An Efficient Lock-free Logarithmic Search Data Structure Based on Multi-dimensional List

Deli Zhang Damian Dechev

Department of Computer Science
University of Central Florida

April 9, 2016

Table of Contents

Concurrent Dictionary

- Abstract Data Type
  - Supports INSERT, DELETE, and FIND
  - Keys are totally ordered
- Typical Implementations
  - Binary Search Trees (BST)
  - Skip-lists
  - Hash Tries
- Lock-freedom
  - Scalable fine-grained synchronization
  - Fault tolerance

Lock-free BST

- Node not found $\neq$ Key not found
  - Embed logical ordering [Drachsler et al. 2014]
  - Use external (leaf-oriented) trees [Ellen et al. 2010]
- Balancing induces bottleneck
  - Relaxed balancing [Bronson et al. 2010]
  - Background balancing [Crain et al. 2013]

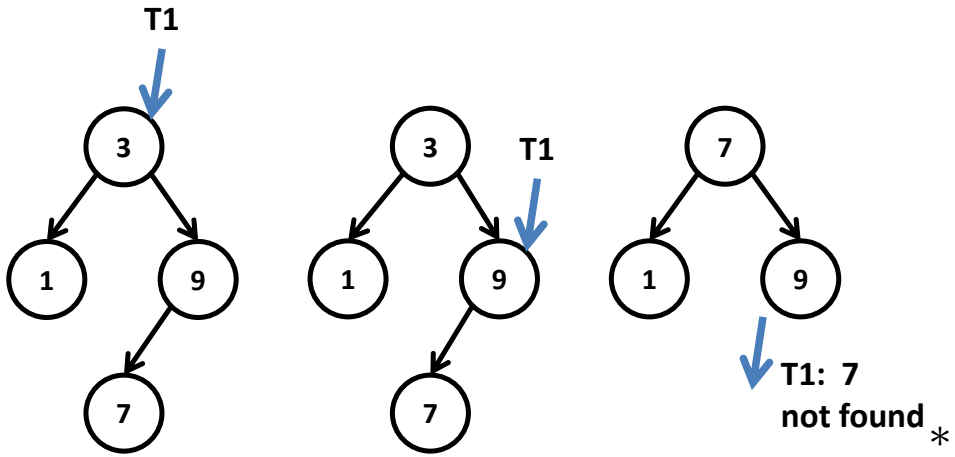
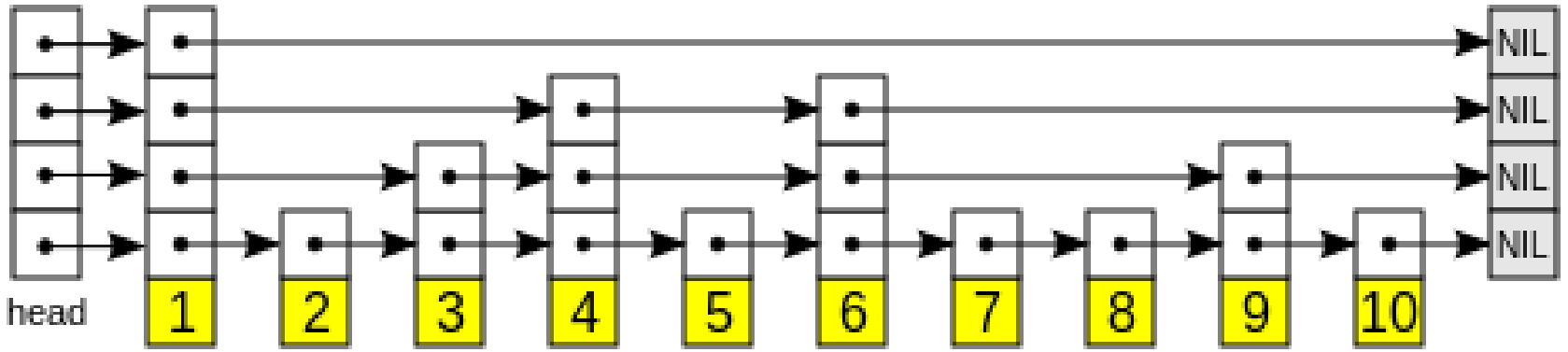


*Illustration from Drachsler et al. 2014

Lock-free Skip-list

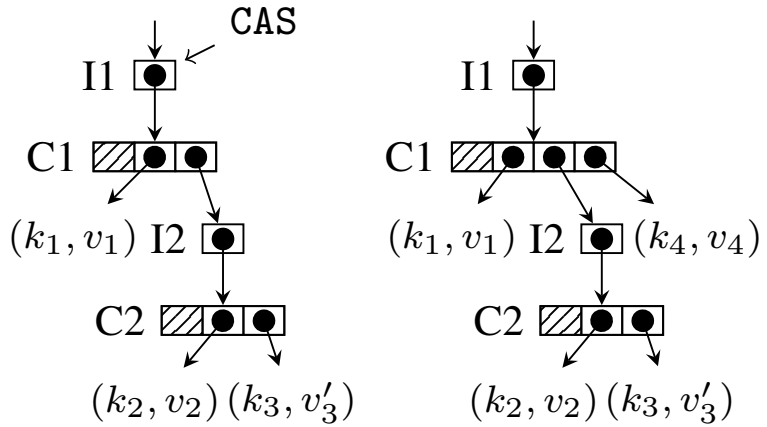
- Probabilistically balanced alternative
  - Bottom level linked list
  - Redundant short-cut links
  - Exponentially lower probability
- Concurrency limitation [Fraser 2004, Crain et al. 2013]
  - INSERT/DELETE update multiple nodes



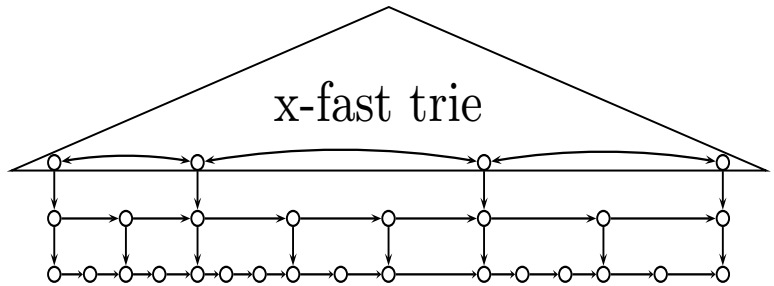
† Illustration from wikipedia

Lock-free Tries

- Ctrie [Prokopec et al. 2012]
  - A hash array mapped trie
  - Value stored internally
  - Hot spot: Intermediate (I) nodes
- SkipTrie [Oshman and Shavit 2013]
  - Y-fast trie with skip-list
  - Require double-word CAS
  - Value stored externally
  - Hot spot: inserting prefixes



Ctrie Insert

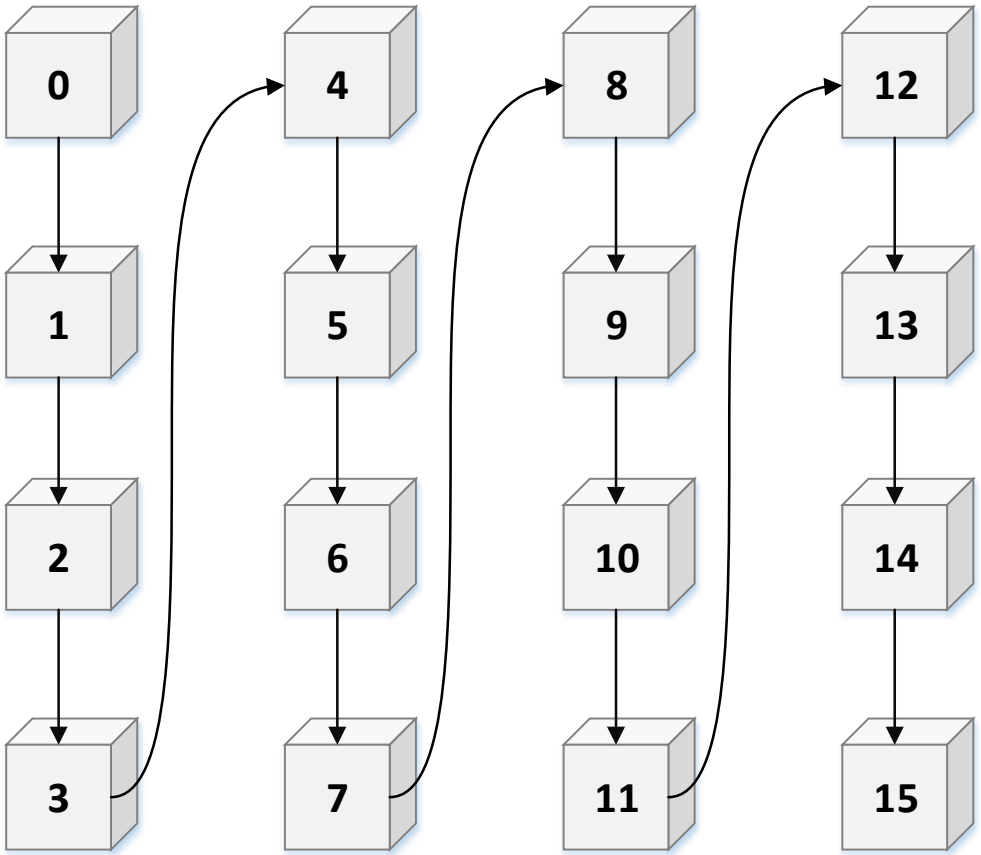


SkipTrie[‡]

[‡] Illustration from Prokopec et al. 2012, and Oshman and Shavit 2013

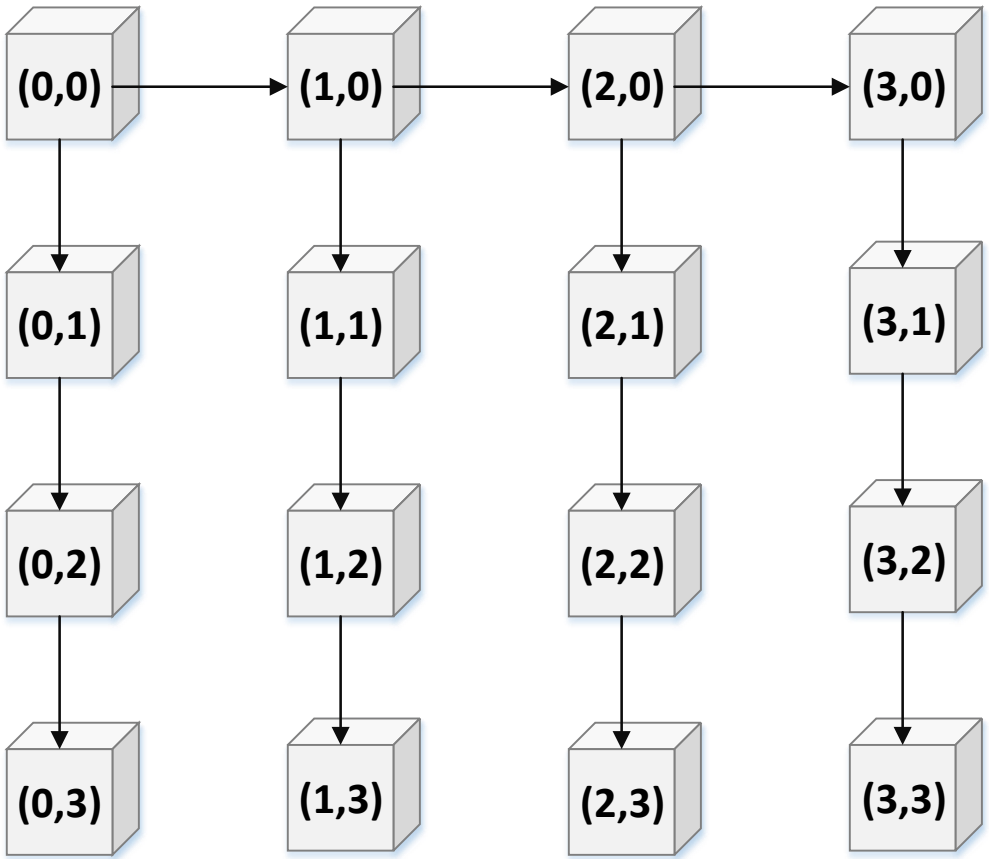
- What makes an efficient lock-free dictionary?
  - Low branching factor — avoids hot spots
  - Localized updates — reduces access conflicts
  - Fixed key positions — simplifies predecessor query
- Proposed Multi-dimensional List (MDList)
  - Lower level nodes have smaller branching factor
  - INSERT/DELETE modifies at most 2 adjacent nodes
  - Unique layout per logical ordering
  - No balancing/randomization required
  - Memory efficient

Ordered Linked List



- Re-arranged into columns
- Partitioned sub-lists
- $\mathcal{O}(N)$ FIND

Ordered 2D List



- Short-cut links on top
- Vector coordinates (d_0, d_1)
- Lexicographical ordering
- Worst-case $\mathcal{O}(\sqrt{N})$ FIND

Multi-dimensional List

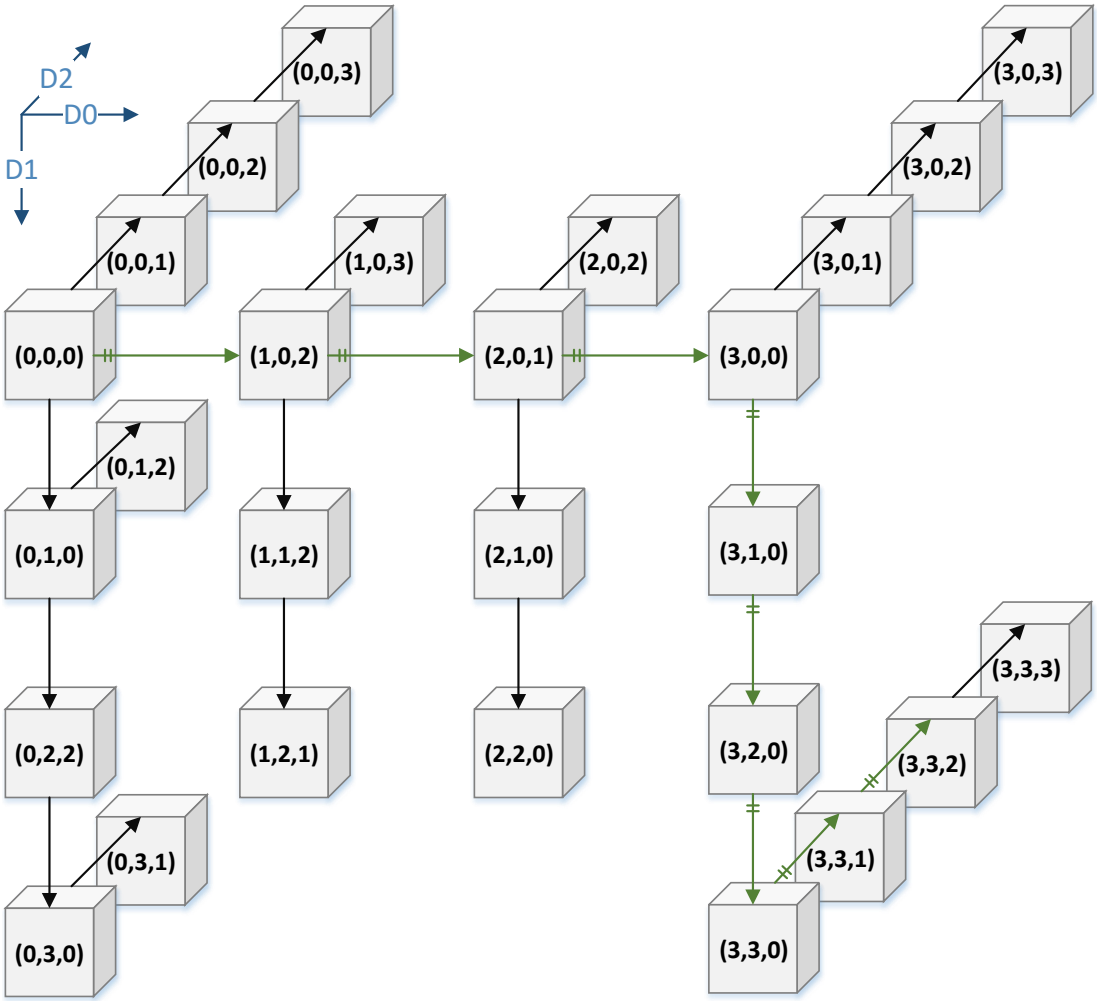
Definition

A D-dimensional list is a tree in which each node is implicitly assigned a dimension of $d \in [0, D)$. The root node's dimension is 0. A node of dimension d has no more than $D - d$ children, where the mth child is assigned a dimension of $d' = d + m - 1$.

Definition

Given a non-root node of dimension d with coordinate $\mathbf{k} = (k_0, ..., k_{D-1})$ and its parent with coordinate $\mathbf{k}' = (k'_0, ..., k'_{D-1})$ in an ordered D-dimensional list: $k_i = k'_i,\ \forall\ i \in [0, d) \wedge k_d > k'_d$.

Coordinate Mapping



- $k \mapsto (d_0, d_1, ..., d_{D-1})$
- *Injective* and *monotonic*
- Preferably uniform
- Base conversion: choose $base = \lceil \sqrt[D]{U} \rceil$

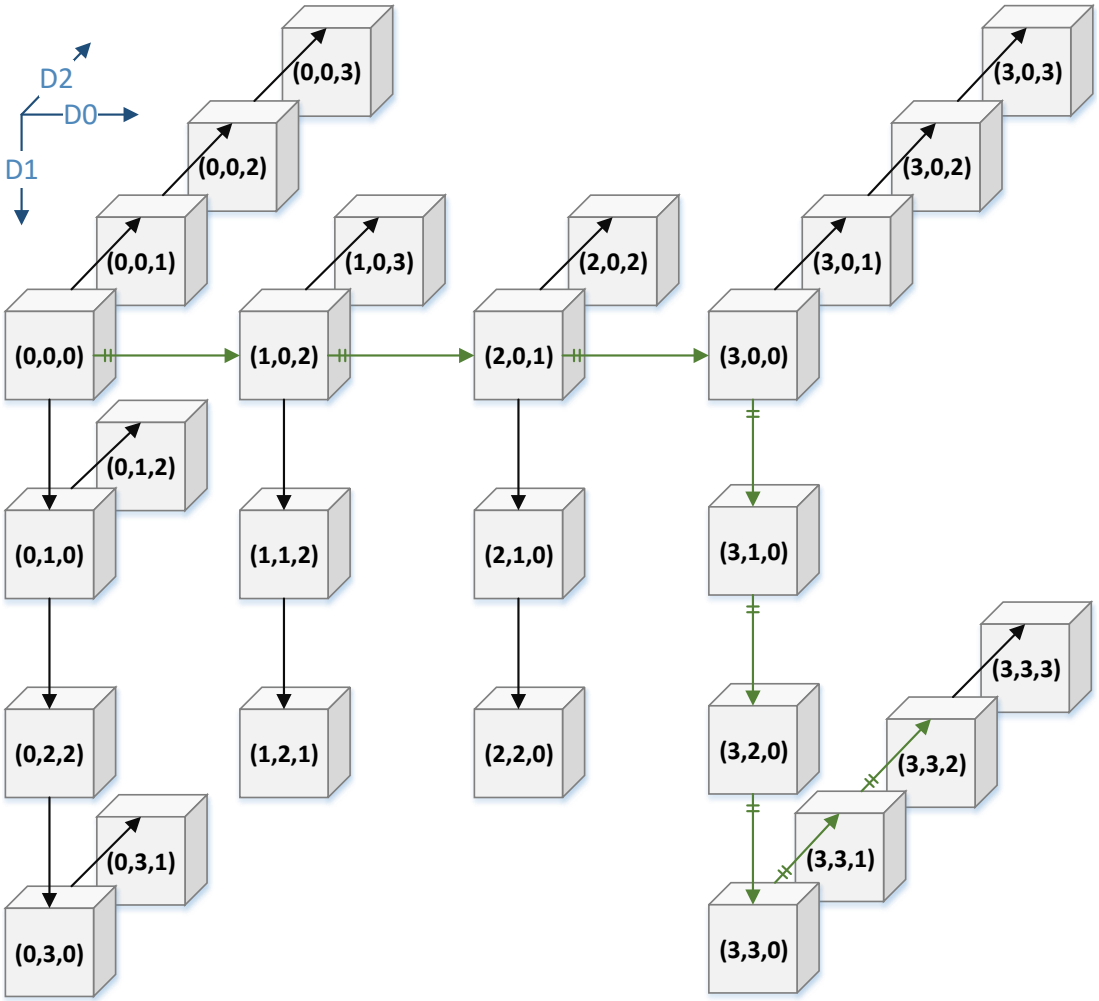
Example

$U = 64, D = 3, base = 4$

$(63)_{10} = (333)_4$

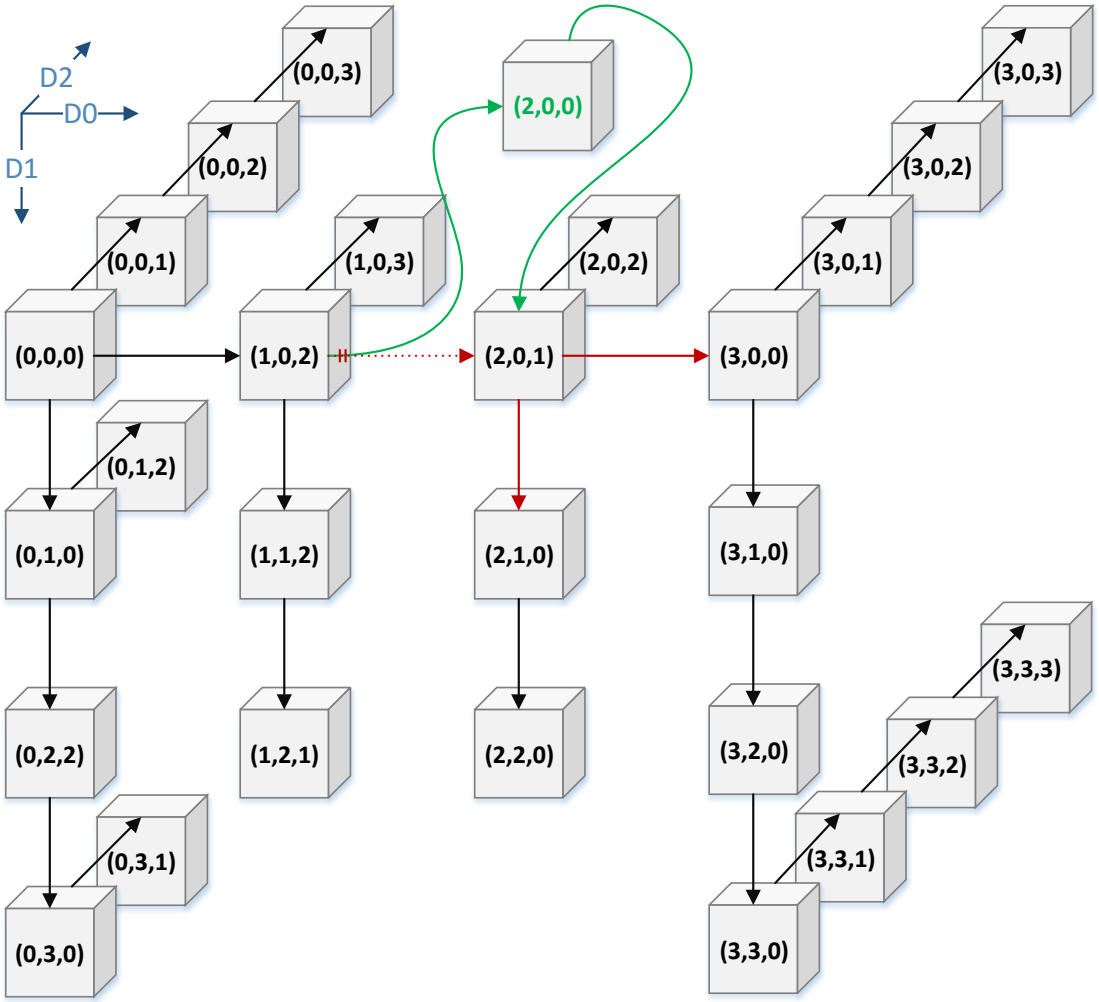
$(34)_{10} = (202)_4$

Find Operation



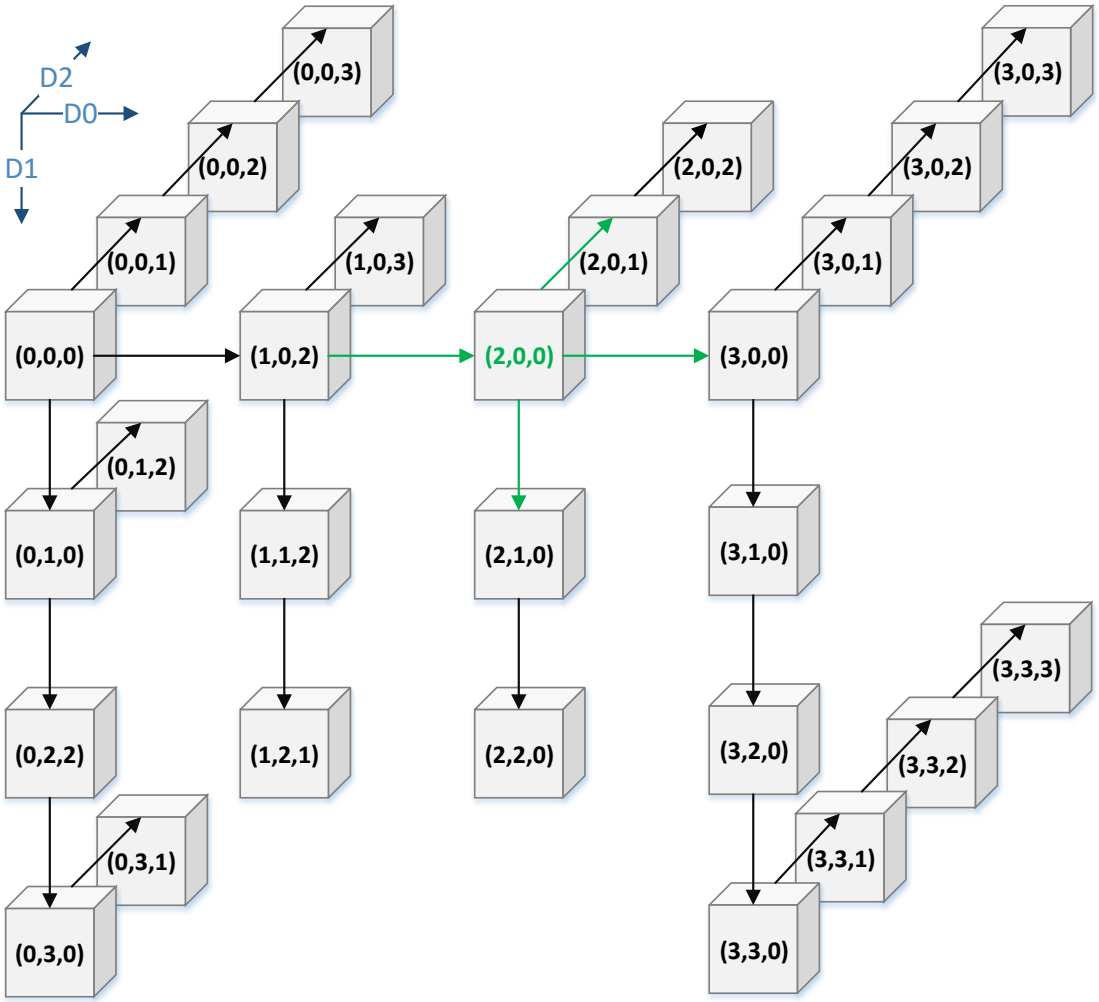
- Recursively traverse sub-lists
- Comparing coordinates from $d = 0$
- Worst-case $\mathcal{O}(D \sqrt[D]{n})$ time
- Choose $D = \log U$ then $\mathcal{O}(\log U \sqrt[\log U]{U}) = \mathcal{O}(\log U)$

Insert — 2 Steps



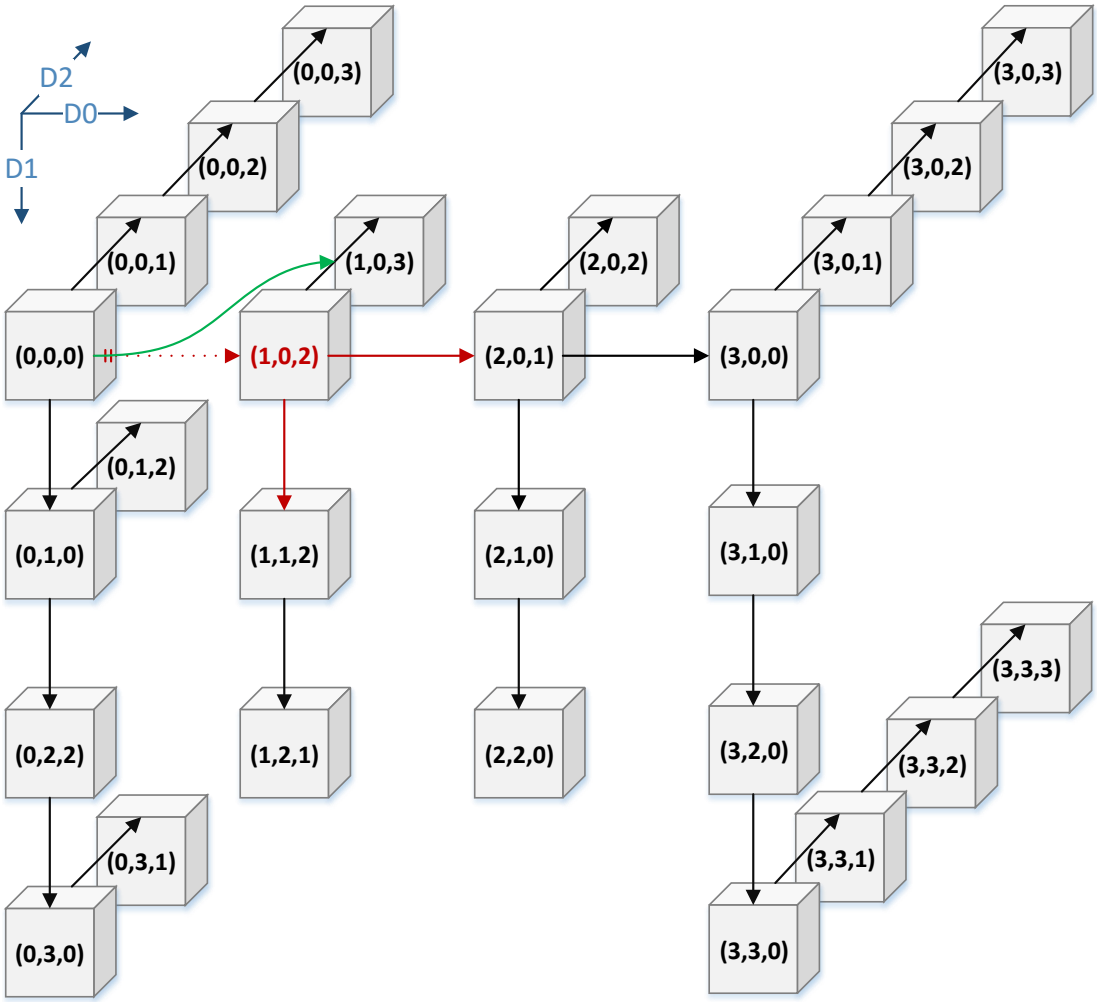
1. Pointer swing
   - Predecessor query locate $pred, curr, d_p, d_c$
   - CAS updates $pred.child[d_p]$

Insert — 2 Steps



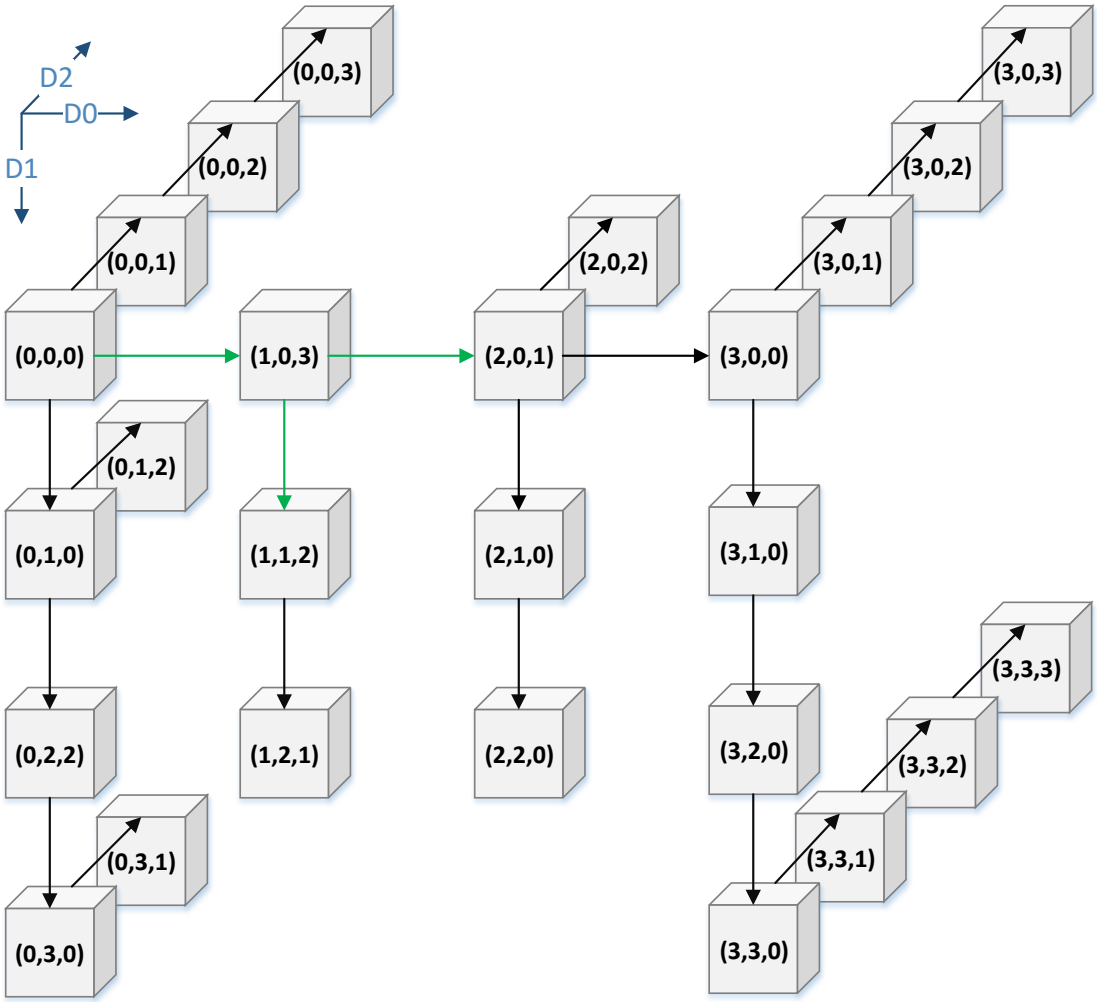
1. Pointer swing
   - Predecessor query locate $pred, curr, d_p, d_c$
   - CAS updates $pred.child[d_p]$
2. Child adoption
   - Necessary if $d_c \neq d_p$
   - $curr.child[d_p : d_c]$ transferred to new node
   - Use descriptor for helping in case process is delayed

Normal Delete — 2 steps



1. Prompt the next node
   - Examine coordinates from $D-1$
   - Swing *pred*'s pointer

Normal Delete — 2 steps



1. Prompt the next node
   - Examine coordinates from $D-1$
   - Swing *pred*'s pointer
2. Child transfer
   - Inverse of child adoption
   - Potential conflicts

Normal Delete Does Not Work

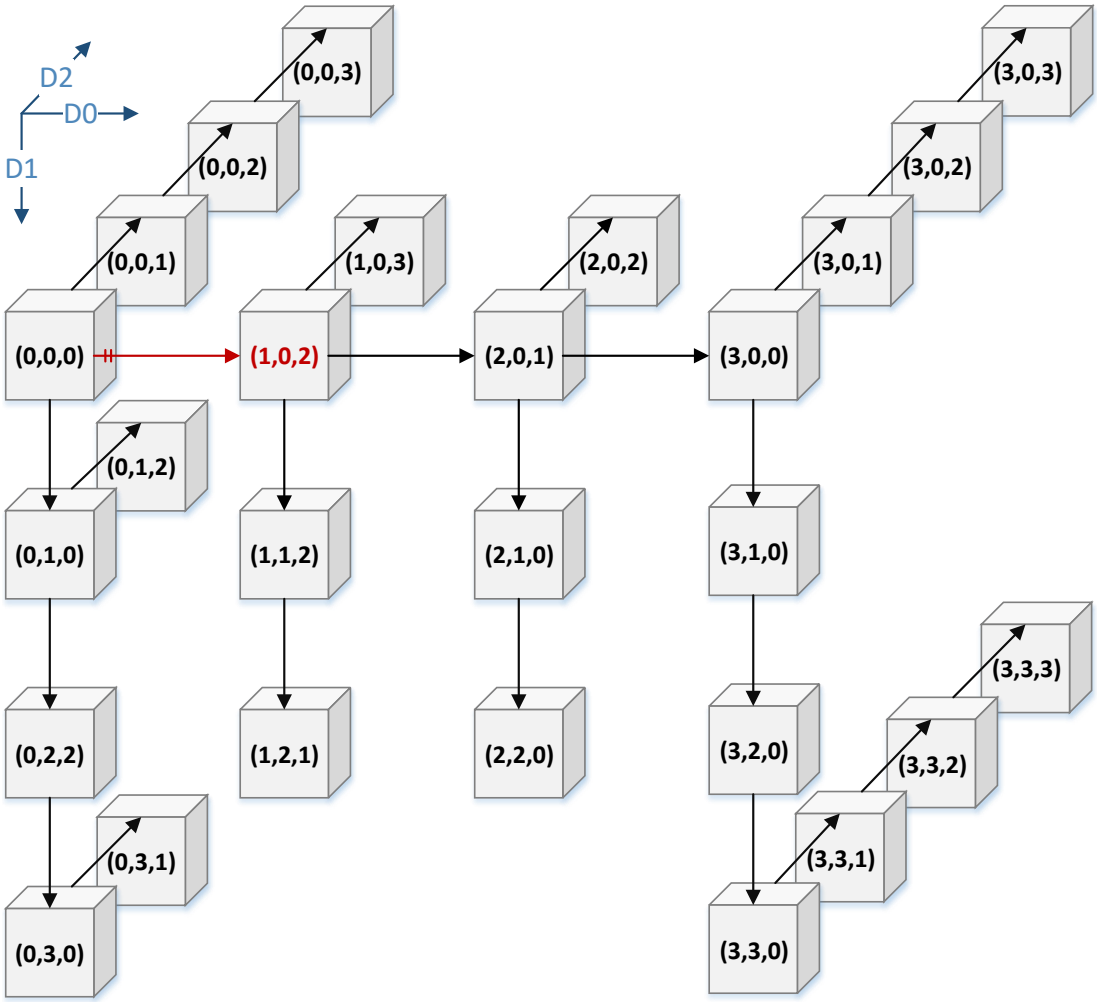
- INSERT may demote a node (i.e., increase its dimension)
- DELETE may promote a node (i.e., decrease its dimension)

Synchronization Issue

Due to the helping mechanism, several threads may execute child adoption on the same node. Without additional synchronization, we cannot know if all of them have finished. Data races may arise among ongoing child adoption and child transfer processes.

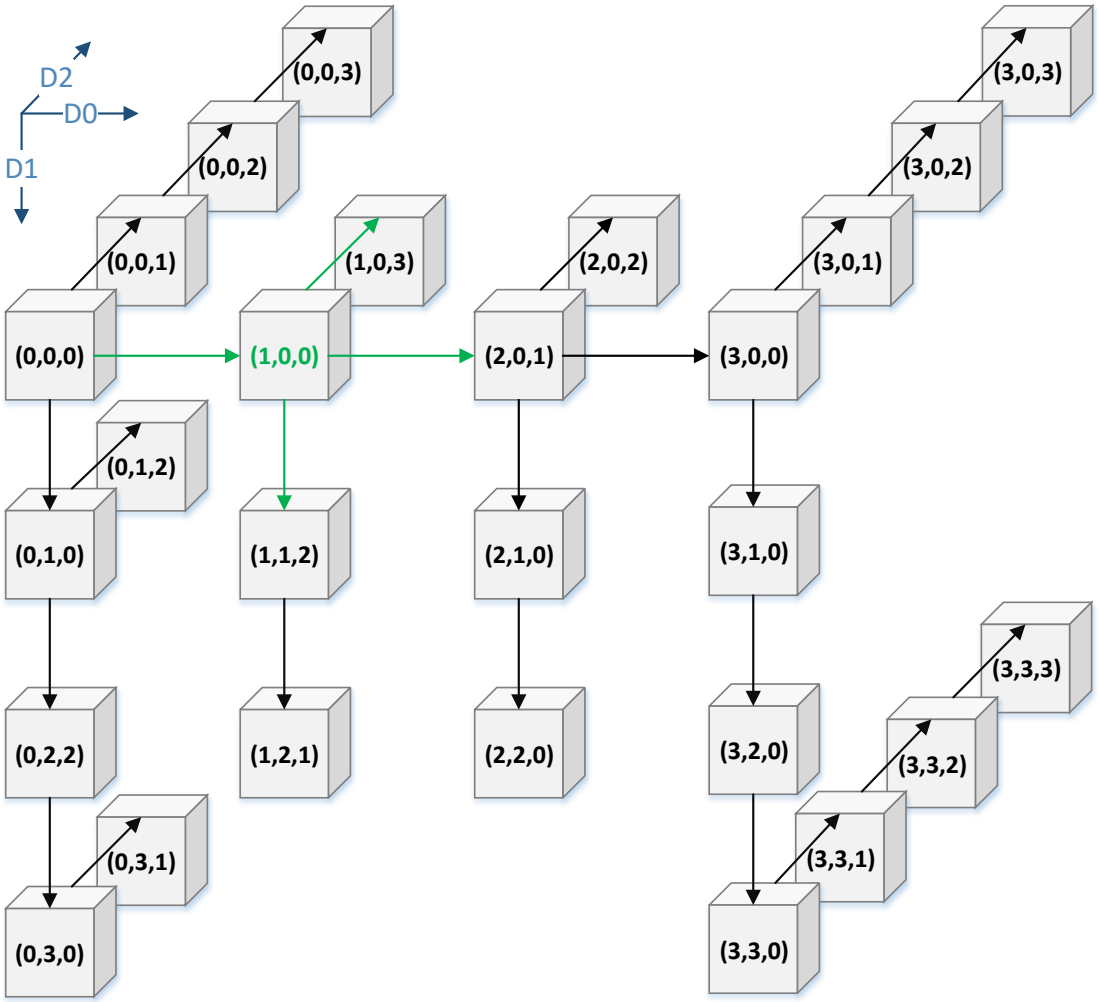
- Solution: Keep dimension change *unidirectional*

Asymmetrical Delete — Decoupled Physical Removal



1. Mark for logical deletion
   - Still valid for routing

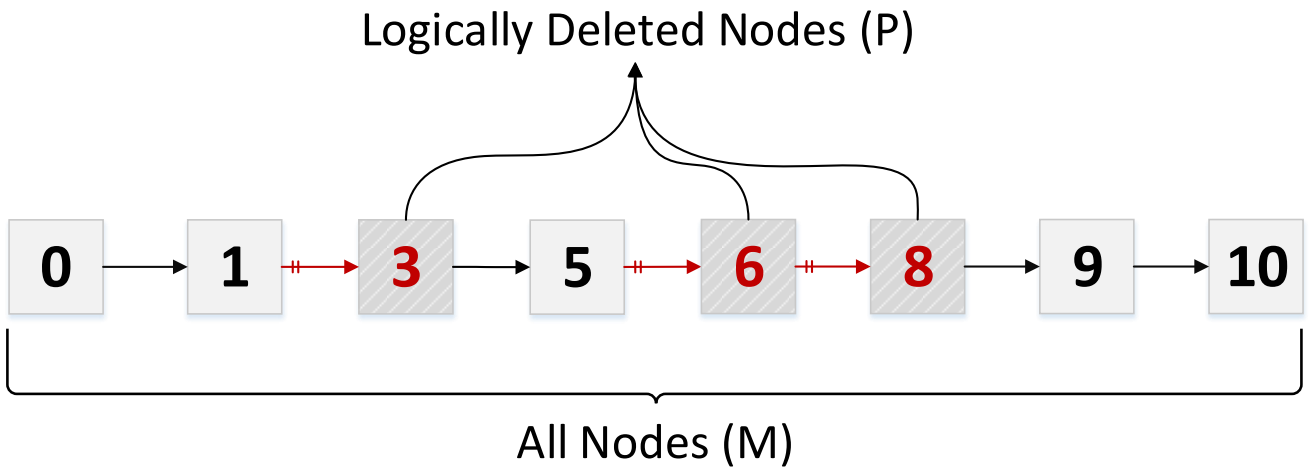
Asymmetrical Delete — Decoupled Physical Removal



1. Mark for logical deletion
   - Still valid for routing
2. INSERT purge marked node
   - Adopt children in $[d_p, D)$

- Reuse child adoption
- Unifies help protocol
- Simplifies synchronization

Abstract State



- The abstract state of the dictionary $S = M \setminus P$
- Linearization points
  - INSERT: when CAS updates predecessor's child pointer
  - DELETE: when CAS marks predecessor's child pointer
  - FIND: when predecessor's child pointer is read

Environment

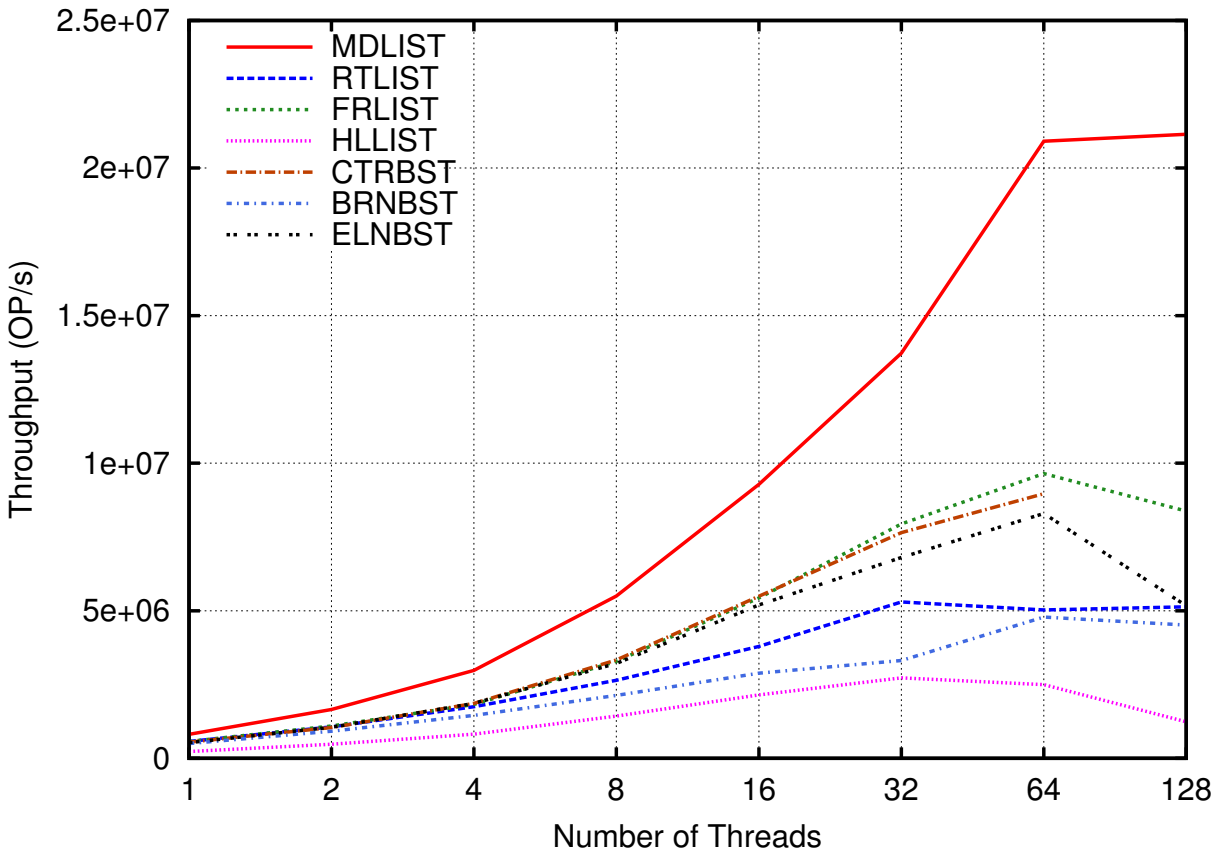
- Hardware
  - 64-core NUMA (4 AMD Opteron @2.1GHz)
  - 6-core (12 w/ Hyper-threading) SMP (Intel Xeon @2.9GHz)
- Software
  - GCC 4.7 w/ O3
  - Disabled memory reclamation
  - Uses thread-caching malloc
- Micro-benchmark
  - Write-dominated, read-dominated, and mixed workloads
  - 1K, 1M, and 1G key universes

Alternatives

- BST
  - Lock-based relaxed AVL by Bronson et al. (BRNBST)
  - Lock-free unbalanced BST by Ellen et al. (ELNBST)
  - RCU-based Citrus tree by Arbel et al. (CTRBST)
- Skip-list (max tower height 30)
  - Lock-free skip-list by Fraser (FRLIST)
  - Lock-free skip-list by Herlihy (HLLIST)
  - Lock-free rotating skip-list by Dick et al. (RTLIST)
- MDList (dimension 16)

‡BRNBST, ELNBST, and HLLIST are based on C++ implementation by Wicht et al. 2012

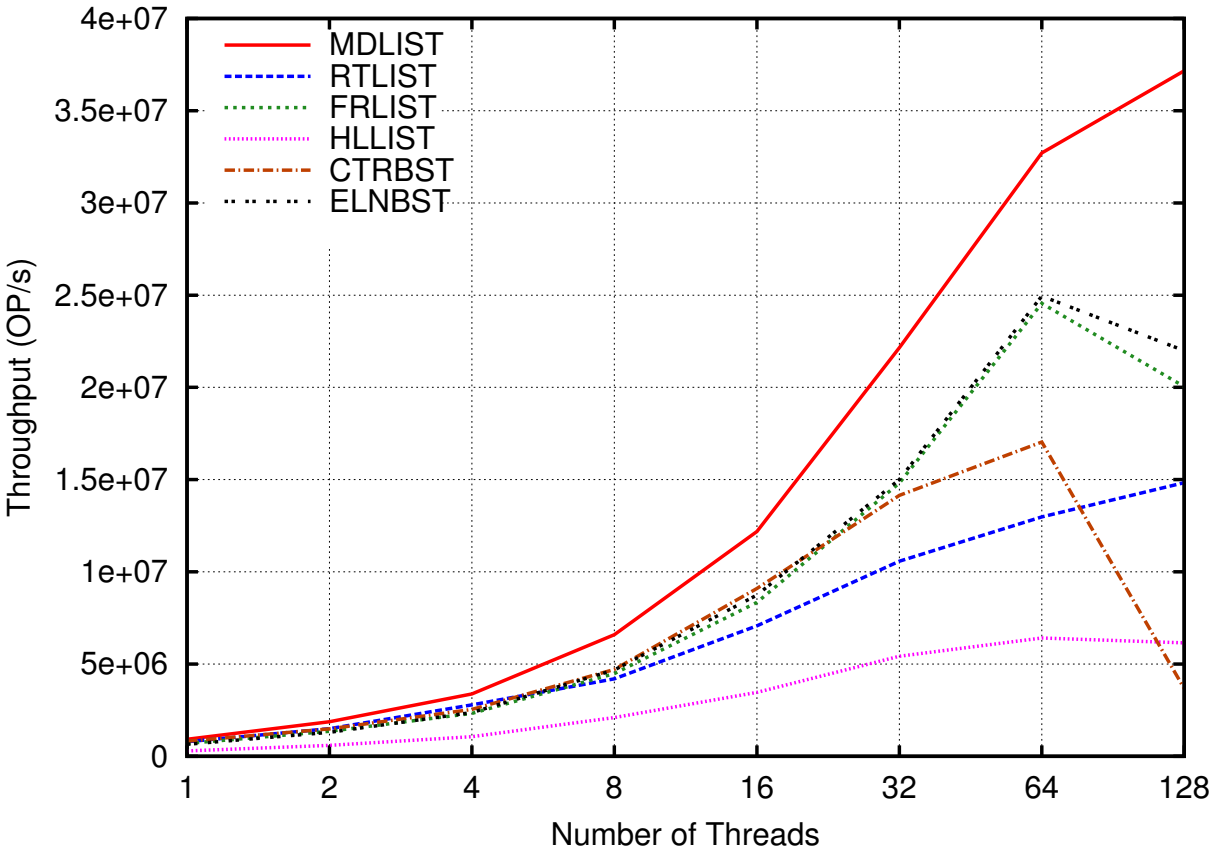
Throughput — NUMA Write-dominated Workload



1G Keys, 50% Insert 50% Delete

- As much as 100% speedup for 64 threads
- Optimized by localized node modifications

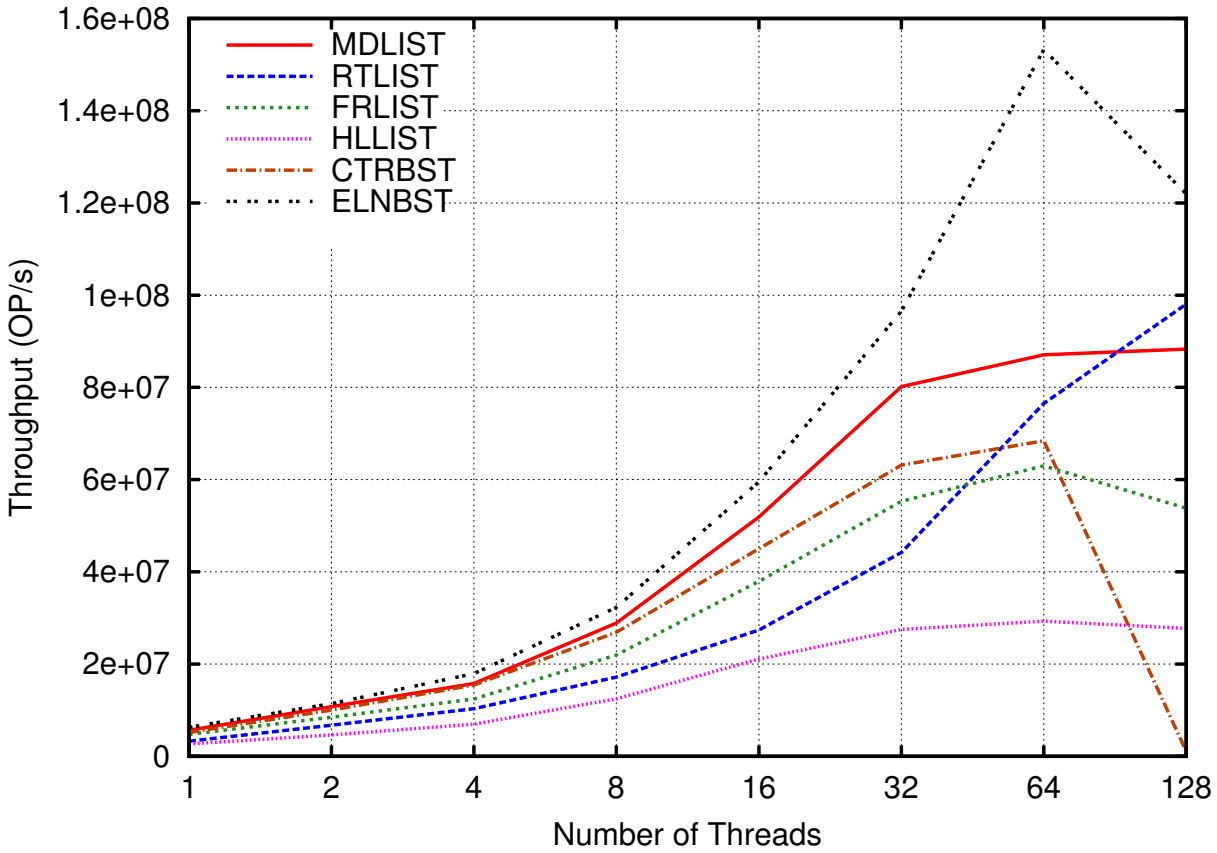
Throughput — NUMA Mixed Workload



1M Keys, 20% Insert 10% Delete 70% Find

- As much as 50% speedup for 64 threads

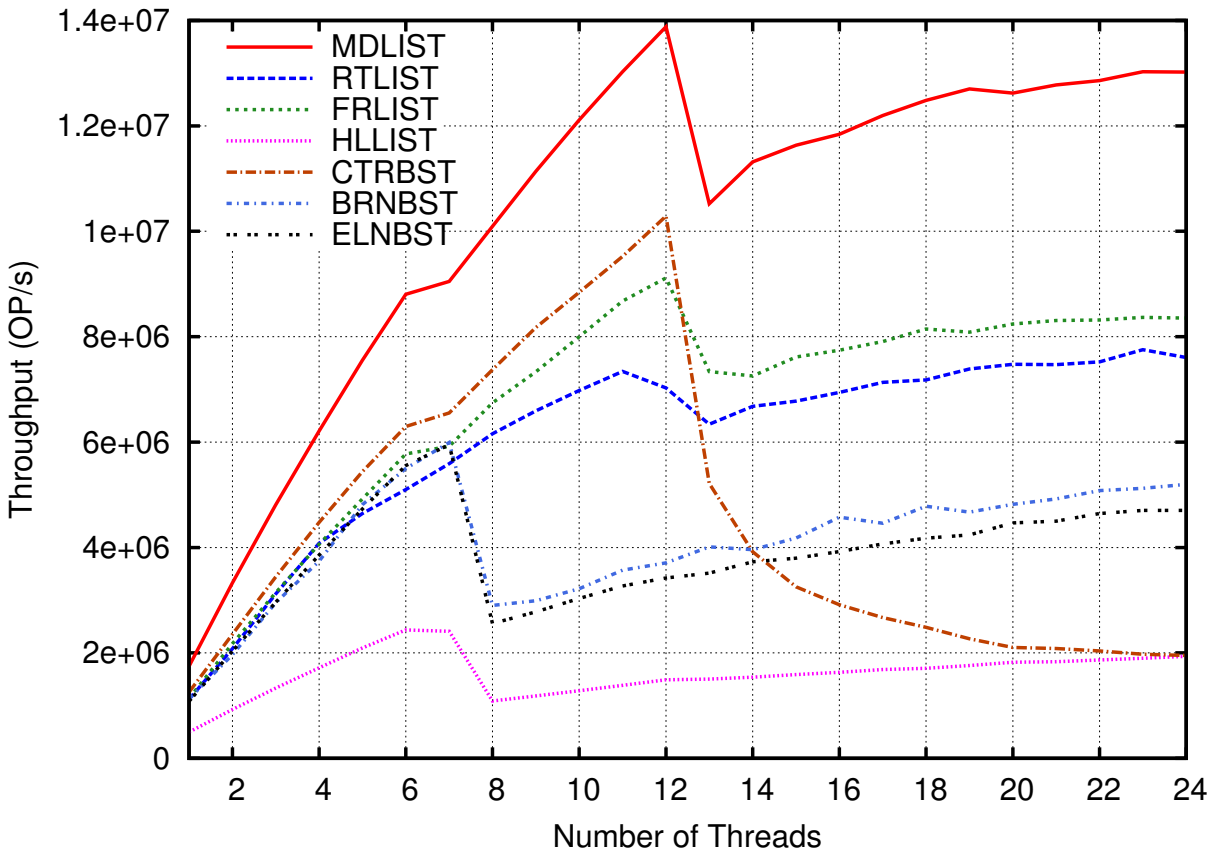
Throughput — NUMA Read-dominated Workload



1K Keys, 9% Insert 1% Delete 90% Find

- BSTs have shallow depth
- 16 dimension is too much for 1000 keys

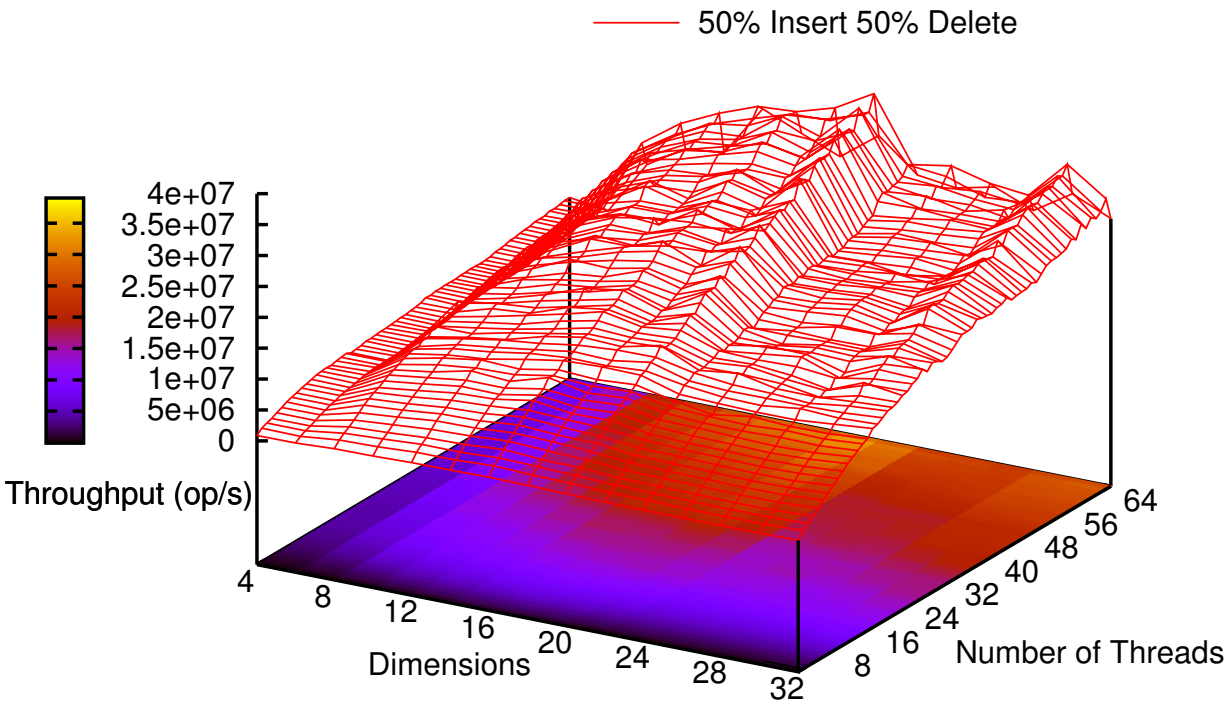
Throughput — SMP Mixed Workload



1M Keys, 30% Insert 20% Delete 50% Find

- 40% speedup for 12 threads
- Able to exploit hyper-threading

Dimension Sweep



1M Keys, 50% Insert 50% Delete

- For any number of threads, max throughput occurs when $D = 20$
- Performance of MDList is workload dependent and concurrency independent

Summary

- Performance characteristics
  - Excels at high level of concurrency and large key space
  - Optimized for write operations
  - Tuning based on workload and key space is possible

Summary

- Performance characteristics
  - Excels at high level of concurrency and large key space
  - Optimized for write operations
  - Tuning based on workload and key space is possible
- Memory overhead per key-value pair
  - Sequential: $D + 4$ bytes — cached coordinates and child pointer
  - Lock-free: additional 10 bytes — child adoption descriptor

Summary

- Performance characteristics
  - Excels at high level of concurrency and large key space
  - Optimized for write operations
  - Tuning based on workload and key space is possible
- Memory overhead per key-value pair
  - Sequential: $D + 4$ bytes — cached coordinates and child pointer
  - Lock-free: additional 10 bytes — child adoption descriptor
- Different key partition from tries

Type	Minimal Key	Branching Factor	Common Prefix Length
MDList	In root node	$K = D - d$	$L = d' \in [d, D)$
Trie	In left-most leaf	K is constant	$L = d$ for all children

Questions?

Thank you!