Parallel *kd*-tree with Batch Updates

Ziyang Men

UC Riverside

Joint work with Zheqi Shen, Yan Gu and Yihan Sun

Parallel Model and I/O Model

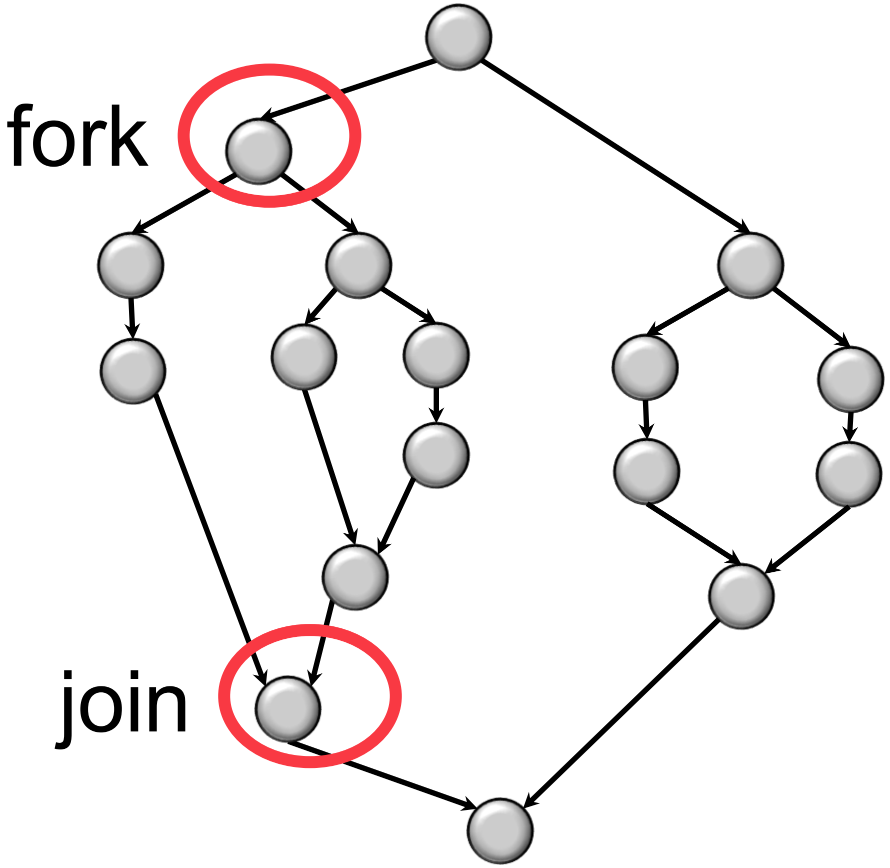
Parallel Model

Shared-memory multi-core setting, using fork-join parallelism assuming binary-forking.

Work-span model

- Work: total number of operations (sequential time).
- Span/depth: longest dependence chain (parallel time).

Work-efficient: work asymptotically the same as the best sequential algorithm.



I/O Model

Two levels

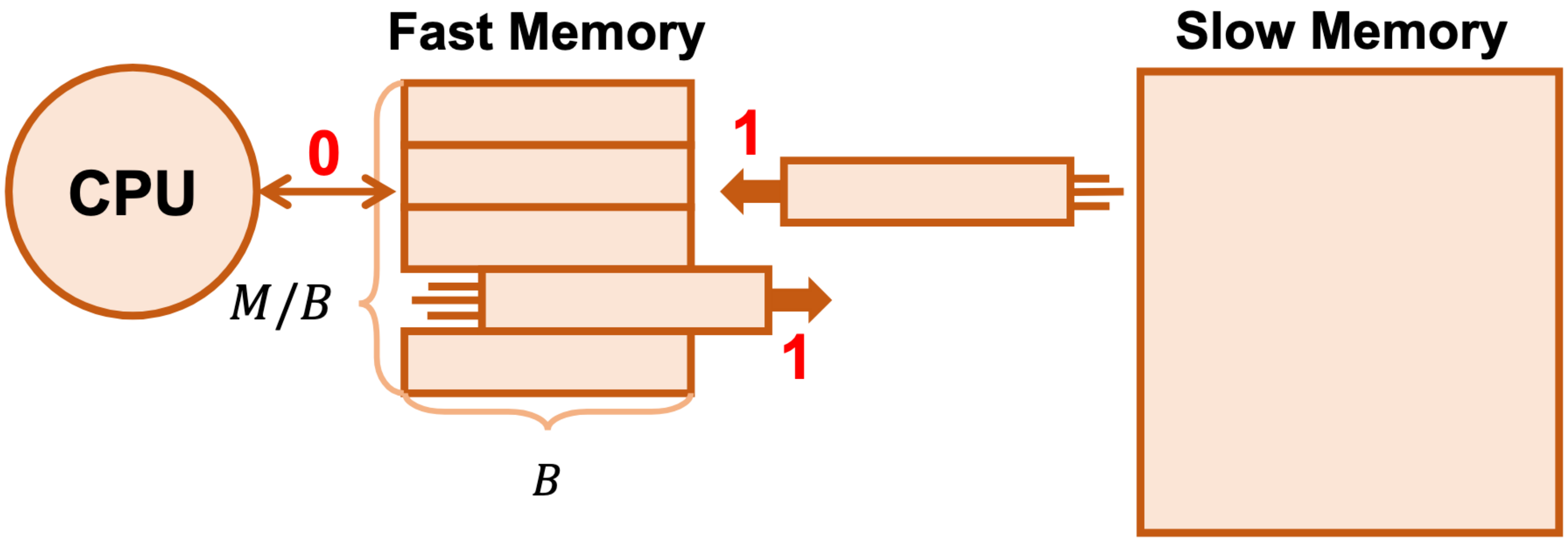
- A fast memory of fixed size M (small).
- A slow memory of unbounded size (large).

Two types of memory transfer

- Read: load a block from slow memory
- Write: write a block to the slow memory

The I/O complexity of an algorithm is:

(read transfer) + # (write transfer)



kd-tree

kd-tree

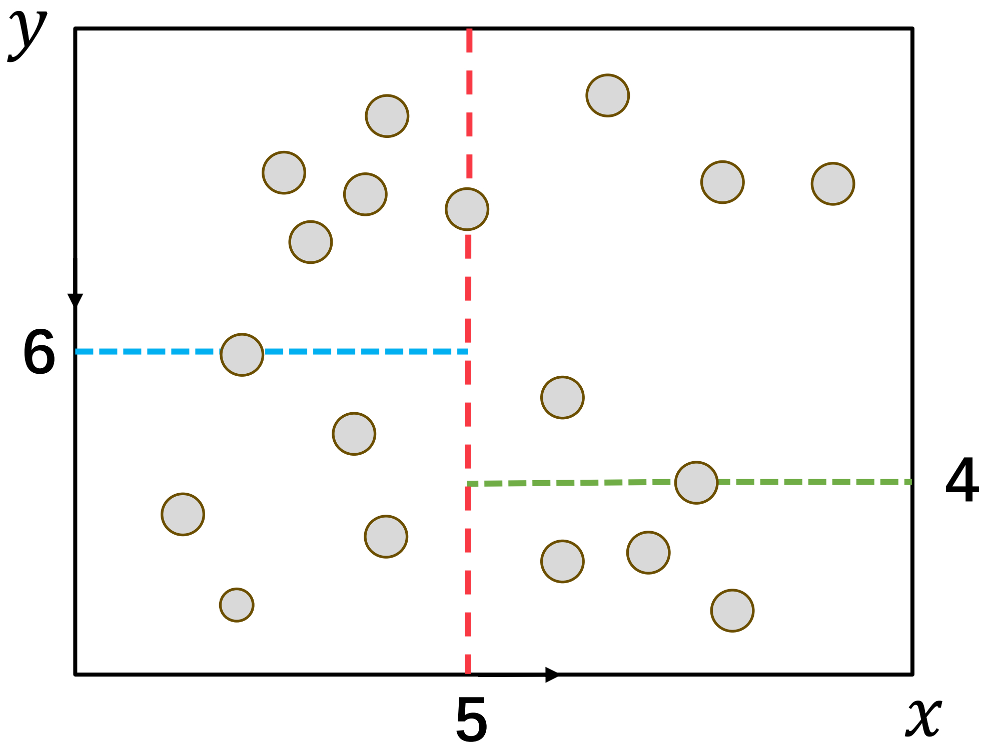
A spatial partition data structure to manage points in the geometric space.

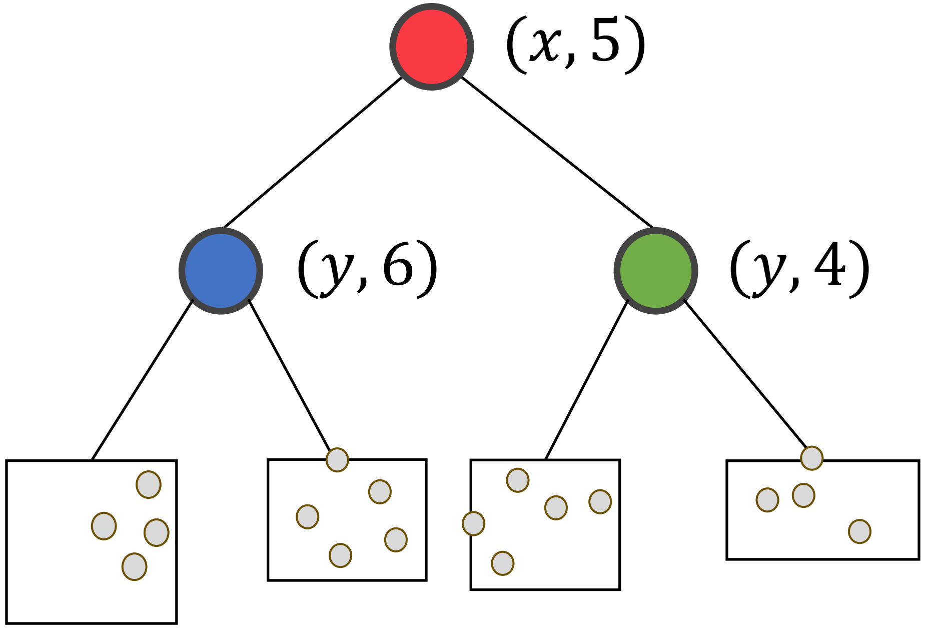
- **Recursively splits the region in the median.**

kd-tree

A spatial partition data structure to manage points in the geometric space.

- Recursively splits the region in the median.
- Every internal node represents a sub-region.
- Every leaf contain a single point.

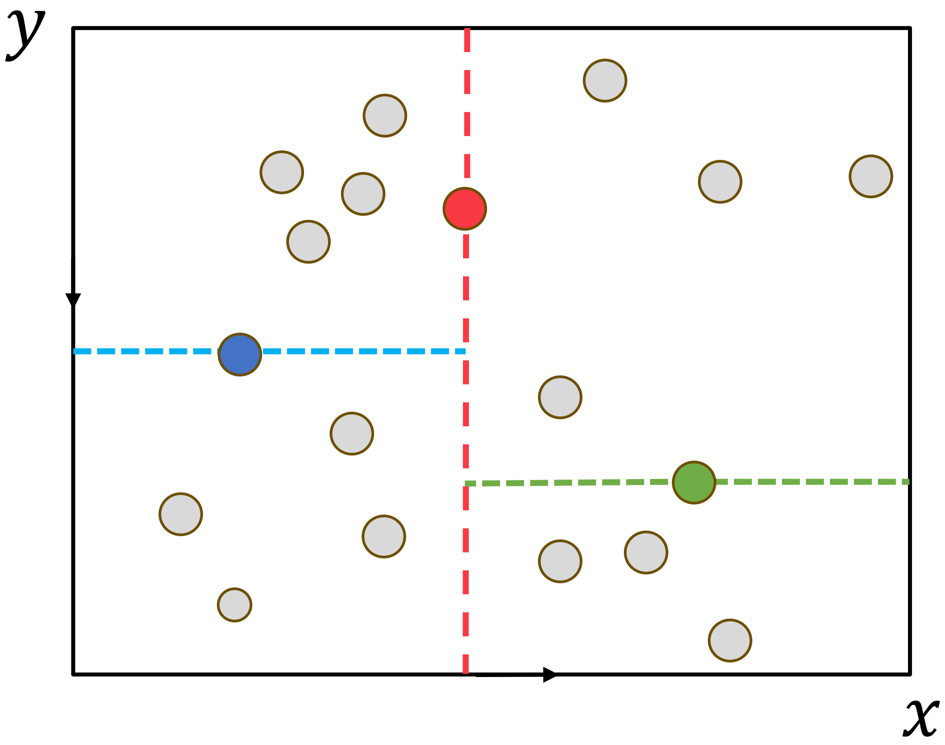




Build a *kd*-tree

1. Find the median of the inputs.
2. Partition into two parts.
3. Recursive.

$O(n \log n)$



Why *kd*-tree ?

- Linear space
- Simple algorithms
- Comparison based
    - Resistant to skewed data
- Scale to reasonably large dimension ($D \approx 10$)
- Support wide range of queries

Challenge 1: dynamic *k*d-tree

To insert or delete points from the tree.

***k*d-tree is generally considered to be a static structure.**

- **Keep it fully balanced to ensure the query efficiency.**
- **Require *rebuilding the whole* tree after updates.**

E.g. Bentely[CACM' 75], Agarwal[PODS' 16], CGAL[CGAL' 20], Bhl-tree[SIGMOD' 22] …

Alternatively, handle updates using logarithmic methods.

- **Decompose a single *k*d-tree into $O(\log n)$ static *k*d-trees.**
- **Update starts from small trees.**
- **Faster update but *slower on query*.**

E.g. Bentely[IPL' 79], Bkd-tree[SSTD' 03], Agarwal[PSCG' 03], Log-tree[SIGMOD' 22] …

Challenges:
Achieve fast update, meanwhile guarantee the query efficiency.

Bentely [CACM' 75]: Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. Commun. ACM 18, 9 (1975), 509–517.

Agarwal [PODS' 16]: Pankaj Agarwal, Kyle Fox, Kamesh Munagala, and Abhinandan Nath. 2016. Parallel algorithms for constructing range and nearest-neighbor searching data structures. In Principles of Database Systems (PODS). 429–440.

CGAL [CGAL' 20]: The CGAL Project. 2020. CGAL User and Reference Manual (5.1 ed.). CGAL Editorial Board. https://doc.cgal.org/5.1/Manual/packages.html.

Log-tree [SIGMOD' 22]: Yiqiu Wang, Shangdi Yu, Laxman Dhulipala, Yan Gu, and Julian Shun. 2022. ParGeo: a library for parallel computational geometry. In European Symposium on Algorithms (ESA).

Bentely [IPL' 79]: Jon Louis Bentley. 1979. Decomposable searching problems. Inform. Process. Lett. 8, 5 (1979), 244–251.

Bkd-tree [SSTD' 03]: Octavian Procopiuc, Pankaj K Agarwal, Lars Arge, and Jeffrey Scott Vitter. 2003. Bkd-tree: A dynamic scalable kd-tree. In International Symposium on Spatial and Temporal Databases (SSTD). Springer, 46–65.

Agarwal [PSCG' 03]: Pankaj K Agarwal,Lars Arge, Andrew Danner, and Bryan Holland-Minkley. 2003. Cache-oblivious data structures for orthogonal range searching. In Proceedings of the nineteenth annual symposium on Computational geometry. 237–245.

Bhl-tree [SIGMOD' 22]: Yiqiu Wang, Shangdi Yu, Laxman Dhulipala, Yan Gu, and Julian Shun. 2022. ParGeo: a library for parallel computational geometry. In European Symposium on Algorithms (ESA).

Challenge 2: static *k*d-tree algorithms

Even for the static *k*d-tree, we are unaware of *k*d-tree algorithm that is highly parallel and I/O efficient.

1. Find the exact median of the inputs.
2. Partition into two parts.
3. Recursive.

- Required by a fully balanced tree.
- High I/O cost.
- Enforce the algorithm proceeds level-by-level.

Prevents the algorithm to be highly parallel and I/O efficient.

Our contribution

Propose the Pkd-tree (Parallel *k*d-tree) that is highly parallel, I/O-efficient, and can support efficient updates.

1. Build a slightly unbalanced tree, achieve a construction algorithm that *optimizes* work, span and I/O complexity.
    - Height: $\log n + O(1)$,
    - Not affect the existing query bound.
2. A *reconstruction-based* update algorithm that guarantees the tree to be weight-balanced.
3. A highly *efficient* parallel implementation.

Close the long-standing gap between the *wide usage of kd-trees* and *lack of a highly efficient parallel implementation*.

Parallel tree construction

Parallel *kd*-tree

Serial construction

1. Find the median
2. Partition points
3. Recursive

$O(n\log n)$

Does it look familiar?

- Quick Sort!
- Sorting time is a lower bound for *k*d-tree construction.

The plain parallel algorithm

Parallel median

Parallel partition

Parallel recursive

$O(n\log n)$ work ✓

$O(\log^2 n)$ span ✓

$O(\frac{n}{B}\log n)$ I/O complexity

- Sorting is $O(\frac{n}{B}\log_M n)$.
- I/O inefficient!

E.g., for OSM [PC’08] with 1.2 billion points:

- The plain parallel algorithm: 56.6s
- Ours: 5.08s

OSM [PC’08]: Mordechai Haklay and Patrick Weber. 2008. Openstreetmap: User-generated street maps. IEEE Pervasive computing 7, 4 (2008), 12–18.

Observations

The plain parallel algorithm

Parallel median
Parallel partition
Parallel recursive

1. Build a fully balanced tree.
    - $\log n$ tree height, ensure the query bound.
    - Find the exact median (high cost).

2. Build one level at once.
    - Find median and partition in each level.
    - More I/O needed.

Our ideas

Convention

1. Build a fully balanced tree
- $\log n$ tree height
- Find the exact median

2. Build one level at once
- Find median and partition in each level
- More I/O needed

Our ideas

1. A slightly *unbalanced* tree.
- $\log n + O(1)$ tree height.
- Use *samples* to estimate the median.

2. Build *multiple* levels at once
- One round of data movement is sufficient.
- I/O efficient *points sieving algorithm* to partition points.

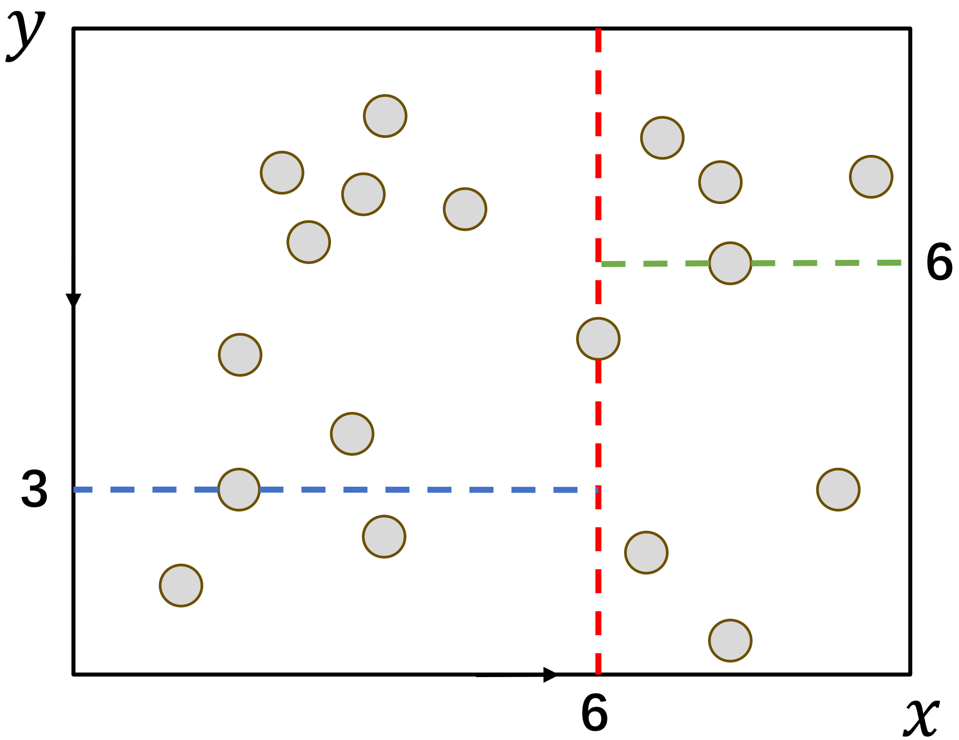
Our parallel tree construction (sketch)

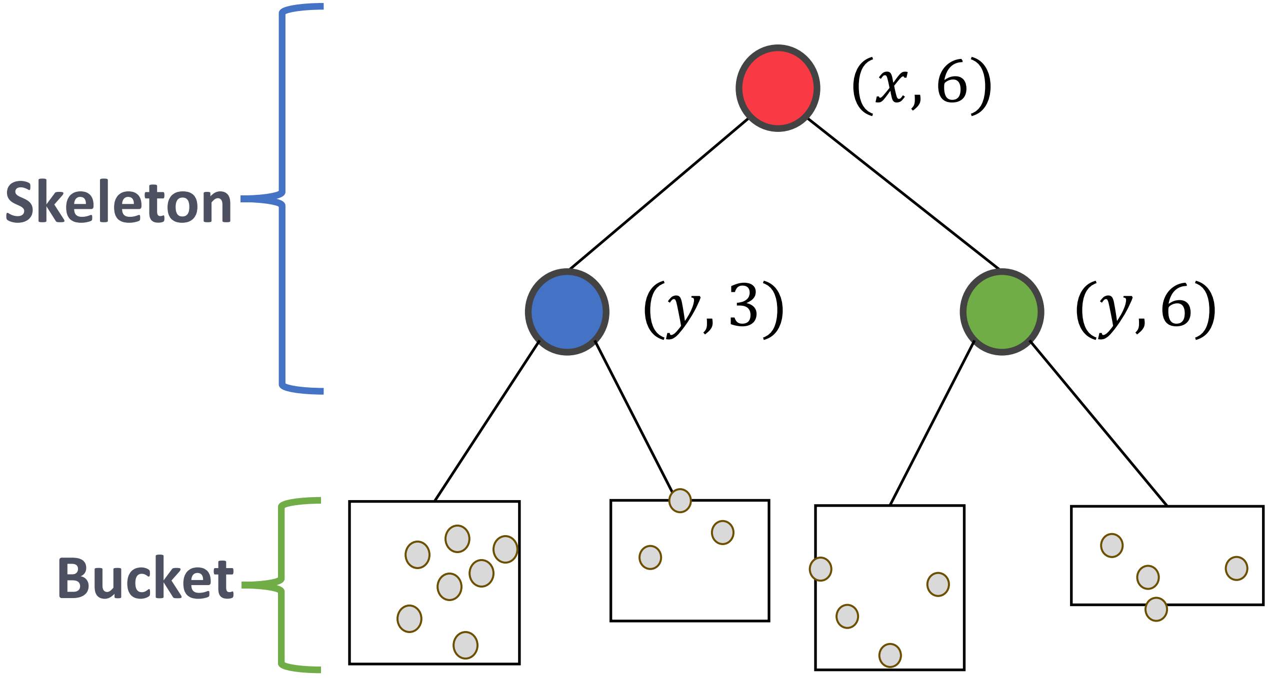
1. Take sufficient samples.
2. Pick the median from samples.
3. Construct multiple levels of tree at once.
4. Sieving points to the corresponding sub-trees.
5. Parallel recurse.

Sampling and build multiple levels

1. Take sufficient samples.
2. Pick the median in samples.
3. Build multiple levels at once.

Serial





Sample size

1. Take sufficient samples.
2. Pick the median in samples.
3. Build multiple levels at once.

Serial

Intuition

- Less samples → faster, may break the tree height bound.
- More samples → slower, yet better tree height guarantee.

Lemma 3.2

Given over-sampling rate σ, the total height of Pkd-tree with size n is

- $O(\log n)$ if $\sigma = \Omega(\log n)$,
- $\log n + O(1)$ if $\sigma = \Omega\left(\log^3 n\right)$.

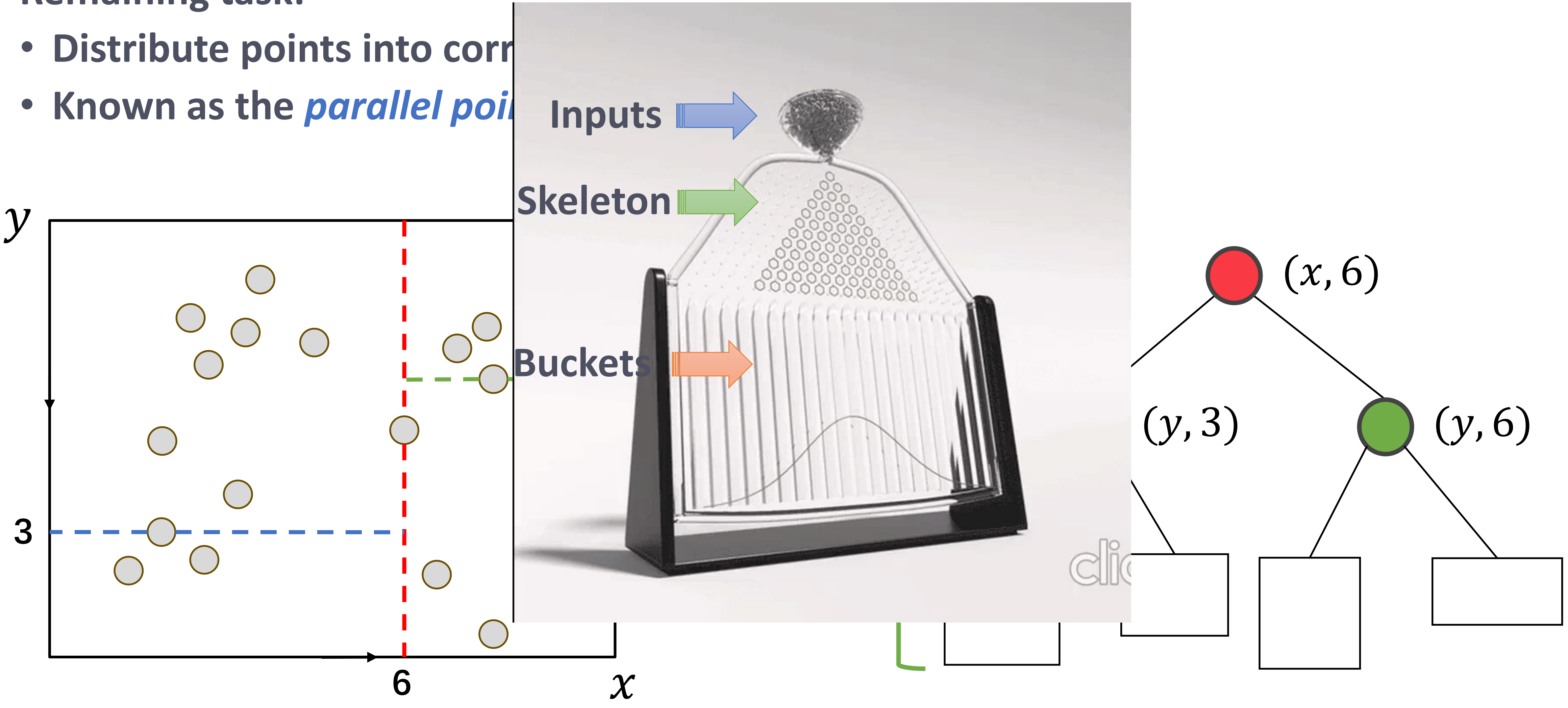
Our parallel tree construction

1. Take sufficient samples.
2. Pick the median from samples. ☑
3. Construct multiple levels of tree at once.
4. Sieving points to the corresponding sub-trees.
5. Parallel recurse.

Parallel Points Sieving

Remaining task:

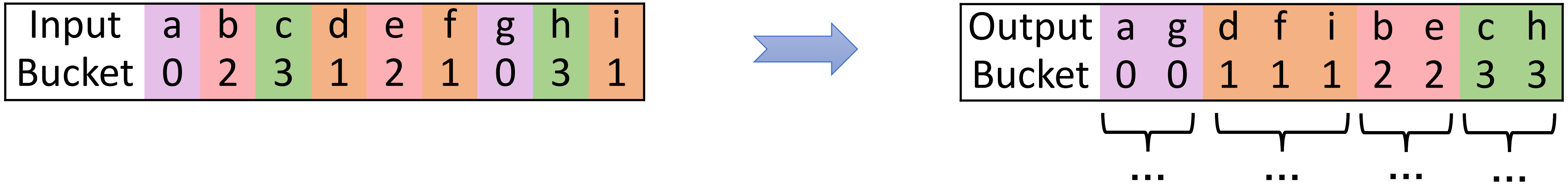
- Distribute points into corr
- Known as the ***parallel poi***



Parallel points sieving

Formally, we want to

- **rearrange the input array to make points in same bucket being contiguous**



Input	a	b	c	d	e	f	g	h	i
Bucket	0	2	3	1	2	1	0	3	1

Output	a	g	d	f	i	b	e	c	h
Bucket	0	0	1	1	1	2	2	3	3

Trivial in the serial setting.

Parallel is hard.

- **Avoid data race, e.g., avoid write two points to same position at same time.**
- **Keep I/O efficiency, e.g., write every point directly to its destination.**

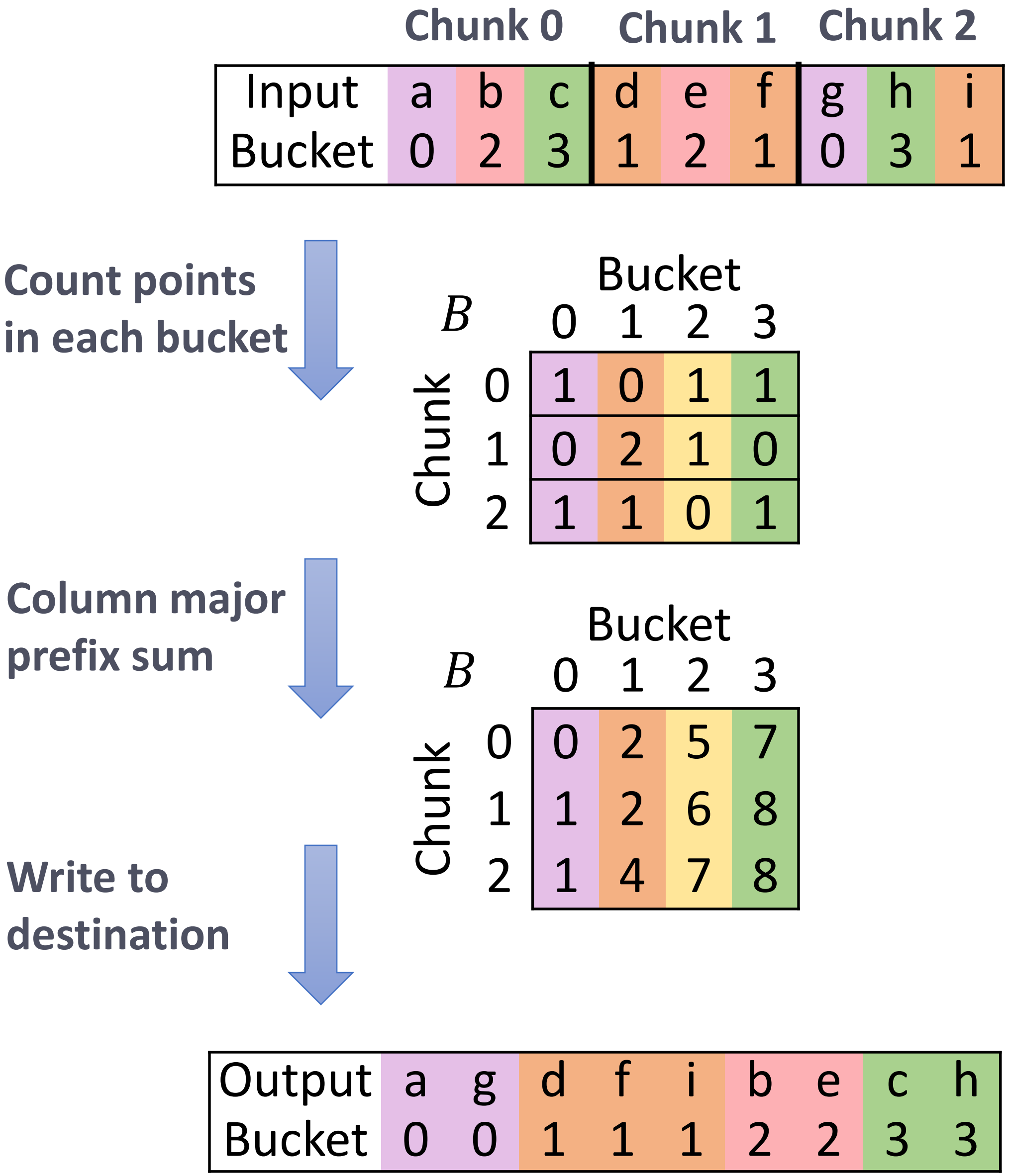
We borrow ideas from the *cache-efficient* parallel sorting [SPAA' 20].

Cache-efficient sieving [SPAA'23]: Guy E. Blelloch, Phillip B. Gibbons, and Harsha Vardhan Simhadri. 2010. Low depth cache-oblivious algorithms. In ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)..

Parallel points sieving

High-level ideas

1. **Divide the input array into chunks.**
2. **Parallel for each chunk:**
   - Count number of points in every bucket.
   - Store answer in array B.
3. **Perform column major prefix sum on B.**
4. **Parallel for each chunk:**
   - Write each point to the destination.



	Chunk 0			Chunk 1			Chunk 2		
Input	a	b	c	d	e	f	g	h	i
Bucket	0	2	3	1	2	1	0	3	1

Count points in each bucket

B		Bucket			
		0	1	2	3
Chunk	0	1	0	1	1
	1	0	2	1	0
	2	1	1	0	1

Column major prefix sum

B		Bucket			
		0	1	2	3
Chunk	0	0	2	5	7
	1	1	2	6	8
	2	1	4	7	8

Write to destination

Output	a	g	d	f	i	b	e	c	h
Bucket	0	0	1	1	1	2	2	3	3

Our parallel tree construction

1. Take sufficient samples.
2. Pick the median from samples.
3. Construct multiple levels of tree at once.
4. Sieving points to the corresponding sub-trees.
5. Parallel recurse.

Tree height:
$\log n + O(1)$

Tree construction	The plain parallel algorithm	Ours (w.h.p)
Work	$O(n\log n)$	$O(n\log n)$
Span	$O(\log^2 n)$	$O(M^{\epsilon}\log_M n)$
I/O	$O(\frac{n}{B}\log n)$	$O\left(\frac{n}{B}\log_M n\right)$

Batch updates

Batch updates

Insert batch points to leaves / remove batch points from leaves

Parallel kd-tree	Layout	Balancing
CGAL [CGAL' 20]	Single	Fully balanced
BHL-tree [SIGMOD' 22]	Single	Fully balanced
Log-tree [SIGMOD' 22]	Logarithmic method	-

Slow update

Slow query

→

Can we achieve fast update & fast query?

Batch updates

Insert batch points to leaves / remove batch points from leaves

Parallel kd-tree	Layout	Balancing
CGAL [CGAL' 20]	Single	Fully balanced
BHL-tree [SIGMOD' 22]	Single	Fully balanced
Log-tree [SIGMOD' 22]	Logarithmic method	-
Pkd-tree	**Single**	**Weight-balanced**

Slow update

Slow query

Fast query

Fast update

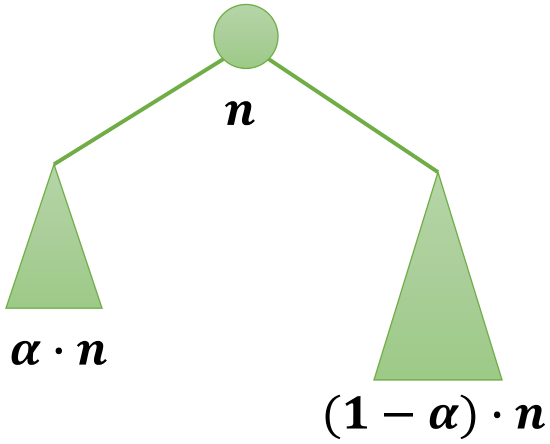
Can we achieve fast update & fast query?

Weight-balanced scheme

Invariant:

Two subtrees can be off from perfectly balanced by a factor of $\alpha, 0 < \alpha < 1$.

imbalance ratio



During batch updates, compared with a fully balanced tree:

- Less rebuilds needed 😇
- May reduce the query efficiency ☹

The behavior is controlled by the imbalance ratio α, more flexible!

Handling of imbalance

Some weight-balanced trees

- **can use *rotate* to re-balance in the amortized constant time.**
- **e.g., AVL-tree, Red-black tree.**

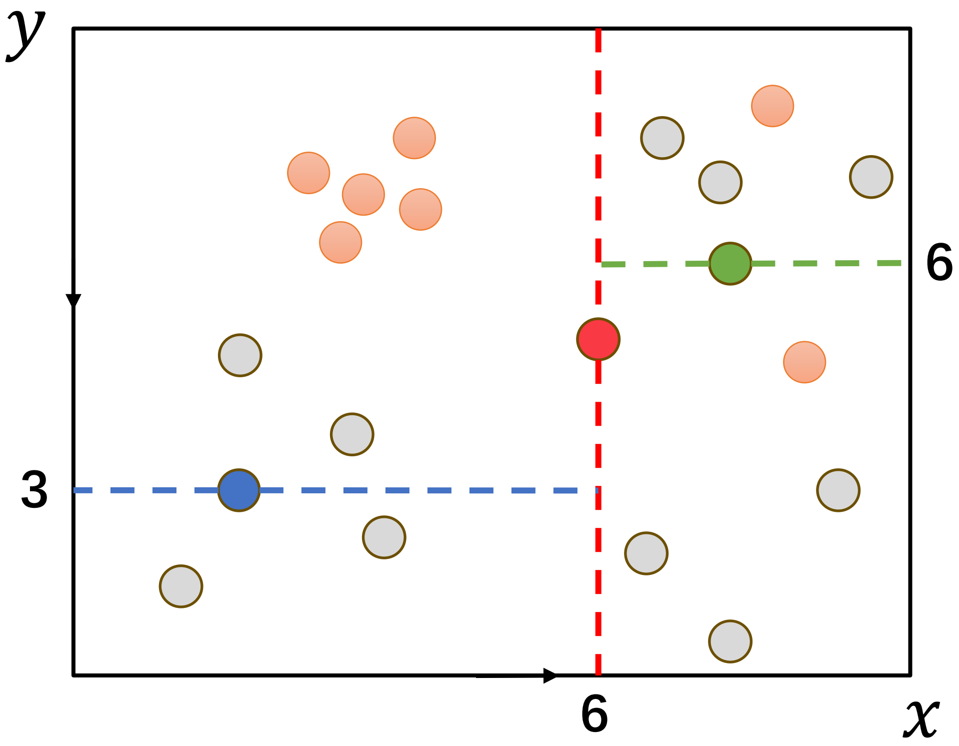
***k*d-tree *does not* support rotation, as tree nodes represent nested regions.**

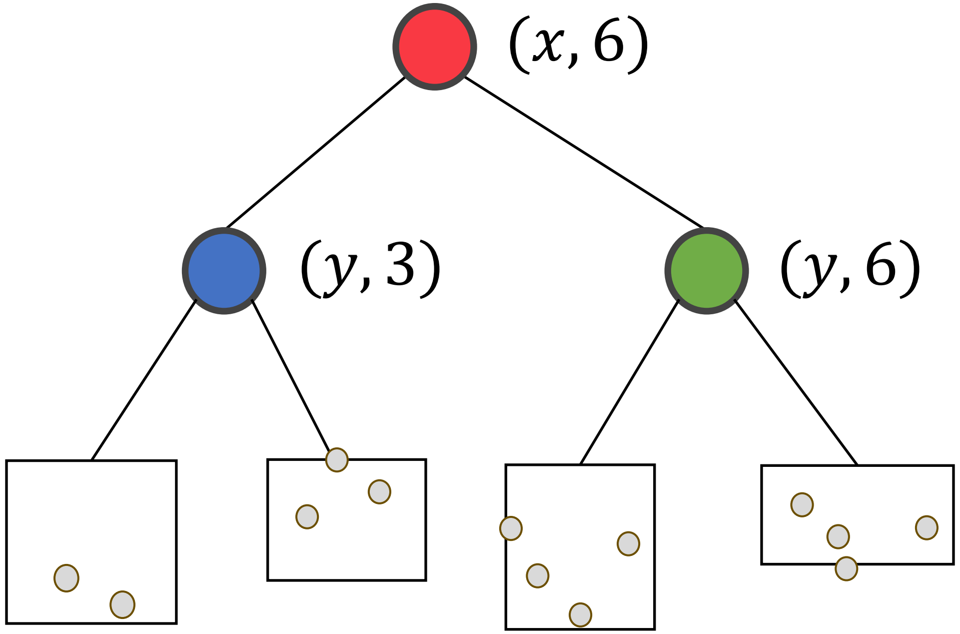
- **Instead rebuild the whole tree, one can only rebuild the imbalanced sub-tree.**
- **Known as *partial rebuild*** [MHO’ 1983].
- **Drawback: still needs to perform rebuild after each updates.**

Our idea:
Use the *partial rebuild* scheme to maintain a *weight-balanced* tree.

Partial rebuild [MHO’ 75]: Mark H Overmars. 1983. The design of dynamic data structures. Vol. 156. Springer Science & Business Media.

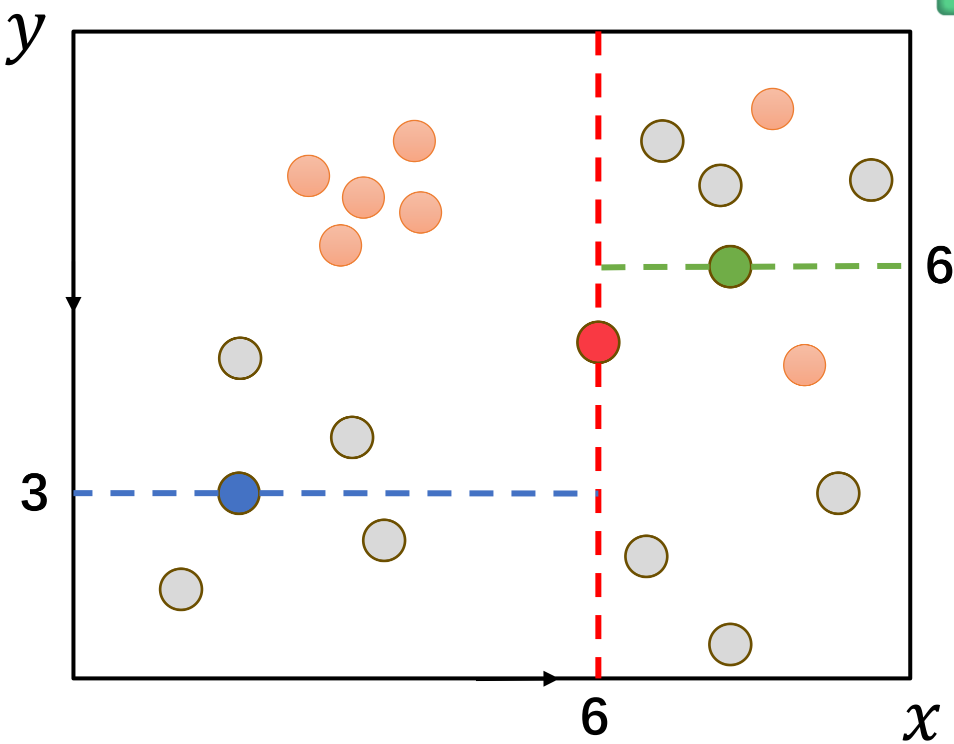
Batch insert

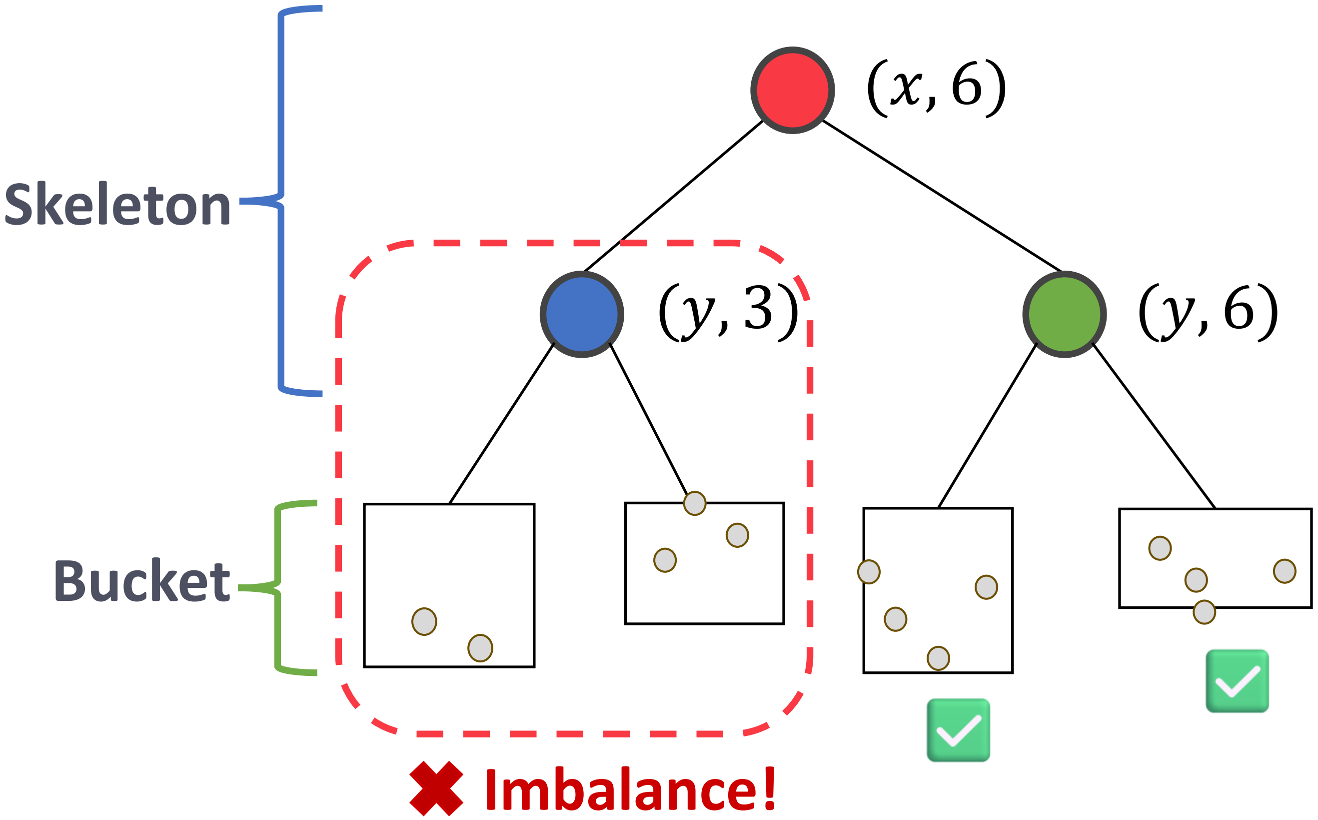




Batch insert

1. Fetch the skeleton.
2. Sieve the insertion points to the bucket.
3. In parallel:
   - Rebuild imbalanced sub-trees; ✗
   - Recurse on other buckets. ✓





Batch updates bounds

1. Fetch the skeleton.
2. Sieve the insertion points to the bucket.
3. In parallel:
    - Rebuild imbalanced sub-trees; // Amortized to tree nodes visit
    - Recurse on other buckets.

Using $m = O(n)$ the batch size

- $O(\log^2 n)$ span,
- $O(\log^2 n)$ work per element,
- $O(\log\left(\frac{n}{m}\right) + (\log n \log_M n)/B)$, I/O cost per element.

Experiments

Implementation

Reduce the memory usage of tree

- **Remove everything that is unnecessary, i.e., the *bounding box.***
- **Query needs to take a bounding box from root and compute new ones on-the-fly.**
- **Drawback: less prune efficiency when dimension is high.**

We use standard query algorithms for single *k*d-trees.

Parameters

- **Build 6 levels at once.**
- **Allow the sub-trees' weight to be off by at most 30% ($\alpha = 0.3$).**

Experiments

Setting

- Use machine with 96 Cores and 1.5 TB RAM.
- Implemented using C++, using ParlayLib [SPAA' 20] for parallelism.

Benchmarks

- Uniform: points are uniformly distributed within a cube.
- Varden: points have very skewed distribution.
- Real-word graphs: scaling to dimension 10 and billions size.

Parlay [SPAA' 20]: Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. 2020. ParlayLib — a toolkit for parallel algorithms on shared-memory multicore machines. In ACM Symposium on Parallelism in Algorithms and Architectures (SPAA). 507–509.

Baselines

Parallel kd-tree	Layout	Construction	Batch Insert	Batch Delete
CGAL [CGAL' 20]	Single	Plain	Rebuild whole tree	Serial Delete
BHL-tree [SIGMOD' 22]	Single	Plain	Rebuild whole tree	Rebuild whole tree
Log-tree [SIGMOD' 22]	Logarithmic method	Plain	Merge and rebuild	Merge and rebuild
Pkd-tree	Single	I/O efficient	Partial reconstruction	Partial reconstruction

---

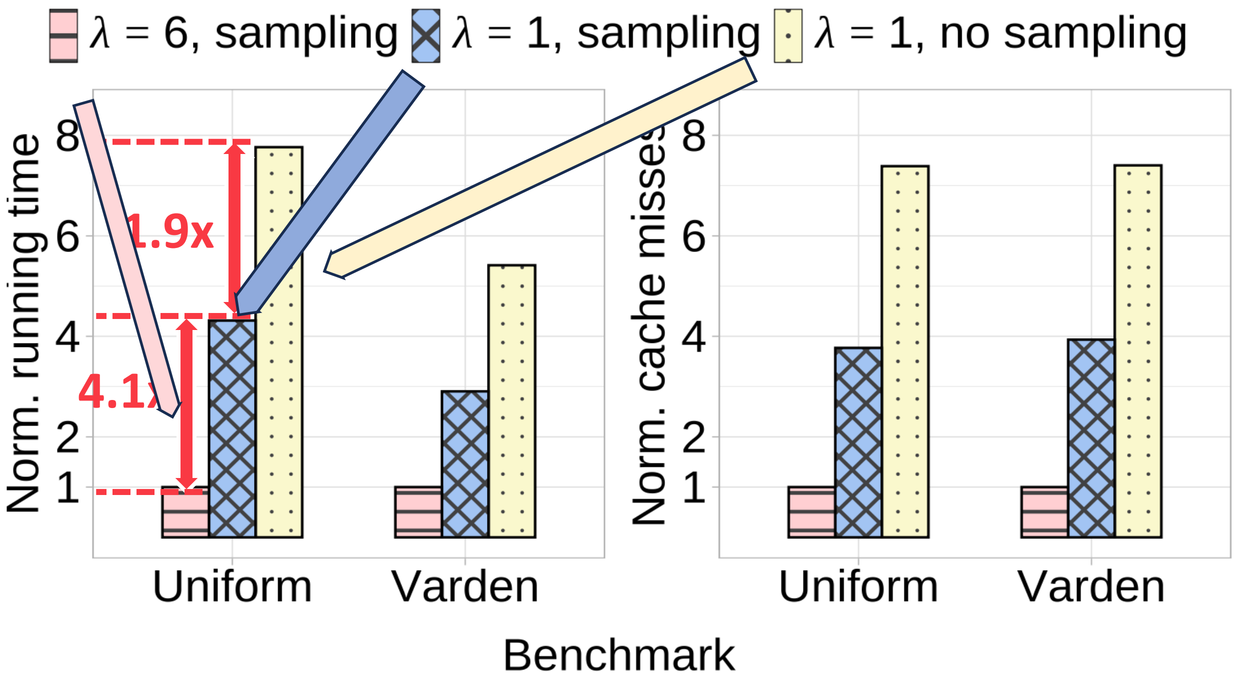
CGAL [CGAL' 20]: The CGAL Project. 2020. CGAL User and Reference Manual (5.1 ed.). CGAL Editorial Board. https://doc.cgal.org/5.1/Manual/packages.html.

Log-tree [SIGMOD' 22]: Yiqiu Wang, Shangdi Yu, Laxman Dhulipala, Yan Gu, and Julian Shun. 2022. ParGeo: a library for parallel computational geometry. In European Symposium on Algorithms (ESA).

Bhl-tree [SIGMOD' 22]: Yiqiu Wang, Shangdi Yu, Laxman Dhulipala, Yan Gu, and Julian Shun. 2022. ParGeo: a library for parallel computational geometry. In European Symposium on Algorithms (ESA).

Tree construction

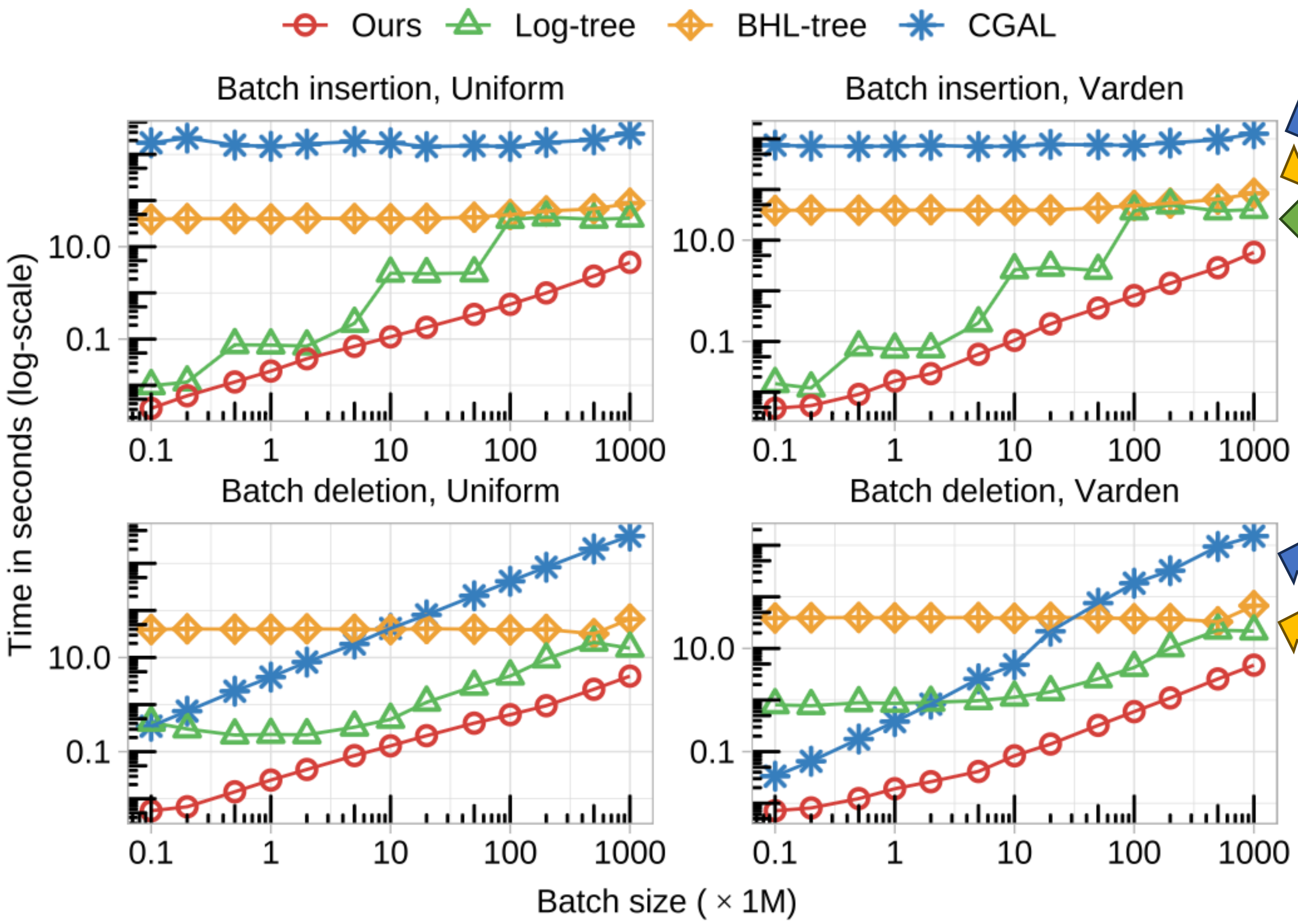
Bench.	*D*	Construction 2	3	5	9
Uniform 1000M	Ours	3.15	3.65	5.67	9.66
	Log-tree	37.9	45.4	58.0	92.7
	BHL-tree	31.7	40.5	58.4	104
	CGAL	1147	1079	1217	1412
Varden 1000M	Ours	3.66	4.78	6.27	11.2
	Log-tree	34.2	41.8	57.8	92.6
	BHL-tree	30.2	39.2	58.7	104
	CGAL	429	390	372	438



Magnitudes faster

Cache optimization is important

Batch update



Batch Insertion

Slow, rebuild the whole tree

Rebuild large buffer trees incurs more time → line jumps

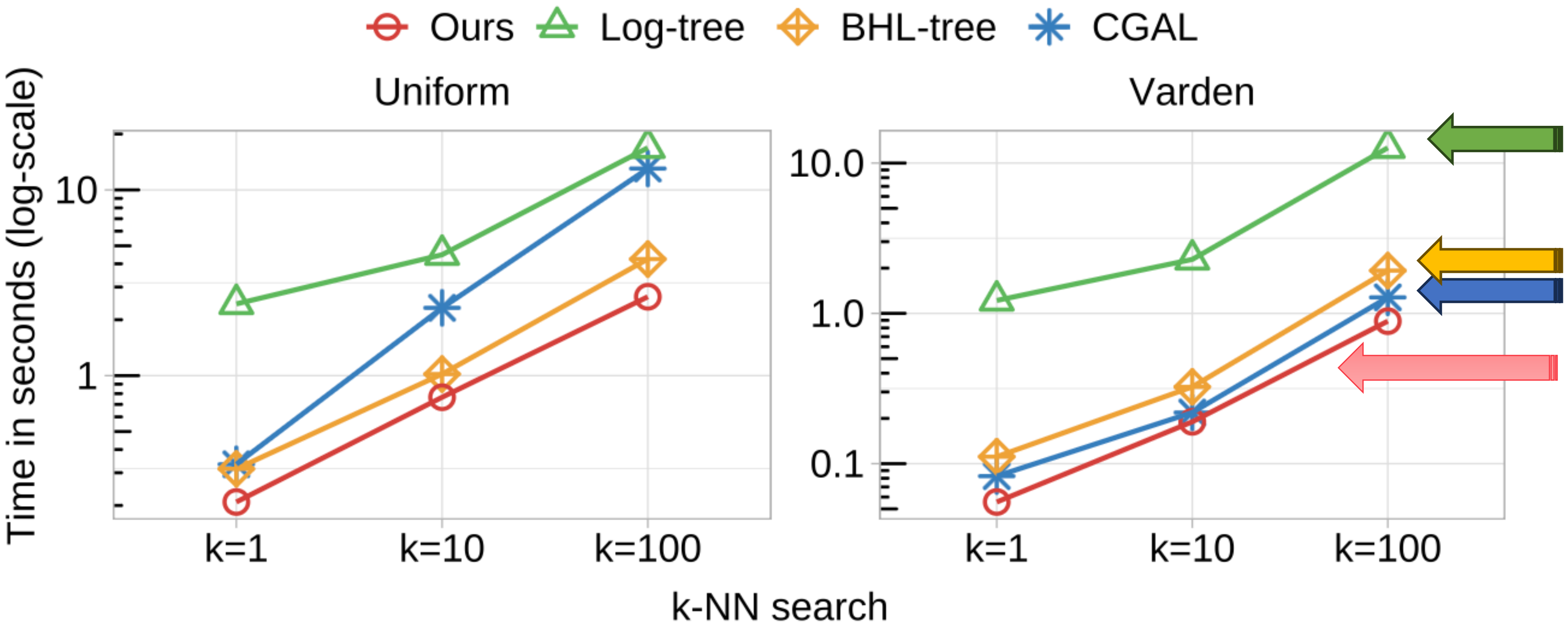
Batch Deletion

Serial deletion, slower for large batch

Rebuild the whole tree, stable

Tree size is 1000M, lower is better

KNN



Query $O(\log n)$ trees, slow.

Single tree, similar time.

Slightly faster due to smaller memory footprint.

Tree size is 1000M, query size is 10M. Lower is better

Imbalance

How the imbalance ratio α impact the update time and query time

- Smaller $\alpha \rightarrow$ more tolerance of imbalance, less rebuild time and slower query;
- Larger $\alpha \rightarrow$ more balance required, more rebuild time and faster query;

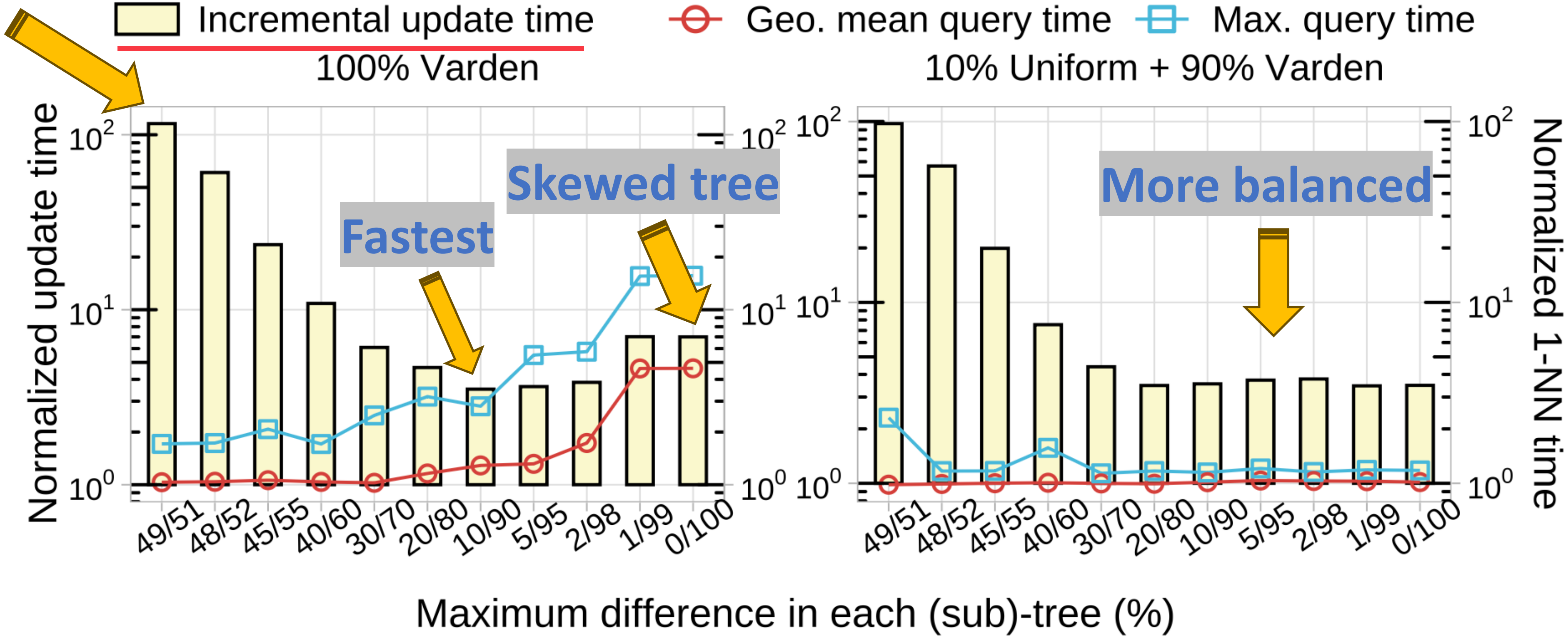
Design of experiments

- Construct a tree by inserting 1000 batches one-by-one. Batch size is 1M.
- Perform 10-NN query after each insertion.
- Test for two distributions: the skewed one and the uniform.

Imbalance – Updates

How the imbalance ratio α impact the update time and query time

- Smaller α → query more balance required, more rebuild time and faster query;
- Larger α → more tolerance of imbalance, less rebuild time and slower;



Frequent rebuild

Fastest

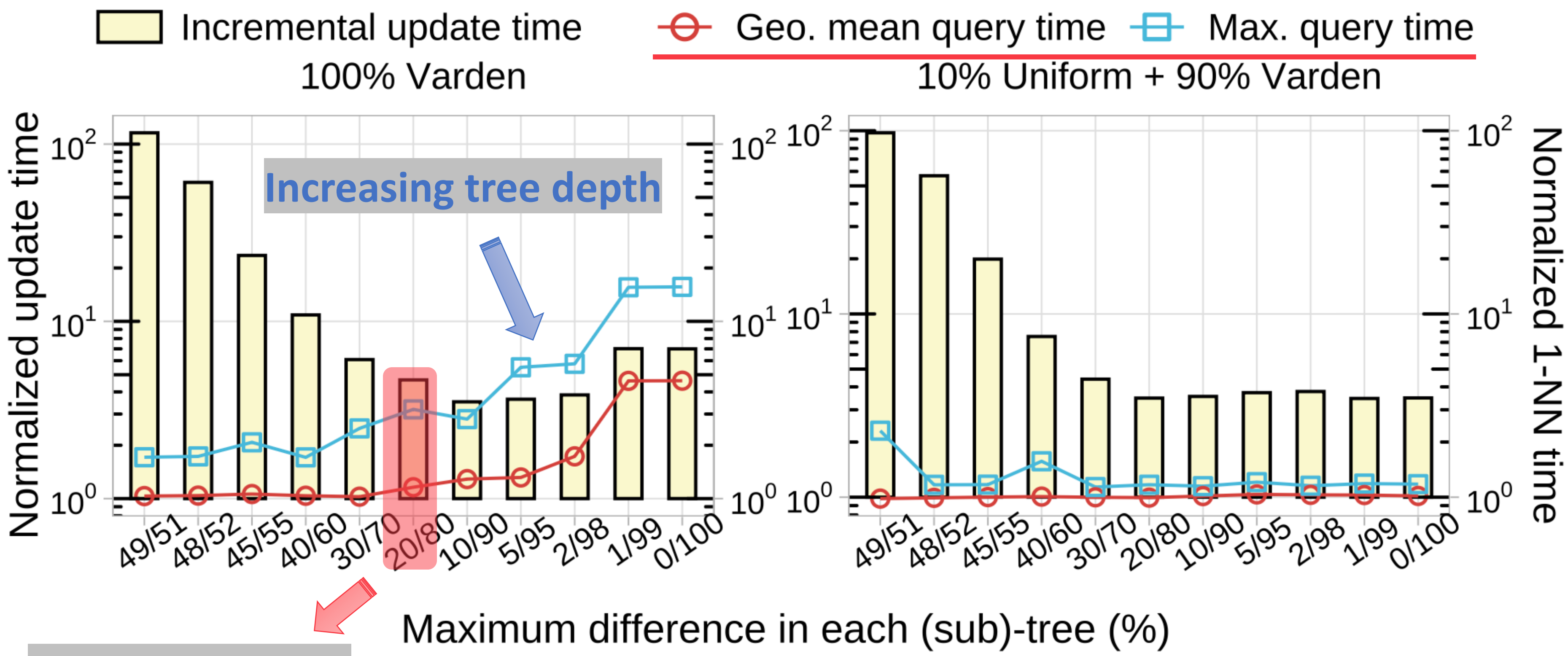
Skewed tree

More balanced

Imbalance - Queries

How the imbalance ratio α impact the update time and query time

- Smaller α → query more balance required, more rebuild time and faster query;
- Larger α → more tolerance of imbalance, less rebuild time and slower;



Increasing tree depth

Best compromise

Summary

The Pkd-tree is a parallel kd-tree that provides both

- strong theoretical guarantee,
- high efficient practical performance.

Tree construction

- Use samples to find the median.
- Build multiple levels at once.
- Cache-efficient points sieving algorithm.
- Ensure tree height to be $\log n + O(1)$.

Batch updates

- Weight-balanced scheme.
- Reconstruct the imbalanced sub-trees.

Paper

Code