

Applications

FORTRAN compilers: COMMON and EQUIVALENCE statements

Kruskal's minimum spanning tree algorithm

Incremental connected components in graphs

Percolation

Finding dominators in flow graphs

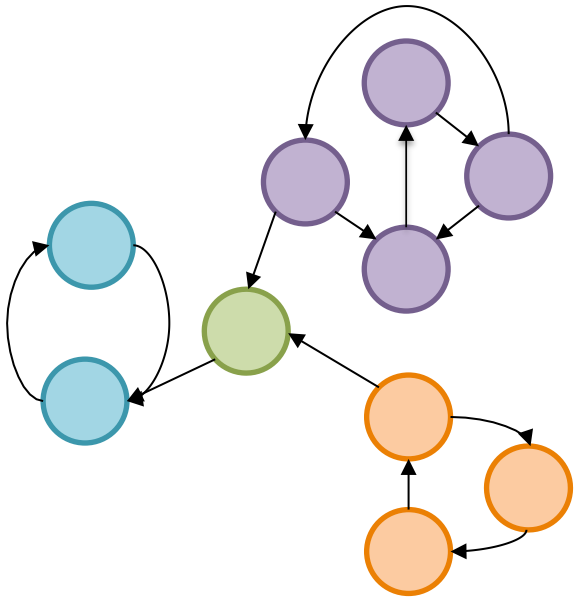
Finding strong components in digraphs

Strong Components

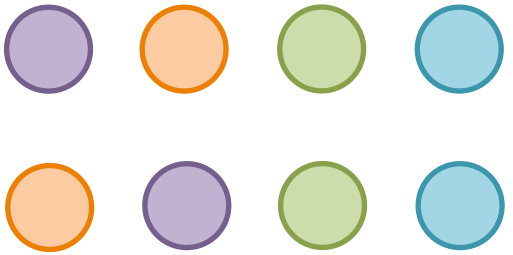
Strong Component: A maximal set of mutually reachable vertices in a directed graph.

Strong components are vertex-disjoint and can be topologically ordered: numbered so no arc leads from a larger to a smaller component.

Goal: Find components, and a topological order.



Two Topological Orders



Model-checking for program verification

In a possibly huge, implicitly defined digraph, discover whether certain sets of states can be visited infinitely often. Equivalently, are these states in a common strong component?

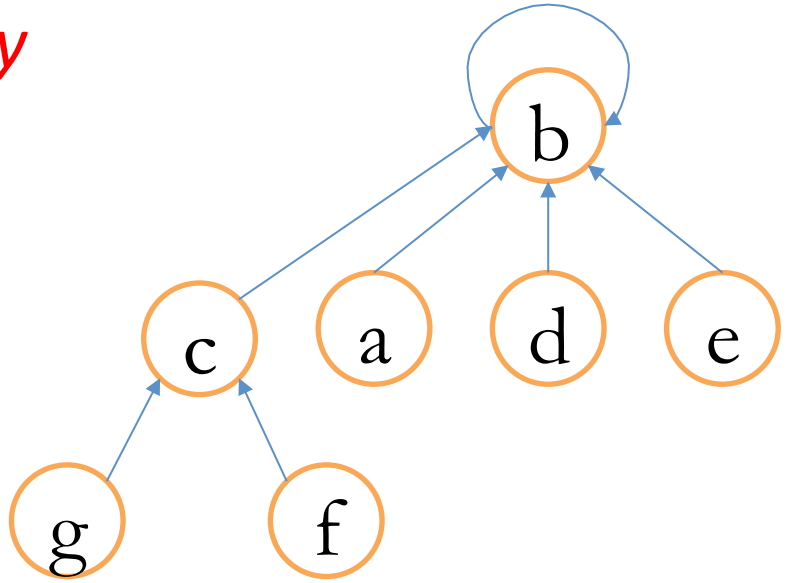
Even though strong components can be found in linear time sequentially (via a special case of disjoint set union) this may not be fast enough.

Can concurrency help?

Disjoint Sets: Compressed-tree Implementation

Represent each set by a rooted tree

- Each set element is a tree node
- Each node *x* has a parent *x.parent*
- Set information (such as value) is stored in the root
- The tree shape is *arbitrary*



Auxiliary (internal) operations

find(x): Return the root of the tree containing node x.

Naïve implementation of *find*:

return if $x.parent = x$ then x else return *find*($x.p$)

link(x, y): Unite the trees with roots x and y.

Implementation of *link*:

make y the parent of x (or x the parent of y):

$x.parent \leftarrow y$ (*or* $y.parent \leftarrow x$)

A link takes time O(1), a find takes time

O(depth of node found).

Sequential implementation of operations

make-set(x): $x.parent \leftarrow x$

same-set(x, y): return *find*(x) = *find*(y)

unite(x, y): if *find*(x) ≠ f*ind*(y) then
link(*find*(x), *find*(y))

A set operation takes O(1) time plus two finds.

Each link takes time O(1), each find takes time O(depth of node found).

Goal: reduce the (amortized) time per *find* by reducing node depths

Improve links: link by *size* or *rank* or *random index*

Improve finds: *compact* find paths

Linking by size: maintain the number of nodes in each tree (store in root). Link root of smaller tree to larger. Break a tie arbitrarily.

Linking by rank: Maintain an integer *rank* for each root, initially 0. Link root of smaller rank to root of larger rank. If tie, increase rank of new root by 1.

Linking by random index: Give each element a unique numeric index chosen uniformly and independently at random. Link root of smaller index to root of larger index.

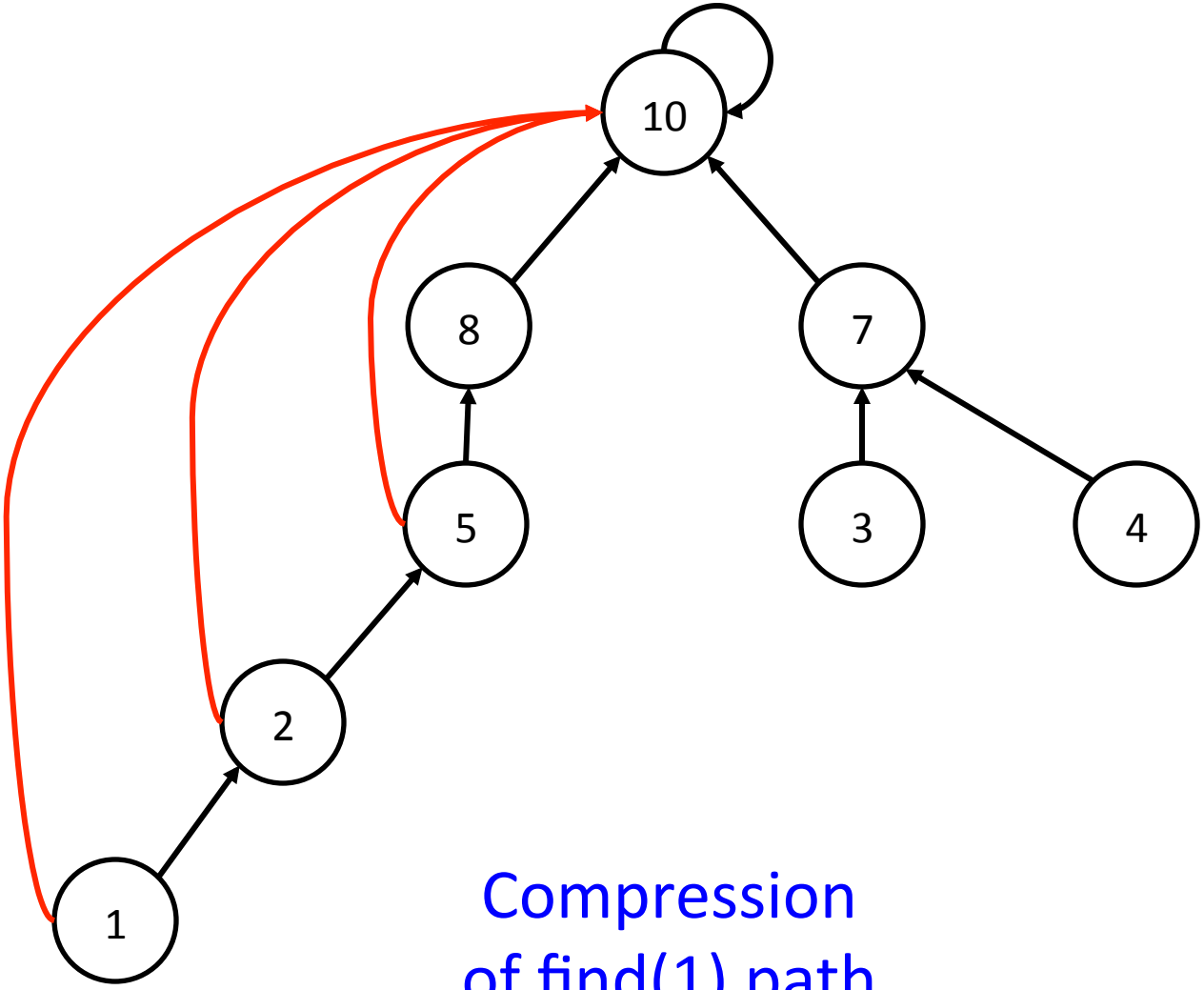
Each of these linking rules reduces the maximum path length to O(log$n$), where n = #nodes: at most $n/2^k$ nodes of height k or greater.

Path Compaction

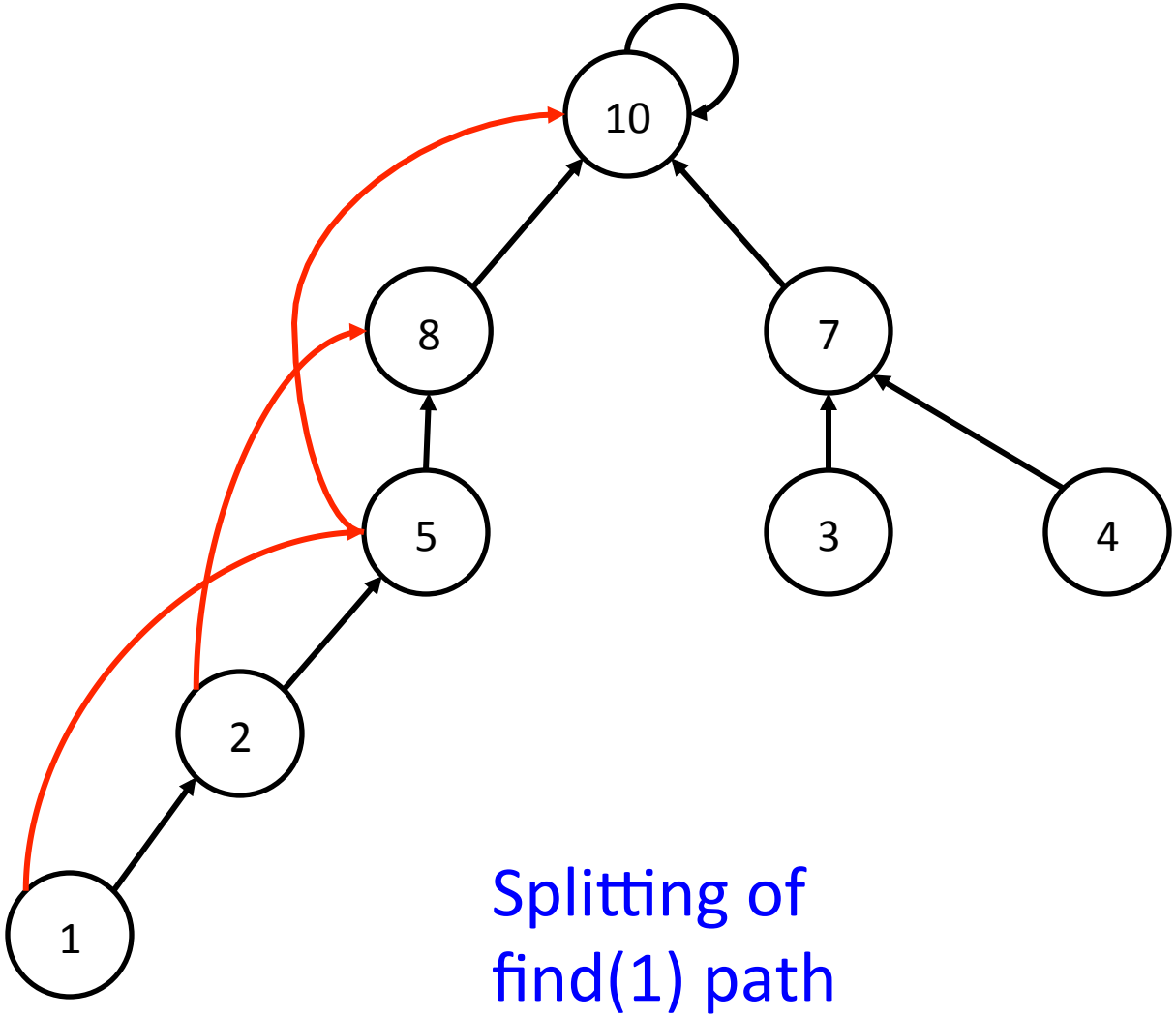
Compression: during each find, make the root the parent of each node on the find path.

Splitting: During each find, replace the parent of each node on the find path by its grandparent.

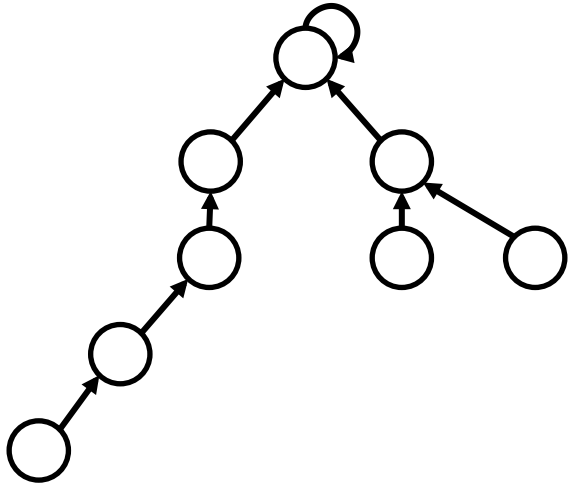
Compression takes two passes over the find path, splitting only one.



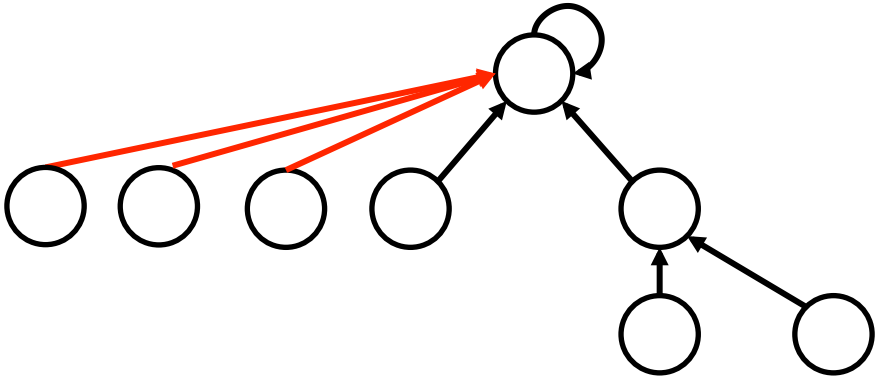
Compression
of find(1) path



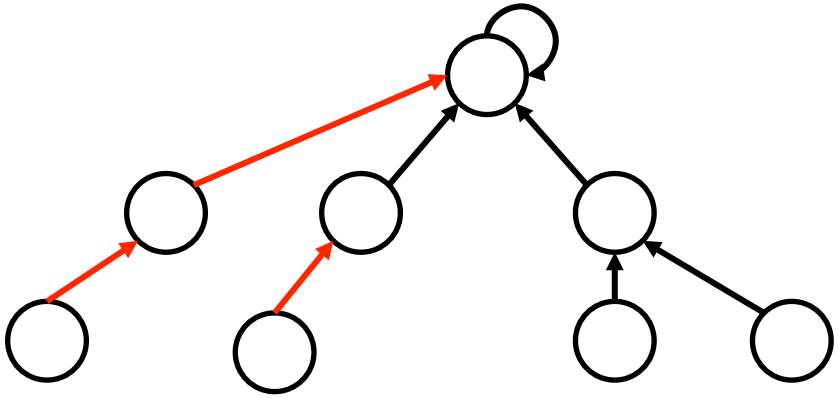
Splitting of
find(1) path



Original



After Compression



After Splitting

Running Time

Assume $n > 1$ *make-set* operations are done first, followed by $m \geq n/2$ intermixed *same-set* and *unite* operations.

Define d, the *find density*, to be $\lceil m/n \rceil$.

With linking by size, rank, or random index, and compression or splitting, total time is

$O(m\alpha(n, d))$

Ackermann's function (Péter & Robinson)

$A_0(n) = n + 1$

$A_k(0) = A_{k-1}(1)$ if $k > 0$

$A_k(n) = A_{k-1}(A_k(n-1))$ if $k > 0$, $n > 0$

$A_1(n) = n + 2$, $A_2(n) = 2n + 3$, $A_3(n) > 2^n$, $A_4(n) >$ tower of n 2's, $A_4(2)$ has 19,729 decimal digits

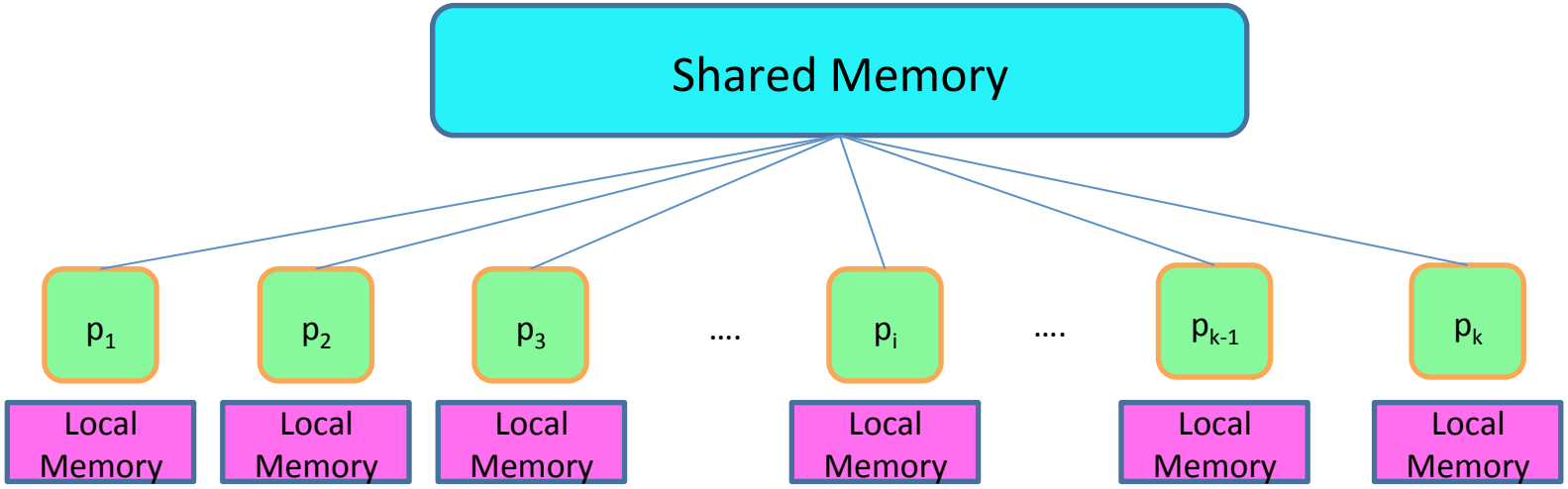
$A_k(n)$ is strictly increasing in both arguments

$\alpha(n, d) = \min\{k > 0 \mid A_k(d) > n\}$

Concurrency
Anderson & Woll, 1994

Computer model:

Shared memory multiprocessor (APRAM)



Asynchronous – Arbitrary delays between operations

Each set operation is done by one process
Several operations can run concurrently (different processes).

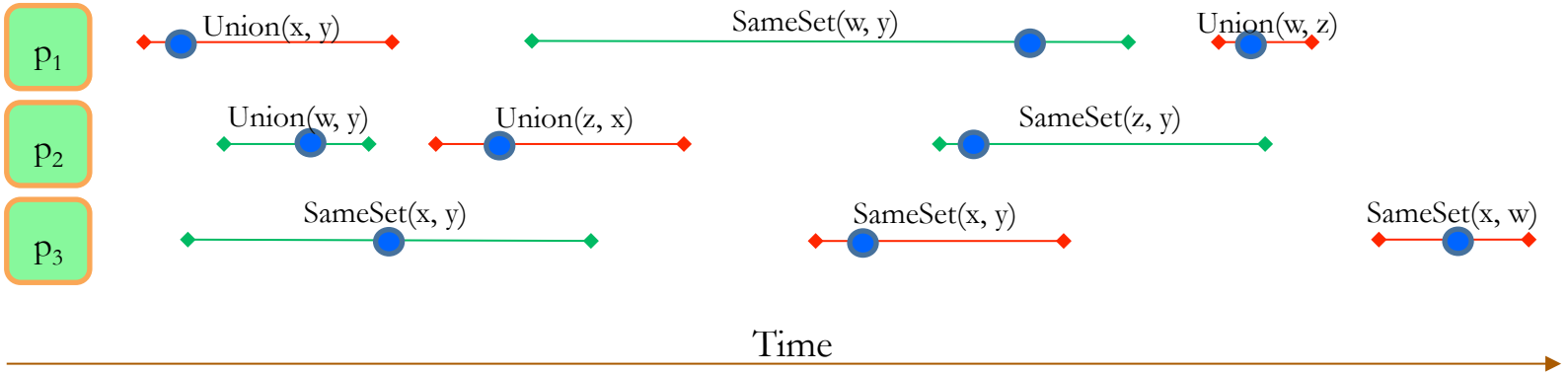
Correctness

Linearizable: Each operation can be assigned a *linearization time* during its execution, different for each operation, such that the outcome of all the operations is the same as if they were executed instantaneously at their linearization times.

Wait-free: Each process completes each of its operations in a bounded number of its own steps.

Correctness

Linearizability [Herlihy, Wing 1990]



Efficiency

Total work: total number of steps taken by all processes, as a worst-case function of n, m, and p (the number of processes).

Goal: Total work not too much bigger than the sequential time bound and sublinear in p: then concurrency may help.

Synchronization Primitives for wait-freedom

Compare & Swap

CAS(*x*, *y*, *z*): **if** *x* = *y* **then** {*x* ← *z*; **return true**}

else return false

Double Compare & Swap

DCAS(*x*, *y*, *z*, *u*, *v*, *w*): **if** *x* = *y* **and** *u* = *v*

then {*x* ← *z*; *u* ← *w*; **return true**}

else return false

Previous work: Anderson & Woll 1994

Concurrent version of linking by rank with splitting using CAS.

Big problem: CAS seems too weak: linking by rank requires changing a rank in one node and a pointer in another.

Their algorithm does not avoid rank ties.

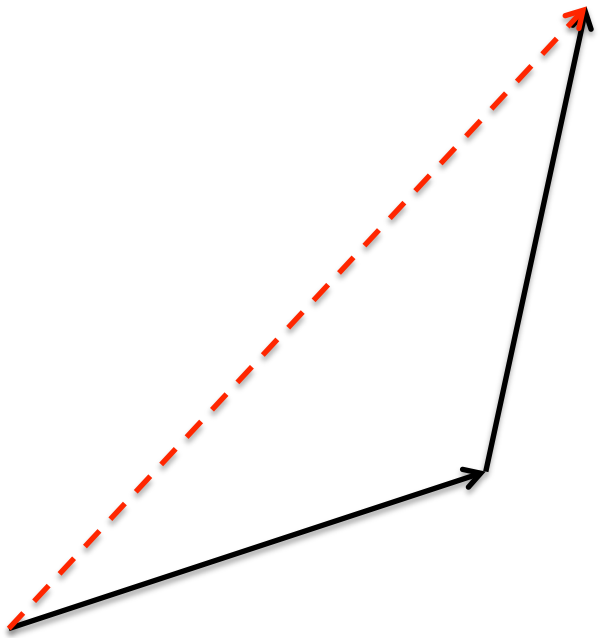
Work bound is $O(m(\alpha(n, 1) + p))$: not so good, and "proof" is buggy: they did not account for interactions between different processors doing splitting along overlapping paths.

Our goal

Simple algorithms with good work bounds, sublinear in p if possible

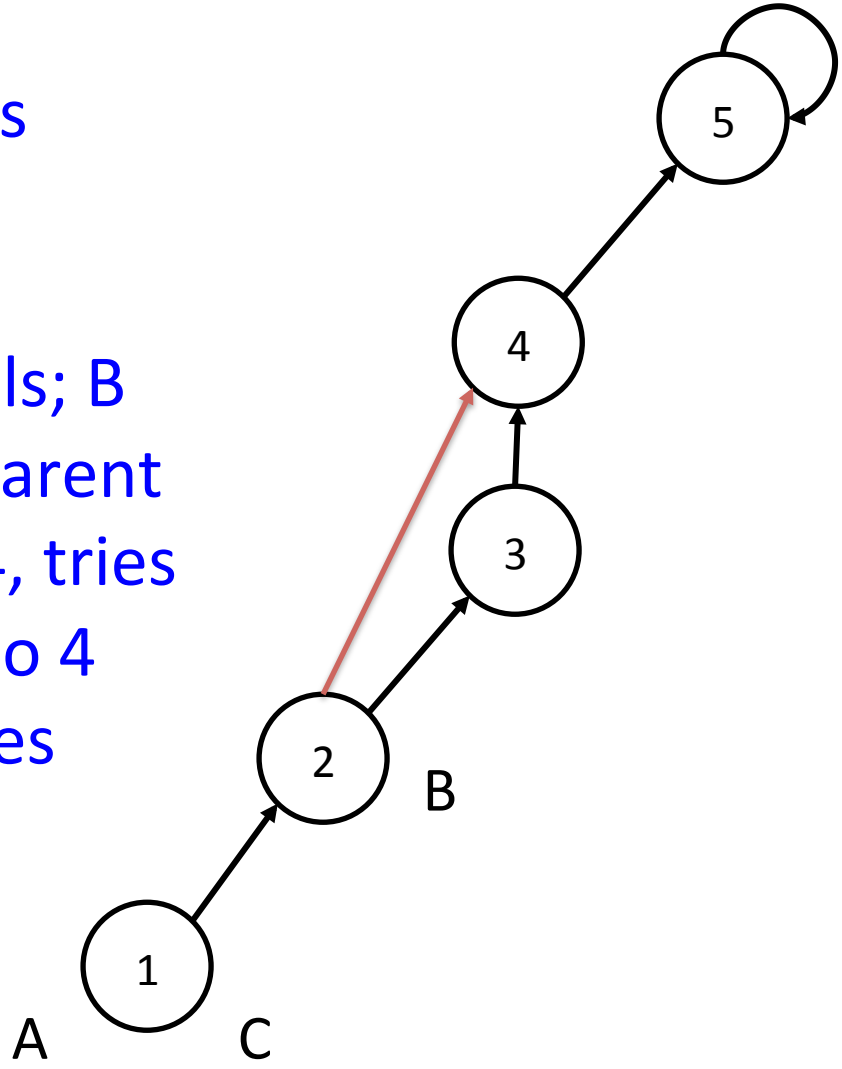
Anderson & Woll gave a simple wait-free implementation of find with splitting using CAS, but their analysis is not correct.

Splitting: can do shortcuts via CAS but concurrent threads can interfere



A CAS can fail because another CAS does a less favorable change:

A visits 1, 2, 3 then stalls; B visits 2, 3, 4, changes parent of 2 to 4; C visits 1, 2, 4, tries to change parent of 1 to 4 but A wakes up, changes parent of 1 to 3.



Our Results

Two splitting algorithms:

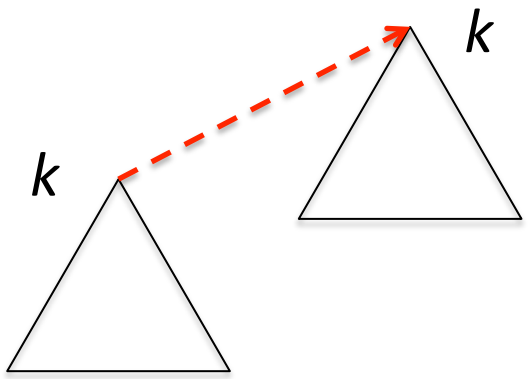
- 1-try splitting: try each parent change once before moving to the next node
- 2-try splitting: try each parent change twice before moving to the next node

We get slightly better bounds for 2-try splitting

Linking

Can do unranked links using CAS

but DCAS needed for ranked links



Linking Algorithms

Four concurrent linking algorithms:

- Linking by rank (DCAS)
- Linking by random index (CAS)
- Hybrid linking by rank via coin-flipping (CAS)
- Hybrid linking by rank via deterministic coin-flipping (CAS)

Our Bounds

Worst-case time per operation

O(log$n$) with or without split/compress

Total work with "1-try" splitting

$O(m(\alpha(n, \lceil m/(np^2) \rceil) + \log(np^2/m + 1)))$

Total work with "2-try" splitting

$O(m(\alpha(n, \lceil m/(np) \rceil) + \log(np/m + 1)))$

Bounds are worst-case for deterministic linking,

Expected for randomized linking

Hybrid linking

link(*v*, *w*): if *v* and *w* have equal rank, first change the parent of *v*, or the rank of *w*?

Flip a fair coin to decide: when trying to link two equal-rank nodes *v* and *w,* with $v < w$ in some (arbitrary) total order, if heads try to make *w* the parent of *v*; if tails, try to add 1 to the rank of *v*

Analysis

Tree depth is logarithmic (worst-case for linking by rank, high-probability for linking by randomized index or hybrid}. This gives O(log$n$) time per operation.

Work bound: Let $d = m/(np)$(2-try splitting) or $d = m/(np^2)$ (1-try splitting). If $d \geq 1$, the sequential analysis extends to give a bound of O($\alpha(n, d)$) work per find plus O(pd) or O(p^2d) work per node (p *or* p^2 times the sequential bound). The total work is O($m\alpha(n, d)$).

Work bound continued

The other case is $d < 1$. The number of nodes of rank at least r is at most $n/2^r$. We apply the sequential argument to the nodes of rank at least $\lg(1/d)$, of which there are at most nd. These nodes account for total work $O(m\alpha(n, 1) + nkd) = O(m\alpha(n, 1))$. The low-rank nodes on find paths account for an additional $O(\log(1/d))$ work per find.

Current Work

Implement sets with values and sets with iteration while preserving efficiency: seems to require relaxing linearization.

Idea: Allow subsets of unite operations to be replaced by equivalent subsets, implement using a bifurcating queue.

Questions for the future

Is our amortized upper bound tight for the problem?

Concurrent strong components?

Other concurrent data structures: binary search trees?

Thanks!