Optimal Batch-Dynamic kd-trees for Processing-in-Memory with Applications

Yiwei Zhao
Carnegie Mellon University
Pittsburgh, PA, USA
yiweiz3@andrew.cmu.edu

Hongbo Kang
Tsinghua University
Beijing, China
khb20@mails.tsinghua.edu.cn

Yan Gu
University of California, Riverside
Riverside, CA, USA
ygu@cs.ucr.edu

Guy E. Blelloch
Carnegie Mellon University
Pittsburgh, PA, USA
guyb@cs.cmu.edu

Laxman Dhulipala
University of Maryland
College Park, MD, USA
laxman@umd.edu

Charles McGuffey
Reed College
Portland, OR, USA
cmcguffey@reed.edu

Phillip B. Gibbons
Carnegie Mellon University
Pittsburgh, PA, USA
gibbons@cs.cmu.edu

ABSTRACT

The kd-tree is a widely used data structure for managing multi-dimensional data. However, most existing kd-tree designs suffer from the memory wall—bottlenecked by off-chip memory latency and bandwidth limitations. Processing-in-memory (PIM), an emerging architectural paradigm, offers a promising solution to this issue by integrating processors (PIM cores) inside memory modules and offloading computational tasks to these PIM cores. This approach enables low-latency on-chip memory access and provides bandwidth that scales with the number of PIM modules, significantly reducing off-chip memory traffic.

This paper introduces PIM-kd-tree, the first theoretically grounded kd-tree design specifically tailored for PIM systems. The PIM-kd-tree is built upon a novel log-star tree decomposition that leverages local intra-component caching. In conjunction with other innovative techniques, including approximate counters with low overhead for updates, delayed updates for load balancing, and other PIM-friendly aspects, the PIM-kd-tree supports highly efficient *batch-parallel* construction, point searches, dynamic updates, orthogonal range queries, and kNN searches. Notably, all these operations are work-efficient and load-balanced even under adversarial skew, and incur only $O(\log^* P)$ communication overhead (off-chip memory traffic) per query. Furthermore, we prove that our data structure achieves *whp* an optimal trade-off between communication, space, and batch size. Finally, we present efficient parallel algorithms for two prominent clustering problems, density peak clustering and DBSCAN, utilizing the PIM-kd-tree and its techniques.

CCS CONCEPTS

• **Theory of computation** → **Parallel algorithms**; **Distributed algorithms**; • **Computer systems organization** → *Heterogeneous (hybrid) systems*; *Parallel architectures*.

SPAA '25, July 28–August 1, 2025, Portland, OR, USA

ACM ISBN 979-8-4007-1258-6/25/07
https://doi.org/10.1145/3694906.3743318

KEYWORDS

space-partitioning index, kd-tree, processing-in-memory, near-data-processing, nearest neighbor search, batch-dynamic algorithms

ACM Reference Format:
Yiwei Zhao, Hongbo Kang, Yan Gu, Guy E. Blelloch, Laxman Dhulipala, Charles McGuffey, and Phillip B. Gibbons. 2025. Optimal Batch-Dynamic kd-trees for Processing-in-Memory with Applications. In *37th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '25), July 28–August 1, 2025, Portland, OR, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3694906.3743318

1 INTRODUCTION

The kd-tree (k-dimensional tree) is one of the most widely used space-partitioning data structures for managing multi-dimensional data. It organizes a set of D-dimensional points[1] using a search tree, and supports a variety of query types, including point search, orthogonal range query, and k nearest neighbor (kNN) search. Beyond search tasks, kd-trees are also fundamental components in numerous clustering problems in computational geometry. The kd-tree supports low-to-medium-dimensional applications efficiently ($D < 15$ in practice), offering linear space consumption, predictable tail latency for many queries (due to its balanced tree structure), simple algorithms for queries, and adaptability to dynamic updates. Since its invention in the 1970s [10], kd-trees have been widely adopted in academia and industry. Applications span computational geometry [25, 53, 76, 81, 97], machine learning [12, 44, 90, 103, 110], computer graphics [28, 58, 67, 111], radars and robotics [20, 64, 78, 88, 107], and scientific simulations [13, 23, 47, 85].

However, as data volumes have grown substantially in recent decades, kd-trees have increasingly suffered from the ***memory wall*** problem. The widening gap between (fast growing) computation speed and (slowly improving) memory access speed has made off-chip *data movement* the dominant cost and primary bottleneck in modern systems. While multi-level caching can mitigate this issue, its effectiveness is limited to workloads with high data locality whose workloads fit within the cache hierarchy.

[1] Based on the original terminology in [10], a kd-tree maintains k-dimensional data. However, to avoid overloading k in the more frequently-used k nearest neighbor (kNN) scenario, we denote the number of dimensions as D in this paper, and use k in the scenario of kNN or clustering problems.

Processing-in-memory (PIM), also known as *near-data-processing (NDP)*, has recently reemerged as a promising architectural and hardware solution to the memory wall problem. By embedding computational units (PIM cores) inside memory modules, PIM enables computation to be performed where the data reside. The traditional von Neumann architecture allows only one type of data movement, where data must be fetched to the CPU from the memory over off-chip channels. In contrast, PIM-based systems also enable computation to be offloaded to PIM cores. This approach enhances both performance and energy by exploiting the low-latency and low-energy on-chip memory accesses of the PIM cores, while also benefiting from on-chip memory bandwidth and computational capacity that scales with the number of PIM modules. Off-chip communication (i.e., off-chip data movement) can be significantly reduced by utilizing on-chip memory accesses and local processing on PIM cores. See §2.1 for additional background on PIM.

This paper focuses on the design and analysis of a PIM-friendly batch-parallel, dynamic (i.e., batch-dynamic) kd-tree with strong asymptotic guarantees. Following prior work on PIM [55, 57], we analyze algorithms using the Processing-in-Memory (PIM) Model [54].

Adapting today's shared-memory kd-trees to PIM systems poses several challenges. First, PIM-friendly data structures must minimize off-chip communication. Second, these PIM data structures must maintain a good load balance (both for computation and communication) across all PIM modules. Because PIM systems typically operate in bulk-synchronous parallel (BSP) rounds [100], it is crucial to avoid stragglers that dominate the execution time in each round. However, achieving load balance under adversarial skew is nontrivial. It requires either (i) dividing tasks and data into very fine granularity (thereby increasing off-chip communication) or (ii) replicating data on multiple PIM modules (adding additional space overhead). This tension highlights a fundamental challenge in PIM data structure design—**(C0)** how to achieve a good **trade-off** between load balance, reduced off-chip communication, and low space consumption?

In addition to (C0), other challenges unique to kd-trees include:
(C1) Communication Overhead in Searches: Operations on a kd-tree storing n data points typically incur a communication overhead of $O(\log n)$ rather than the ideal communication overhead of $O(1)$. This overhead arises primarily from the $O(\log n)$-length search path within the search tree: $O(\log n)$ communication per top-down search query and $O(k \log n)$ per kNN search query.
(C2) Dynamic Update Overhead (of Auxiliary Structures): kd-trees augment each node with auxiliary metadata, such as a counter for subtree size. However, maintaining these metadata directly during dynamic updates introduces additional overhead, particularly in terms of off-chip communication costs in the PIM setting.
(C3) Lack of Summarization for Node Properties: To achieve low costs for kd-tree operations on PIM, it is beneficial to construct descriptive summaries for each node, which capture information such as positional data and local neighborhood geometry. However, the semi-balanced structure of kd-trees (see §2.2) presents unique challenges. Existing summarization techniques either fail to accommodate imbalanced tree shapes [54, 55] or impose excessive costs in communication or space [57].

We address the aforementioned challenges through the design of the **PIM-kd-tree**, a PIM-friendly batch-dynamic kd-tree. Our approach begins with a novel *log-star decomposition*, which partitions the kd-tree into $O(\log^* P)$ groups to mitigate (C1). Within each group, we construct intra-group caching on each node, storing metadata about other nodes in the same group to reduce off-chip communication during search operations. To tackle (C3) while maintaining low update and space overhead, we introduce a node categorization scheme based on subtree size, i.e., the number of descendants of each node. This categorization facilitates load-balanced decomposition even under imbalanced tree shapes. To support this and also to address (C2), we design a PIM-friendly approximate counter to maintain subtree sizes. These counters are lightweight, are updated infrequently, and are sufficiently accurate to ensure $O(\log n)$ tree height *whp*[2]. In combination with other techniques (push-pull search [55], delayed updates and efficient constructions), our design achieves a good trade-off between skew resistance, reduced communication, and space consumption; thereby addressing the core challenge (C0).

In summary, our main contribution is the design and analysis of PIM-kd-tree, which supports efficient construction, point queries (LeafSearch), dynamic updates (Insert and Delete), orthogonal range queries and (approximate) kNN on a PIM system. The cost bounds[3] of these operations can be found in Table 1. In addition, we prove a lower bound for the trade-off between search communication cost, space and batch size in the PIM setting in Theorem 5.1 and show that the cost bounds of our PIM-kd-tree are ***optimal*** *whp*. Our lower bound proof relies on the recent lower bounds for cell-probe dynamic succinct dictionaries [65]. Furthermore, we present PIM-friendly algorithms for two commonly-occurring clustering problems: density peak clustering (DPC) [84] and DBSCAN [31], which utilize the PIM-kd-tree and its techniques. All of our algorithms and data structures have the following properties:

- **Work-Efficient**: The sum of the CPU and PIM work asymptotically matches the CPU work of the best prior shared-memory designs.
- **PIM-Offloaded**: The majority of the work is offloaded to the PIM modules. The CPU has asymptotically less work than shared-memory designs.
- **Highly-Parallel** and **Skew-Resistant**: Our design has low (polylogarithmic) CPU span and load-balanced PIM execution *whp* even against adversarial skew.
- **Communication-Reduced**: We reduce the (external-memory) communication factor from $O(\log n)$ or $O(\log_M n)$ in prior work to our PIM-based communication factor of $O(\log^* P)$.
- **Nearly Linear Space**: Our design has space consumption that is only a factor of $O(\log^* P)$ greater than optimal (linear).

The paper is organized as follows. §2 reviews kd-trees, the PIM model and other preliminaries used in this paper. §3 presents the main structure of our PIM-kd-tree design and the main techniques we use. §4 describes the efficient algorithms for PIM-kd-tree operations and their cost bounds. §5 proves the lower bounds and shows

[2]We use $O(f(n))$ **with high probability** (***whp***) (in n) to mean $O(cf(n))$ with probability at least $1 - n^{-c}$ for $c \geq 1$.
[3]All these cost bounds (including both shared-memory and PIM-based results) include an implicit linear factor of the dimension D, not shown in the asymptotic notation. Additionally, we will represent logarithms of the form $\max\{1, \log_{(\cdot)}(\cdot)\}$ and $\max\{1, \log^*(\cdot)\}$ using $\log_{(\cdot)}(\cdot)$ and $\log^*(\cdot)$, respectively. We believe this improves the readability of the paper.

Table 1: Space, work, and communication bounds in the PIM Model for different approaches. *Total work* refers to the sum of the CPU and PIM work (for shared-memory designs, *total work* equals *CPU work*, and is shown as a single column). *Communication* refers to the sum total of words sent to/from the shared memory (PIM modules) for shared-memory designs (PIM designs, respectively). Work and communication bounds for operations marked with ⋆ are for a batch of S operations. † are *whp* bounds and ‡ are in-expectation bounds. See Table 2 for all notations.

Approach	Space	Operation	CPU work	Total work	Communication
Log-tree [1, 79]	$O(n)$	LeafSearch⋆	$O\left(S\log^2\frac{n}{S}\right)^\dagger$		$O\left(S\log^2\frac{n}{S}\right)^\dagger$
		Insert/Delete⋆	$O\left(S\log n\right)^\dagger$		$O\left(S\log n\right)^\dagger$
PKD-tree [70]	$O(n)$	Construction	$O\left(n\log n\right)^\dagger$		$O\left(n\log_M n\right)^\dagger$
		LeafSearch⋆	$O\left(S\log\frac{n}{S}\right)^\dagger$		$O\left(S\log\frac{n}{S}\right)^\dagger$
		Insert/Delete⋆	$O\left(\frac{S}{\alpha}\log^2 n\right)^\dagger$		$O\left(\frac{S}{\alpha}\log_M n\log n\right)^\dagger$
		kNN⋆	$O(Sk\log n)^\ddagger$		$O(Sk\log n)^\ddagger$
		$(1+\epsilon)$-ANN⋆	$O(Sk\epsilon^{-D}\log n)^\ddagger$		$O(Sk\epsilon^{-D}\log n)^\ddagger$
PIM-kd-tree	$O(n\log^* P)$	Construction	$O\left(n(\log P+\log\log n)\right)^\dagger$	$O\left(n\log n\right)^\dagger$	$O\left(n\log^* P\right)^\dagger$
		LeafSearch⋆	$O\left(S\min\{\log^*(P),\log\frac{n}{S}\}\right)^\dagger$	$O\left(S\log\frac{n}{S}\right)^\dagger$	$O\left(S\min\{\log^* P,\log\frac{n}{S}\}\right)^\dagger$
		Insert/Delete⋆	$O\left(\frac{S}{\alpha}(\log P+\log\log n)\log n\right)^\dagger$	$O\left(\frac{S}{\alpha}\log^2 n\right)^\dagger$	$O\left(\frac{S}{\alpha}\log^* P\log n\right)^\dagger$
		kNN⋆	$O(Sk\log^* P)^\ddagger$	$O(Sk\log n)^\ddagger$	$O(Sk\log^* P)^\ddagger$
		$(1+\epsilon)$-ANN⋆	$O(Sk\epsilon^{-D}\log^* P)^\ddagger$	$O(Sk\epsilon^{-D}\log n)^\ddagger$	$O(Sk\epsilon^{-D}\log^* P)^\ddagger$
ParGeo [46, 101]	$O(n)$	DPC	$O(n(1+\rho)\log n)^\ddagger$		$O(n(1+\rho)\log n)^\ddagger$
		2d-DBSCAN	$O(n(k+\log n))^\dagger$		$O(n\log_M n)^\dagger$
PIM Clustering	$O(n\log^* P)$	DPC	$O(n(\log P+\log\log n+\rho\log^* P))^\ddagger$	$O(n(1+\rho)\log n)^\ddagger$	$O(n(1+\rho)\log^* P)^\ddagger$
	$O(n)$	2d-DBSCAN	$O(n\log P)^\dagger$	$O(n(k+\log n))^\dagger$	$O(n)^\dagger$

Table 2: Notations used in this paper.

Notation	Definition
P	Number of PIM modules
M	CPU cache size (in words)
n	Number of data points stored in the kd-tree (i.e., its size)
D	Number of dimensions
S	Batch size
$T(N_i)$	Number of descendants of tree node N_i (including itself)
k	Size of query neighborhood (in range query, kNN, clustering)
α	Balance parameter for kd-trees, $\alpha = O(1)$ (see §2.2)
σ	Over-sampling rate for kd-trees (§2.2)
β	Approximate counter probability parameter (§3.3)
ρ	Cubical density in density peak clustering (§6.1)

the optimality of our design. In §6, we use PIM-kd-tree techniques to solve two clustering problems as illustrations of its applications. §7 and §8 provide additional discussions and conclude the paper.

2 PRELIMINARIES

2.1 PIM Architecture and Computation Model

In this paper, we use the Processing-In-Memory (PIM) Model [54] to analyze algorithms and data structures. Experimental results from prior work [55, 57] show that the PIM Model is a good representation of a bank-level-in-memory-processing (BLIMP) system, which is commonly used in commercial real-world PIM systems like UPMEM [99] and Samsung PIMs [86].

The PIM Model is composed of a host CPU side and a PIM side of P PIM modules. The CPU side features a standard multicore architecture with a shared on-chip cache that holds M words. Each *PIM module* integrates a small on-chip *local memory* of $O(N/P)$ words (where N denotes the problem size or the total space consumed), collectively referred to as *PIM memory*, and a general-purpose but relatively weak compute unit known as the *PIM core*. The host CPU is capable of loading programs onto PIM modules, initiating their execution, and monitoring their completion. In terms of data access, the host CPU interacts with both its on-chip cache and the off-chip local memory of all PIM modules. However, each PIM core is restricted to accessing its own local memory. Communication between the host CPU and the PIM modules is achieved through the CPU execution of read/write instructions over the local memories of the PIM modules. PIM modules cannot communicate with each other directly; they rely on memory read/write manipulations by the CPU. This model operates under the assumption that programs execute in rounds similar to bulk-synchronous parallel (BSP) rounds [100], where in each round, the CPU can (1) conduct CPU computations, (2) read / write data to the local memory of PIM modules, or (3) launch PIM programs and wait for their completion.

To evaluate algorithms that involve both CPU and PIM components (distributed across P PIM modules), the PIM Model integrates both shared-memory and distributed metrics. For computations on the multicore CPU, it quantifies the **CPU work** (the total number of instructions executed by the CPU) and **CPU span** (the critical path length), under a binary forking model [5, 16, 18] with a work-stealing scheduler [19]. For off-chip communication, it measures the **communication time**, which is defined as the *maximum* number of word-sized messages sent to/from any PIM module within a round. (To facilitate comparisons with shared-memory computations, we also define the (off-chip) **communication**, which is the sum total of words sent to/from the off-chip shared memory

(PIM modules) for shared-memory algorithms (PIM algorithms, respectively).) For PIM programs, it measures the **PIM time**, the *maximum* work on any PIM core within a round. For executions with multiple rounds, the given cost metric is determined separately for each round, and the results of each round are aggregated to produce the final bound. Because both communication time and PIM time are based on the maximum across all PIM modules, it is crucial to design algorithms that ensure good load balance across PIM modules, even under adversarially chosen (skewed) workloads.

Definition 1 (PIM-balanced [54]). *An algorithm is **PIM-balanced** if it takes $O(W/P)$ PIM time and $O(I/P)$ communication time, where W is the PIM work across all P modules and I is the communication.*

The PIM-kd-tree is a *batch-parallel* data structure. Queries and updates are processed in batches of $S \geq 1$ operations of the same type, executed in parallel [55, 57, 89]. Our algorithm applies to any batch size S, but our analysis focuses on large batch sizes (i.e., $S = \Omega(P \log^2 P)$), as in the analysis of existing parallel data structures [3, 15, 40, 54, 56, 57, 98, 102].

The PIM-kd-tree adopts the PIM Model with its relatively simple PIM architecture in order to explore the fundamental theoretical challenges (e.g., CPU-PIM communication, load balancing) in PIM, which can be generalized to multiple real-world PIM platforms without being overfitted to machine-specific features.

Practical Relevance. The PIM Model [54] has been used as an analytical tool in prior works [45, 55, 57, 75], many with experiments. In particular, Kang et al. [55] showed that optimized algorithms under the PIM Model translate to practical and efficient implementations on real-world UPMEM machines [99], with all metrics in theory having their practical insights. Hence, while this paper focuses on theory, its results are expected to be of practical relevance.

Prior PIM-based Indexes. Early works on PIM-based index structures [26, 27, 68] adopt range-partitioning structures, where the key space is divided into disjoint key ranges and each key range is stored locally on a PIM module. Such designs are optimized in off-chip communication, but suffer from adversarial skew.

More recent skew-resistant indexes [54, 55, 57] first randomly distribute tree/skiplist nodes to PIM modules for skew resistance, and then divide the search index into different layers and apply different replication strategies to each layer to reduce off-chip communication. However, their techniques cannot be directly applied to kd-trees due to the challenges mentioned in §1.

None of the prior PIM work studied kd-trees.

Other Applications on PIM. Although the idea of PIM dates back to the 1970s [95], it has gained renewed attention recently, due to the development in 3D-stack memory fabrication [50] and the release of real-world PIM products [49, 86, 99]. Hundreds of academic works have been published (see the references of [7, 73, 74]). PIM systems have been widely used in accelerating applications of databases [9, 11, 51, 54, 55, 57, 59, 66], machine learning [21, 24, 48, 83, 108, 109], graph processing [43, 61, 91, 96], genome analysis [22, 33, 34, 60, 71] and security [2, 4, 32, 38, 42].

2.2 kd-trees and PKD-trees

kd-tree. The kd-tree is one of the most widely-used space-partitioning indexes for managing multi-dimensional data. The kd-tree is a space-partition binary tree, with each of the interior (non-leaf) nodes signifying an axis-aligned splitting hyperplane $\langle d, x \rangle$, where d is a dimension and x is a value. Despite no strict requirement, the classic kd-tree implementation picks the dimension with the widest stretch and uses the object-medians as the splitting hyperplane. Points to the left of the hyperplane (those with the d-th dimension coordinates smaller than x) are stored in the left subtree and the others are stored in the right subtree. This will lead to a perfectly balanced tree structure with $\log_2 n$ tree height.

However, this classic kd-tree solution is generally considered as a static data structure: finding the exact median is not only slow, but almost any update to the tree causes a tree rebuild. A classic solution to avoid frequent rebuilds is the *logarithmic method* [1, 10, 79], which maintains $O(\log n)$ of such perfectly-balanced trees with power-of-2 sizes. However, querying on $O(\log n)$ kd-trees leads to a significant slowdown, usually by more than a logarithmic factor.

Parallel kd-tree (PKD-tree). Men et al. [70] recently introduced the PKD-tree, which achieves high efficiency both theoretically and practically. Their key insight is to allow bounded imbalance in the tree, slightly relaxing the height guarantees. Theoretically, it is shown that the known kd-tree query bounds are not affected by relaxing the height guarantee to be $\log_2 n + O(1)$ instead of $\log_2 n$. Practically, their experimental results showed that the benchmark query performance on a large variety of query types remains mostly unchanged as long as the imbalance ratio at every node is at most 9-to-1. Hence, a scapegoat-like reconstruction-based rebalancing scheme, where a node is reconstructed whenever its imbalance exceeds 9-to-1, yields a highly efficient update algorithm for kd-trees in practice.

More formally, we define the **subtree size** $\mathbf{T(N_i)}$ for a node N_i in a tree as the number of descendants (including itself) of N_i in the tree. Following Men et al. [70], we define a kd-tree to be ***α*-balanced** iff for every non-leaf node N_i, $T(c)/T(c')$ is bounded by $(1+\alpha)$, where c and c' are the children of N_i and $T(c) \geq T(c')$.[4] We say a kd-tree is **strictly-balanced** iff $\alpha = O(1/\log n)$; its height is guaranteed to be $\log n + O(1)$. A kd-tree is **semi-balanced** iff $\alpha = O(1)$; its height is guaranteed to be $O(\log n)$. In the remainder of this paper, we assume that all kd-trees are semi-balanced (and may be strictly-balanced).

To achieve high efficiency, instead of building the tree one level each, PKD-tree builds a treelet skeleton (the interior nodes for several levels) based on a carefully selected sample set, and flushes all points to the leaves together. By controlling the skeleton size close to the cache size, the algorithm can achieve optimal I/O complexity. To build h levels at a time, a sketch sized $2^h \cdot \sigma$ needs to be sampled, where σ is the *over-sampling rate*.

The following two lemmas present bounds for construction and updates on the PKD-tree (see also Table 1). These bounds are analyzed using the binary-forking work-span model [5, 16, 18] and the ideal-cache (I/O) model [36] (where M and B are the cache and cache line sizes, respectively).

Lemma 2.1 (Shared-memory Construction [70]). *A PKD-tree of size n can be constructed in optimal $O(n \log n)$ work, $O(\log^2 n)$ span and $O(Sort(n)) = O(\frac{n}{B} \log_M n)$ cache complexity, all whp.*

[4] In [70], α-balanced is defined as the subtree sizes of each child of N_i being within a factor of $(0.5 \pm \alpha)$ of $T(N_i)$, for $\alpha \in [0, 0.5]$; our results also hold for this definition.

Lemma 2.2 (Shared-memory Updates [70]). *Updating (inserting or deleting) a batch of size $m = O(n)$ on an α-balanced PKD-tree of size n has amortized $O(\log^2 n/\alpha)$ work per element, $O(\log^2 n)$ span, and amortized $O(\log(n/m) + (\log n \log_M n)/B\alpha)$ cache complexity per element, all whp.*

In this paper, the PIM-kd-tree will further improve these shared-memory bounds by using PIM features.

2.3 Other Preliminaries

Load Balance. We use balls-into-bins lemmas to prove load balance on PIM systems, where a bin is a PIM module and a ball with weight w corresponds to a task with w work or w communication.

Lemma 2.3 (Weighted Balls into Bins [87]). *Uniformly randomly placing weighted balls with total weight $W = \sum w_i$ and each $w_i < W/(P \log P)$ into P bins yields $O(W/P)$ weight per bin whp.*

Iterated Functions. We use $\log_B^{(i)} n := \begin{cases} \log_B n & i = 1 \\ \log_B \left(\log_B^{(i-1)} n\right) & i \geq 2 \end{cases}$ in this paper. We also use the log-star function defined as $\log_B^* n := \min\{i \mid \log_B^{(i)} n \leq 1\}$. We omit the notion of base B when $B = 2$.

3 PIM-kD-TREE DATA STRUCTURE

In this section, we introduce our design of the PIM-kd-tree. Recall that there exist multiple challenges (C0-C3 in §1) in designing PIM-efficient kd-trees. The main challenge for construction is to achieve *PIM-balance*—different PIM modules should execute similar workload even if the update/query distribution is adversarially skewed. A straightforward design for a PIM-based kd-tree is to partition the tree into P disjoint subtrees and store each on a different PIM module. However, an adversary can construct a batch with all the queries touching a small subspace, overflowing the work on the PIM modules where the corresponding subtrees reside while making the other PIM modules idle.

To resolve this problem, the PIM-kd-tree randomly stores all the tree nodes on different PIM modules using hash-based approaches, so that balls-into-bins analysis (Lemma 2.3) guarantees that the paths can be searched simultaneously across different PIM modules in a load-balanced fashion. Such distributed nodes are called *master* nodes in this paper. However, a naive design only with distributed master nodes cannot reduce the amount of off-chip communication—one remote access will occur on every tree edge during searches, since the master nodes of the parent and children are stored on different PIM modules, which makes the amount of communication no different from shared-memory designs. This contradicts the purpose of using PIM systems.

To *reduce communication* without violating load balance, the PIM-kd-tree builds local *on-chip caching* on each master node, with replicated information from its nearby tree nodes that are likely to be collocated during searches. Later search queries will only need to traverse these on-chip caches locally on one PIM module and do not need to go to other master nodes on other PIM modules.

In §3.1, we introduce how to organize this caching and replication with low overheads in space and updates and without violating load balance. Then in §3.2 we give a construction algorithm to build the PIM-kd-tree structure (including all the caching). §3.3 and §3.4 present other techniques we use to maintain tree properties with lightweight overheads.

3.1 Main Structure

Log-star Tree Decomposition. As a first step to build a PIM-kd-tree, we decompose all the nodes in a kd-tree into ($\log^* P$+1) groups by subtree size as follows. Denote $H_j = \log^{(j)} P$ for $1 \leq j \leq \log^* P$, and $H_0 = P$. Recall from §2.2 that $T(N_i)$ is the number of N_i's descendants in the kd-tree. If $T(N_i) \geq H_0 = P$, then we categorize N_i to be in Group 0. Otherwise, there exists a positive integer $j \in [1, \log^* P]$ where $H_j \leq T(N_i) < H_{j-1}$, and we categorize N_i to be in Group j.

Figure 1 shows an example of decomposing the top part of a tree using the log-star decomposition. Each group in the decomposition is a forest, each of whose tree is a subtree of the original kd-tree. The smaller j is, the higher position Group j lies in.

The log-star tree decomposition depends only on the *subtree size* (the number of descendants) of each node, unlike the prior PIM-tree [55] which uses the *height* of each node. This is because a kd-tree can be semi-balanced, so that a node's height is an imprecise indicator of its position in the tree. For example, in an α-balanced tree with $\alpha = 2$, a node with $\log n$ depth from the root can range from being a leaf to having $\Theta(\sqrt{n})$ descendants.

Replication Method I: Top-down Caching. We use the following replication strategy in the PIM-kd-tree based on the log-star tree decomposition:

- All Group 0 nodes and their tree structure are replicated on all P PIM modules.
- For each node N_i in Group j ($1 \leq j \leq \log^* P$), together with the storage of the master node, the PIM-kd-tree stores on the same PIM module a copy of N_i's every descendant in Group j and their corresponding intra-group subtree structure.

We give an example of the top-down caching in Figure 2c. In an ideal case, with the log-star tree decomposition and the replication, a search from root to leaf will only require $O(\log^* P)$ off-chip communication. Traversing each group is local on one PIM module in a replicated intra-group subtree, and off-chip communication is only required across the borders between different groups.

Replication Method II: Bottom-up Caching. In addition to replication strategy I to benefit top-down tree searches, we also introduce bottom-up caching to benefit the bottom-up tree searches that occur as part of kNN queries. To be specific, for each node N_i in Group $j \geq 1$, we store both the master node and a chain of all N_i's ancestors in Group j on the same PIM module. An example is given in Figure 2d. In an ideal case, similar to top-down searches, such bottom-up caching ensures that a bottom-up search incurs $O(\log^* P)$ off-chip communication due (solely) to inter-group pointer chasing.

Final Replication Strategy: Dual-way Caching. We combine both the top-down caching and the bottom-up caching in the PIM-kd-tree, as shown in Figure 2b. Both top-down searches and bottom-up searches will benefit from the local replicated nodes to reduce off-chip communication. In an ideal case, a search path between any two positions in the PIM-kd-tree will incur at most $O(\log^* P)$ off-chip communication.

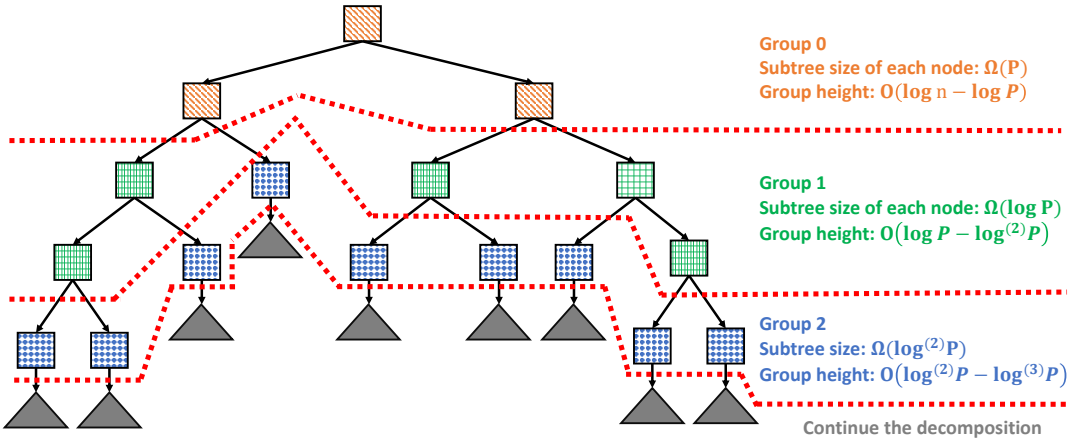
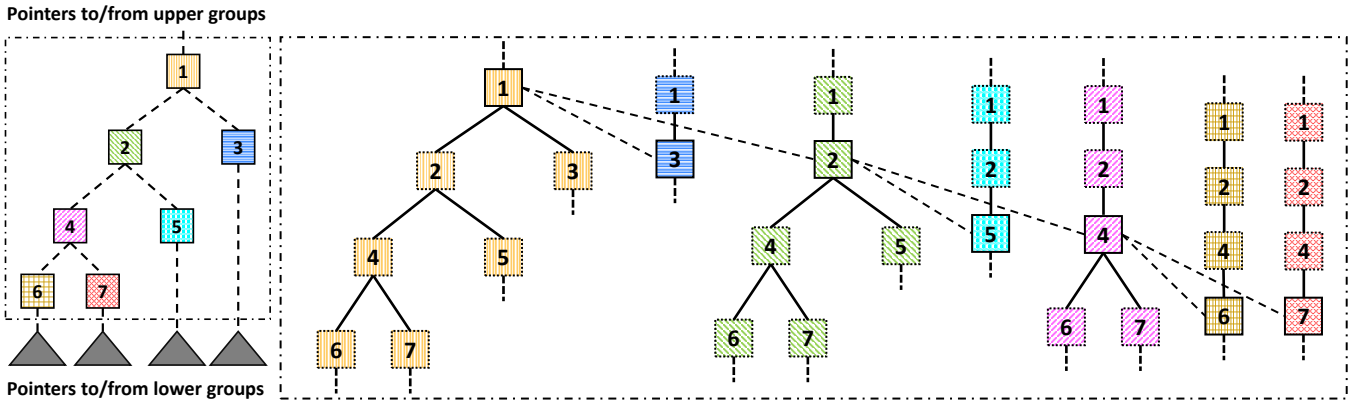
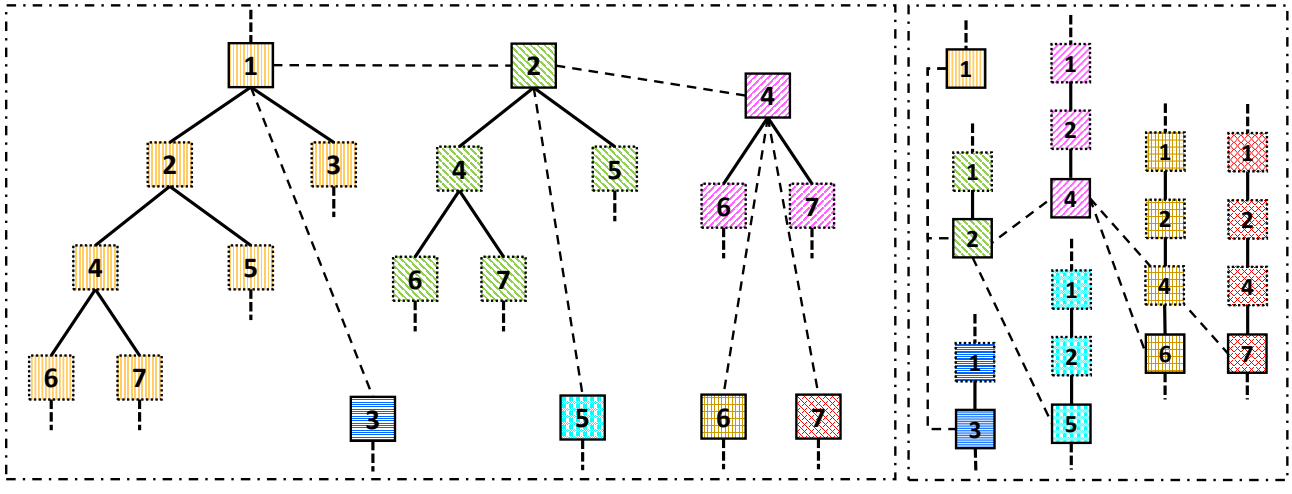


Figure 1: An example of the log-star decomposition in the top part of a tree, showing Groups 0, 1, and 2. Nodes with the same color/hashing are in the same group. The red dash line represents the borders between groups. The decomposition depends only on the subtree size $T(N_i)$ of each node N_i, rather than N_i's height. The height of each group is proved in Lemma 3.2.



(a) No caching. (b) Dual-way caching.



(c) Top-down caching. (d) Bottom-up caching.

Figure 2: Examples of applying different replication strategies to an intra-group subtree with 7 internal nodes from Group j ($j \geq 1$). Different colors represent 7 different PIM modules where the data are physically stored. Nodes with solid frames are master nodes. Nodes with dashed frames are replicated copies built in caching. Solid node-linking lines are intra-module (on-chip) bidirectional pointers, and dashed node-linking lines are inter-module (off-chip) bidirectional pointers.

Space Consumption. We will now show a PIM-kd-tree with n data points takes $O(n \log^* P)$ space (Theorem 3.3). We first prove that the number of nodes in each group is bounded.

For a given $t > 0$, we define nodes N_i with $T(N_i) \geq t$ as **high-level** nodes, and nodes with $T(N_i) < t$ as **low-level** nodes. High-level nodes both of whose children are also high-level nodes are **inner** high-level nodes, and high-level nodes with at least one low-level child are **borderline** nodes. All the high-level nodes comprise a single subtree of the upper part of the kd-tree, with the borderline nodes being the leaves in this subtree.

Lemma 3.1 (Bounded Number of High-level Nodes). *For an α-balanced kd-tree with size n and any $t > 0$, the number of nodes N_i in the kd-tree with $T(N_i) \geq t$ is $O(n/t)$.*

Proof. It suffices to prove that the number of borderline nodes is $O(n/t)$, since all high-level nodes form a binary tree.

For each of the less than $2n$ low-level nodes in the original kd-tree with n keys, determine its unique lowest ancestor that is a borderline node. For each borderline node N_i with two low-level children, all of its descendants (other than N_i itself) regard N_i as their lowest borderline ancestor; since $T(N_i) \geq t$, this is at least $t-1$ low-level nodes. For each borderline node N_i with one high-level child c and one low-level child c', all descendants of c' regard N_i as their lowest borderline ancestor. Because $T(c)/T(c') \leq (1+\alpha)$ and $T(c) \geq t$, we have $T(c') \geq t/(1+\alpha)$. Thus in either case, there are $\Omega(t)$ low-level nodes for each borderline node, and hence at most $O(n/t)$ borderline nodes, implying $O(n/t)$ high-level nodes. □

Lemma 3.1 indicates that Group j will have $O\left(n/\log^{(j)} P\right)$ nodes in total. We now prove that the height of each group is also bounded.

Lemma 3.2 (Bounded Height of Log-star Decomposition). *For a kd-tree and its log-star decomposition, the maximum height of all intra-group subtrees in Group $j \geq 1$ is $O\left(\log^{(j)} P\right)$.*

Proof. To handle the corner case $j = 1$, define $\log^{(0)} P$ to be P. Because the kd-tree is α-balanced, the subtree size of the larger child of a non-leaf node N_i will be at least a factor of $(1+\alpha)/(2+\alpha)$ smaller than $T(N_i)$. Thus each level we go down in the kd-tree, the subtree sizes will decrease by at least this factor. The highest nodes in Group j have subtree sizes of $\Theta\left(\log^{(j-1)} P\right)$, and the lowest nodes have subtree sizes of $\Theta\left(\log^{(j)} P\right)$. Thus, the height of Group j must be at most $\log_{\frac{1+\alpha}{2+\alpha}}\left(\frac{\log^{(j-1)} P}{\log^{(j)} P} \cdot \Theta(1)\right) = O\left(\log \frac{\log^{(j-1)} P}{\log^{(j)} P}\right) = O\left(\log^{(j)} P\right)$. □

Theorem 3.3 (Space Consumption). *An α-balanced PIM-kd-tree containing n data points takes a total of $O(n \log^* P)$ space to store all the data points and auxiliary structures.*

Proof. The theorem is proved by showing that each group takes $O(n)$ space to store its replicas. First, due to Lemma 3.1, Group 0 has $O(n/P)$ nodes, and there are P copies, resulting in $O(n)$ space.

For Group $j \geq 1$, the number of additional replicated node copies equals the number of ancestor-descendant pairs inside this group, multiplied by 2× due to dual-way caching. In other words, the space overhead equals 2× the sum of the number of intra-group ancestors over all the nodes inside the group. There are $O\left(n/\log^{(j)} P\right)$ nodes in the group (Lemma 3.1), and the number of the intra-group ancestors of each node is bounded by the group height, which is $O\left(\log^{(j)} P\right)$ by Lemma 3.2. Thus, each group takes $O(n)$ space. □

3.2 The Construction Algorithm

In this section, we describe the algorithm for constructing a PIM-kd-tree with cost analysis. This algorithm will also be a subroutine in the batched update algorithms in §4.2.

Construction on a Single Module. We first propose an on-chip algorithm to construct a PIM-kd-tree sized n', shown in Algorithm 1. This module could be either the CPU host or one PIM module (and n' should fit into the CPU cache or the PIM local memory). The high-level idea is that we first build a kd-tree in parallel as in [70]. Then we decompose the kd-tree using log-star decomposition, and create replicas for each intra-group subtree in parallel.

Lemma 3.4 (On-chip construction). *Algorithm 1 takes a computation cost of $O(n' \log n')$ work and $O(\log^2 n')$ span whp.*

Proof. By Lemma 2.1, Line 2 in Algorithm 1 costs $O(n' \log n')$ work *whp* and $O(\log^2 n')$ span. Lines 8-10, 12, 16-17, 20-23 and 25-28 can be implemented using linear-work parallel primitives based on semi-sort [30] and prefix sum [16]. Lines 7, 11, 19 and 24 can be implemented using linear-work treefix operations [16] and tree construction [3]. All of these takes at most $O(n' \log^* P)$ work *whp* and at most $O(\log n' \log\log n')$ span *whp*. □

The Main Algorithm for Construction. Now we describe the main algorithm to construct an entire PIM-kd-tree across all PIM modules (Algorithm 2). The high-level idea is that we first sample some points to build a sketch that fits in the CPU cache, acting as the highest part of the kd-tree. Then we offload the construction of each lower subtree and its replicas on a PIM module. Finally, we send back the constructed tree structure to the CPU and redistribute all the master nodes and their replicas to the PIM modules.

To ensure load balance, the sketch size is set to $O(P\sigma)$ which fits in the CPU cache. We divide the remaining part of the kd-tree into a forest with P subtrees. By choosing $\sigma = \log^3 n$, all the P subtrees will have size $O(n'/P)$ [70]. Thus, if the CPU cache size $M = \Omega(P \log^3 n)$, we can afford to assign each PIM module with a subtree and offload the construction of these subtrees and their replicas to the PIM side. Finally, we send back the constructed replicas and rearrange a load-balanced storage. Only three phases of CPU-PIM interaction are needed, and the cost bounds are given in Theorem 3.5.

Theorem 3.5 (PIM-kd-tree Construction). *Assume a PIM system with CPU cache size $M = \Omega(P \log^3 n)$. Constructing a PIM-kd-tree containing n' points takes $O\left(n'(\log P + \log\log n')\right)$ CPU work, $O(\log n' + \log^2 P + (\log\log n)^2)$ CPU span, $O(\frac{n'}{P} \log \frac{n'}{P})$ PIM time and $O(\frac{n'}{P} \log^* P)$ communication time, all whp.*

Proof. Because we can directly use the shared-memory construction when $n' = O(M)$, we discuss here only the case when $n' =$

Algorithm 1: On-chip PIM-kd-tree Construction

Input: A sequence of data points $Pts[1:n]$
Output: A corresponding PIM-kd-tree $Tree$

1 **Function** *On_Chip_Build*(Pts) :
2 $Tree'$ ← Parallel PKD-tree construction on Pts
   `// During PKD-tree construction, also record the subtree sizes T() of each internal node`
3 $Groups$ ← Log_star_decompose($Tree'$)
4 $Tree$ ← Replicate($Tree', Groups$)
5 **return** $Tree$
6 **Function** *Log_star_decompose*($T[1:n]$) :
7 Leaffix: Compute the subtree size of each node
8 **parallel for** *each node* N_i *in* T **do**
9 **If** N_i *and parent*(N_i) *not in the same group* **then**
10 Label N_i as "group root"
11 Rootfix: Each node gets its closest ancestor who is a "group root"
12 $Groups$ ← A set of subtrees, where nodes with the same "group root" ancestor fall into the same subtree
13 **return** $Groups$
14 **Function** *Replicate*($T'[1:n], Groups$) :
15 **parallel for** *each subtree ST in Groups* **do**
16 **If** ST == *Group* 0 *nodes* **then**
17 Replicate P copies of ST into T
18 **Else then**
   `// Build top-down subtree caching`
19 Rootfix: Record the height of each node in ST
20 Replicate *subtree_height* copies of subtree ST
21 **parallel for** *each node in each subtree copy* **do**
22 **If** *Node ID is smaller than copy ID* **then**
23 Remove the node from the subtree copy
24 Leaffix: Combine all subtree copies into T
   `// Build bottom-up ancestor caching`
25 **parallel for** *each copied node in each subtree copy* **do**
26 Create pair {copied node ID, subtree root ID}
27 Semi-sort the pairs with pair.first
28 Combine the sorted pairs into ancestor chains
29 **return** T

Algorithm 2: PIM-kd-tree Construction in PIM Systems

Input: A sequence of data points $Pts[1:n]$
Output: A corresponding PIM-kd-tree $Tree$ across P PIM modules

1 **Function** *Build_PIM-kd-tree*(Pts) :
   `// In CPU cache: Lines 2-6`
2 Pts' ← Sample $P \cdot \sigma$ points from Pts
3 $CacheForest[1:P]$ ← On_Chip_Build(Pts')
4 **parallel for** *Each node* N_i *in Pts* **do**
5 idx ← Search N_i in $CacheForest$
6 Send N_i to PIM module idx
   `// On PIM module i: Lines 7-8`
7 T_i ←On_Chip_Build(Received points)
8 Send T_i to CPU
   `// In CPU cache: Lines 9-10`
9 Collect all T_i and build replications on the highest levels
10 $Tree$ ← Send each intra-group subtree copy to a random PIM module and store
11 **return** $Tree$

$\Omega(M)$. We use $\sigma = \log^3 n$ here to ensure load balance in later executions [70]. Line 3 in Algorithm 2 takes $O\left(P \log^3 n(\log P + \log\log n)\right)$ CPU work and $O(\log^2 P + (\log\log n)^2)$ CPU span *whp*. Line 5 takes $O\left(n'(\log P + \log\log n')\right)$ CPU work and $O(\log n')$ CPU span (the height of the tree). Line 7 takes $O(\frac{n'}{P}\log\frac{n'}{P})$ PIM time *whp* due to Lemma 3.4. Line 8 takes $O(\frac{n'}{P}\log^* P)$ communication time *whp* due to Theorem 3.3. Line 9 takes $O(n'\log^* P)$ CPU work and $O(\log^2 n')$ span. Line 10 sends $O(n'\log^* P)$ total communication. We will prove the load balance on Line 10 in §3.4. □

Construction is Work-efficient. For Theorem 3.5, we can see that the total computation work on the CPU and PIM is $O(n'\log n')$, making the construction algorithm work-efficient compared to shard-memory designs [17, 70, 102]. Meanwhile, the CPU execution is polylogarithm span and the PIM execution is load-balanced *whp*, exploiting adequate parallelism in the system.

3.3 Approximate Probabilistic Counters

In our kd-tree design for PIM, in order to keep the tree α-balanced, it is essential to deploy a counter on each internal node to record the size of the subtree. Ideally speaking, all replicas should maintain an *accurate* version of the subtree size counter so that a search anywhere in the PIM system can immediately detect any α-balance violations. However, this would incur unaffordable update costs, because the changes in counter values during update would accumulate in the higher parts of the tree, while our replication strategy in §3.1 is to create *more* copies at higher levels, which would all need updating.

Although there exist many approximate counter designs in streaming/sketching algorithms (e.g., [52, 62, 63, 69, 104]), they mainly focus on optimizing space usage and sweep times. We are more interested in the Morris counter family (originally [72]; see the more recent Steele counters [93, 94, 105]), which perform lazy updates. However, these designs are either too frequently updated in our settings or not accurate enough to be directly used in the PIM-kd-tree, which might violate α-balance (affecting the tree height) or increase the costs for dynamic updates.

Approximate Counter Design. We introduce our *approximate probabilistic counter* variant (Algorithm 3) for PIM-kd-trees by coupling the update probability with both the subtree size and the total tree size. The high-level idea is that to avoid frequent counter updates while keeping a well-estimated value *whp* in n, every time an

Algorithm 3: Increment an approximate counter

Input: An approximate counter *Counter*
Function *Increment-Counter*(*Counter*) :
  $V \leftarrow Counter$.value
  $p \leftarrow \log n/(\beta V)$
  If *Success in a coin throwing with probability* p **then**
    $Counter'$.value $\leftarrow V + 1/p$
  return $Counter'$

incremental operation is requested, the actual counter with value V is updated with probability $p = \log n/(\beta V)$. Here $\beta = O(1)$ is the ***approximate counter probability parameter***.

Estimation Accuracy. Lemma 3.6 shows that our approximate counters are accurate enough under certain conditions.

Lemma 3.6 (Counter Accuracy). *Assume that ΔV incremental operations are executed on a probabilistic counter with original value V. Let V' be the final counter value after incrementing using Algorithm 3. Then $V' - V = \Delta V + o(\Delta V)$ whp in n if $\Delta V = \Omega(\beta V)$ and $\Delta V = O(V)$.*

Proof. Let $V_1 = V + \Delta V$. The estimated counter value is no worse than the case where every of the ΔV increments succeeds with probability $p = \log n/(\beta V_1)$. $V' - V$ equals $1/p$ multiplied by a binomial random variable with success probability of p and ΔV trials. By a Chernoff bound [80, 87], we have $Pr\{V' - V \geq (1+\epsilon)\Delta V\} \leq \exp(-\epsilon^2 \Delta V p/2) = \exp\left(-\frac{\epsilon^2 \Delta V \log n}{2\beta V_1}\right) = n^{-\frac{\epsilon^2 \Delta V}{2\beta V_1}} = n^{-\epsilon^2 \Omega(1)}$. □

There is an analogous lemma for decrement operations, where the Chernoff bound shows that $Pr\{V - V' \geq (1+\epsilon)\Delta V\} \leq \exp\left(-\frac{\epsilon^2 \Delta V \log n}{2\beta V}\right) = n^{-\epsilon^2 \Omega(1)}$.

Effect on the Tree Height. In our PIM-kd-tree, we keep an approximate counter on every copy of every node. By setting $\beta = \Theta(\alpha)$, we can ensure that the kd-tree is α-balanced *whp* even when using these approximate probabilistic counters to record the subtree sizes of each node. Lemma 3.7 shows that using approximate counters in the PIM-kd-tree *does not change the tree height bounds*. In §4.2, we will show how to maintain these approximate counters with low cost for Insert/Delete queries.

Lemma 3.7 (Tree height). *Suppose our approximate counters are used for subtree sizes, with $\beta = \Theta(\alpha)$. Then the height of a PIM-kd-tree is (i) $O(\log n)$ whp for $\alpha = O(1)$ and (ii) $\log n + O(1)$ whp for $\alpha = O(1)/\log n$.*

Proof. The tree height analysis from PKD-tree [70] takes as input that each recursive level of sampling-based construction generates an α-balanced sketch *whp* in n, and concludes that the tree has a height described above. Similarly, if that analysis takes as input that our approximate counters keep every internal node α-balanced *whp* in n, it will conclude that the PIM-kd-tree has the same height bounds. □

3.4 Auxiliary Techniques for Load Balance

Push-Pull Search. We use *push-pull search* [55, 57] as a key routine in the PIM-kd-tree to guarantee load balance under skewed workloads. Compared with distributed systems, PIM systems have an additional powerful host side (§2.1), which enables many shared-memory techniques to be adopted. The push-pull search exploits such features by taking advantage of both sides, compared with the always-offloading designs of many previous distributed algorithms.

Push-pull search uses the contention information in the batch to decide where the computation in a PIM-kd-tree is actually executed (in CPU cache or on PIM modules). We give an example here for LeafSearch (i.e., a top-down search from root to leaf) to illustrate how push-pull search works. Within a PIM execution round with batch size S, suppose that m queries want to search the internal node N_i to decide which child subtree to go to. As long as m is below a threshold that would cause load imbalance (defined below), then all the m search queries are *pushed* (i.e., sent) to the PIM module that stores N_i, and the search traverses the local replicated intra-group subtree of N_i. Otherwise, if m exceeds the threshold, then N_i and pointers to its children are *pulled* (i.e., fetched) from the PIM side to the CPU, and a parallel search is carried out on the CPU. During pulling, the local intra-group caching is not fetched to avoid communication imbalance. After the on-CPU search, the m queries are divided into two groups based on which child node to go to, and push-pull detection is recursively executed level-by-level until all leaves are reached. By resolving potential contentions on the CPU, each round is PIM-balanced:

Lemma 3.8 (Push-pull Search [55]). *On a search tree with node fanout C and applying our caching, we set the push-pull threshold to be $\tau = C \cdot H_{G1}$ where H_{G1} is the maximum height of intra-group subtrees. Then a batch of $S = \Omega(\tau P \log P)$ LeafSearch queries incurs $O(S/P)$ communication time whp to traverse each group.*

Delayed Construction on Group 1. For Groups $j \geq 1$, each intra-group subtree is sized $O\left(\log^{(j-1)} P/\log^{(j)} P\right)$, where we will again define $\log^{(0)} P$ to be P. When creating replicated intra-group subtrees on PIM modules (Algorithm 2, Line 10), we need to ensure that $\log^{(j-1)} P/\log^{(j)} P = O(S/(P \log P))$, so that Lemma 2.3 can be applied to guarantee load balance. Any $S = \Omega((\log P/\log\log P) \cdot P \log P)$ such that $S \leq M$ suffices for Group $j = 2$, and hence for all Groups $j \geq 2$. Since $M = \Omega(P \log^3 n)$ and $n \geq P$ (otherwise the entire tree fits in the CPU cache), such S exist. For Group 1, however, each intra-group subtree is sized $O(P/\log P)$, so load balance will not be satisfied if the replicas are directly pushed to the PIM side in the same phase.

We propose a ***delayed construction*** strategy on nodes in Group 1 for guaranteed load balance. In the first batch of construction, for all the nodes that are in an intra-group subtree sized $\Omega(S/P \log P)$, only their master nodes are constructed and their local caching is not built. These master nodes are marked as *unfinished*. We record the total number of unfinished intra-group subtrees on the CPU. If this number is smaller than $P \log P$, we keep executing the later batches coming from the outside. When this accumulated number exceeds $P \log P$, an extra execution phase is triggered where the caching of all unfinished nodes are constructed.

Proof for load balance in Theorem 3.5. Now we analyze the case to construct a tree with size $S = \Omega(P \log^2 P)$. With delayed construction, each master node who is not delayed in the current round will have a local caching size of $O(\log P)$ which is $O(S/(P \log P))$. This allows Lemma 2.3 to be used and guarantees the load balance of work and communication in the current round.

The load balance for the extra construction phase of unfinished nodes is also guaranteed. Since the entire tree is α-balanced, all unfinished Group 1 subtrees have the same asymptotic numbers of unfinished nodes. Constructing at least $P \log P$ unfinished subtrees together allows the usage of Lemma 2.3. □

Lemma 3.9 (Communication overhead with unfinished nodes). *For a batch of $S = \Omega(P \log P)$ queries, assume that each query touches at most $O(S/(P \log P))$ nodes in one intra-group subtree. Let t be the communication time in a PIM-kd-tree to finish the construction of unfinished nodes first and execute the batch, where the construction cost is ignored. Then the communication time is $\Theta(t)$ whp to execute the query batch while still leaving the nodes unfinished in the PIM-kd-tree.*

Proof. Each unfinished intra-group subtree introduces an extra communication of $O(S/(P \log P))$, and there are at most P unfinished subtrees in the system The communication overhead of unfinished nodes is $C = \Theta(tP) + O(P \log P \cdot S/(P \log P)) = \Theta(tP) + O(S) = \Theta(tP)$. After introducing push-pull search and delayed construction, load balance can be strictly guaranteed. Thus the communication time of leaving the nodes unfinished is still $\Theta(t)$ *whp*. □

4 OPERATIONS

In this section, we show how the PIM-*k*d-tree efficiently supports LeafSearch, dynamic updates, and nearest neighbor and orthogonal range queries. All operations are *whp* work-efficient, PIM-balanced, and have $O(\log^* P)$ communication and space overheads.

4.1 Leaf Search

The LeafSearch(Q) operation takes a query batch Q consisting of S points as input, and for each query point returns the leaf node the point would reside in. In shared-memory implementations, this operation takes $O(S \log(n/S))$ work and communication *whp* (Table 1).

Algorithm 4 presents pseudocode for LeafSearch(Q). The high-level idea is to first search through Group 0 by dividing the searches evenly onto arbitrary PIM modules, and then use push-pull search to traverse through each Group $j \geq 1$. Theorem 4.1 bounds the cost of this algorithm. The proof is a direct combination of shared-memory search [70], Lemma 3.8 and Lemma 3.9.

Theorem 4.1 (LeafSearch costs). *A batch of $S = \Omega(P \log^2 P)$ LeafSearch on a PIM-kd-tree of size n takes $O(S \min\{\log^* P, \log \frac{n}{S}\})$ CPU work, $O(\log^* P \log n)$ CPU span, $O(\frac{S}{P} \min\{\log^* P, \log \frac{n}{S}\})$ communication time and $O(\frac{S}{P} \log \frac{n}{S})$ PIM time, all whp.*

4.2 Dynamic Updates

We first discuss how to implement batch-dynamic Insert on a PIM-*k*d-tree. Our batched Insert adopts a partial reconstruction method similar to [70] and executes in a two-stage fashion. In the first stage, we execute a LeafSearch helper to locate the positions in the PIM-*k*d-tree for new points to be inserted, with two modifications compared with the original LeafSearch. In the second stage, we commit the insertion to the positions found in the first stage.

Algorithm 4: LeafSearch

Input: A batch of points $Q[1:S]$
Output: Leaf node addresses $Lf[1:S]$

1 **Function** LeafSearch(Q) **:**
  `// On CPU: Lines 2-5`
2 Initialize $Subset[1:P]$
3 **parallel for** $i \leftarrow 1:P$ **do**
4 $Subset[i] \leftarrow Q\left[\frac{(i-1)S}{P} + 1 : \frac{iS}{P}\right]$
5 Send each $Subset[i]$ to PIM module i
  `// On PIM module i: Lines 6-9`
6 Initialize $G1Roots[1:\frac{S}{P}]$
7 **parallel for** $j \leftarrow 1:\frac{S}{P}$ **do**
8 $G1Roots[j] \leftarrow$ Traverse Group 0 to find the Group 1 root node with key $Subset[i][j]$
9 Send $G1Roots[\cdot]$to CPU
  `// On CPU: Line 10`
10 $Lf[1:S] \leftarrow$ Collect $G1Roots[1:P]$
  `// Between CPU and PIM: Lines 11-13`
11 **for** $gid \leftarrow 1$ *to* $\log^* P$ **do**
12 **parallel for** $j \leftarrow 1:S$ **do**
13 $Lf[j] \leftarrow$ Use push-pull search to traverse the subtree in Group gid rooted at $Lf[j]$ for $Q[j]$
14 **return** $Lf[1:S]$

Modification I to LeafSearch Helper: Counter Updates. Every internal node in the PIM-*k*d-tree includes an approximate counter denoting the approximate number of points contained in its subtree (§3.3). When we execute the LeafSearch helper from root to leaf, every time the helper touches an internal node, we would want to increment the approximate counter of the node, because the new point will be inserted into its subtree. When we traverse a group, we traverse until we reach the lowest node on the search path that is still in the group. Then we use Algorithm 3 (with $\beta = \Theta(\alpha)$) to increment the approximate counter on this lowest node. If the increment is successful, we update all copies of this node. In addition to incrementing the counters of the lowest nodes, this update also increments the counters of their ancestors in the same group.

Lemma 4.2 (Counter maintenance cost). *Maintaining the approximate counters during a batch of $S = \Omega(P \log^2 P)$ Inserts takes an amortized $O(\frac{S}{\alpha P} \log^2 n)$ PIM time and $O(\frac{S}{\alpha P} \log n \log^* P)$ communication time whp.*

Proof. A lowest node in Group j has a subtree size of $\Theta(\log^{(j)} P)$. Thus, the success probability for an increment is $\Theta(\log n/\alpha \log^{(j)} P)$. Each successful increment triggers $\Theta(\log^{(j)} P)$ communication (the number of copies) and $O((\log^{(j)} P)^2)$ PIM work (by Lemma 3.2, each copy has $O(\log^{(j)} P)$ ancestors). Thus, for Group j, the cost

to maintain approximate counters is an amortized $\Theta(\frac{1}{\alpha}\log n)$ communication and $O(\frac{1}{\alpha}\log^{(j)} P \log n)$ PIM work. The total communication (PIM work) is an amortized $\Theta(\frac{1}{\alpha}\log n)$ ($O(\frac{1}{\alpha}\log^2 n)$, respectively) per insertion. In addition, $S = \Omega(P\log^2 P)$ and Lemma 2.3 guarantee PIM balance *whp*. □

Modification II to LeafSearch Helper: Terminal Position. The original LeafSearch returns a leaf node as the terminal position. In the Insert version, however, we apply increments to the approximate counters during the search and detect whether α-balance is violated at the current position. If imbalance is detected, the current node is returned without further heading to the leaf.

Stage 2: Partial Reconstruction. After Stage 1, where all imbalanced internal nodes are returned, we rebuild the entire imbalanced subtree together with the new points to be inserted using Algorithm 2. If no imbalance occurs on one search path and a leaf node is returned, we directly insert the new points into the leaf. In addition, we also promote nodes on the search path into upper groups based on the updated approximate counter value of the nodes.

Insertion Cost. The overall cost of Insert is shown in Theorem 4.3.

Theorem 4.3 (Insertion). *Inserting $S = \Omega(P\log^2 P)$ points into a PIM-kd-tree of size n takes $O(\frac{S}{\alpha}(\log P + \log\log n)\log n)$ CPU work, $O(\log^2 n)$ CPU span, $O(\frac{S}{P\alpha}\log^* P \log n)$ communication time and $O(\frac{S}{P\alpha}\log^2 n)$ PIM time, all whp and amortized.*

Proof. We use an amortization argument similar to PKD-tree [70] for the costs of partial reconstruction. Algorithm 2 takes *whp* $O(n'(\log P + \log\log n'))$ CPU work, $O(n'\log\frac{n'}{P})$ PIM work and $O(n'\log^* P)$ communication to build a subtree of size n' (see Theorem 3.5). For an α-balanced subtree of size n', we would need to insert another $\Theta(\alpha n')$ new points to trigger a partial reconstruction due to imbalance. The amortized cost per point on this subtree root is $O(\frac{1}{\alpha}(\log P + \log\log n'))$ CPU work, $O(\frac{1}{\alpha}\log n')$ PIM work and $O(\frac{1}{\alpha}\log^* P)$ communication. Now sum over the tree height $O(\log P)$, which gives an amortized $O(\frac{\log n}{\alpha}(\log P + \log\log n))$ CPU work, $O(\frac{1}{\alpha}\log^2 n)$ PIM work and $O(\frac{\log n}{\alpha}\log^* P)$ communication *whp*. Finally, we combine Lemma 3.8 for load balance, Theorem 4.1 for search cost and Lemma 4.2 for overhead to maintain approximate counters, all of which lead to the final bounds. □

Deletion. The Delete operation on PIM-kd-tree is a symmetric case to Insert. Their executions are extremely similar, starting with LeafSearch to update the approximate counters and find the highest nodes with imbalance, and then partially reconstructing all imbalanced subtrees whose requested points are removed. Theorem 4.4 gives the cost bounds of Delete; the proof is omitted.

Theorem 4.4 (Deletion). *Deleting $S = \Omega(P\log^2 P)$ points from a PIM-kd-tree containing n points takes $O(\frac{S}{\alpha}(\log P + \log\log n)\log n)$ CPU work, $O(\log^2 n)$ CPU span, $O(\frac{S}{P\alpha}\log^* P \log n)$ communication time and $O(\frac{S}{P\alpha}\log^2 n)$ PIM time, all whp and amortized.*

4.3 Nearest Neighbors and Orthogonal Ranges

k Nearest Neighbors. A k nearest neighbor (kNN) query takes a point p as input and returns the k nearest neighbors of p inside the dataset. Given arbitrary datasets, kNN queries might require worst-case poly(n) work, but better bounds exist if there are constraints on the dataset distribution.

A sequential kNN algorithm first locates the leaf for the kNN center. Then it uses a Fibonacci heap to maintain the k nearest candidates. We denote r as the radius of this candidate set ($r = \infty$ for fewer than k candidates). The kNN backtracks for all leaf nodes with distance less than r and terminates when all untracked leaves are further than r away. The PIM-kd-tree executes kNN in a similar fashion, but parallelizes the searches across PIM modules.

The 1977 claim [35] that kNN in a kd-tree takes an average-case $O(k\log n)$ work has recently [46] been proven for ***k*NN-friendly datasets*** (see Definition 2 in the Appendix for the precise definition). In practice, when $k \ll n$, the small neighborhoods are very likely to meet the *locally uniform* property of kNN-friendly datasets. The PIM-kd-tree improves the bounds from those works [35, 46] to the ones in Theorem 4.5.

Theorem 4.5 (kNN costs). *If the dataset is kNN-friendly, then executing a batch of $S = \Omega(P\log^2 P)$ k-NN queries over n points takes $O(Sk\log^* P)$ CPU work, $O(k\log S)$ CPU span, $O(\frac{Sk}{P}\log^* P)$ communication time and $O(\frac{Sk}{P}\log n)$ PIM time, all in expectation.*

Proof. [46] proved that a kNN query on a kNN-friendly dataset touches $O(k)$ leaf nodes in expectation. During the backtracking, the dual-way caching can be used for local traversing. Thus, we can combine Lemmas 3.7, 3.8, and 3.9 and Theorem 4.1 to get the final bounds. □

Approximate Nearest Neighbors (ANN). Suppose point q in the kd-tree is the k-th nearest neighbor of a given point p. Then a $(1+\epsilon)$-approximate k nearest neighbor query returns a point q' such that $dist(p,q) \le dist(p,q') \le (1+\epsilon)dist(p,q)$. A shared-memory sequential ANN algorithm [6] executes the same as kNN, except that it terminates when all untracked leaves are further than $r/(1+\epsilon)$ away. It takes $O(Sk\epsilon^{-D}\log n)$ work and communication. Meanwhile, PIM-kd-tree takes $O(k\epsilon^{-D}\log^* P)$ communication per query, and is PIM-balanced and work-efficient, as shown in Theorem 4.6.

Theorem 4.6 (ANN costs). *Executing a batch of $S = \Omega(P\log^2 P)$ $(1+\epsilon)$-approximate k nearest neighbor queries over n points takes $O(Sk\epsilon^{-D}\log^* P)$ CPU work, $O(k\log S)$ CPU span, $O(\frac{Sk\epsilon^{-D}}{P}\log^* P)$ communication time and $O(\frac{Sk\epsilon^{-D}}{P}\log n)$ PIM time, all in expectation.*

Proof. The priority search method has been proven to touch only $\Theta(k\epsilon^{-D})$ nodes per query and takes work $O(h)$ for each touched node [6], where h is the tree height. We can combine Lemmas 3.7, 3.8, and 3.9 and Theorem 4.1 to get the final bounds. □

Orthogonal Range Queries. An *orthogonal range query* takes a box whose borders are orthogonal to the dimensions of the dataset as input, and returns all the points in the dataset that lie in the given box. Our PIM-kd-tree supports such queries using a top-down search and keeping track of all candidate tree nodes whose representative bounding box intersects with the query box.

Lemma 4.7 gives the worst-case bound for the execution of orthogonal range queries. When we build a strictly-balanced PIM-kd-tree with $\alpha = O(1)/\log n$ and $h = \log n + O(1)$, the work bound in

Lemma 4.7 is reduced to $O\left(k + n^{\frac{D-1}{D}}\right)$. The total communication bound cannot be reduced using PIM since the original bound for shared-memory implementation [17] is already tight.

Lemma 4.7 (Orthogonal range query). *An orthogonal range query takes worst-case $O\left(k + 2^{\frac{D-1}{D}h}\right)$ PIM work and communication, where h is the PIM-kd-tree height and k is the number of points in the query box. The execution is PIM-balanced whp when the total number of touched nodes is $\Omega(P \log^2 P)$.*

5 LOWER BOUNDS AND OPTIMALITY

Trading-off Communication and Space. In our replication strategy for the PIM-kd-tree, we apply an intra-group caching policy to every group in the log-star decomposition. However, we can see that there exists a variant design where we only apply the policy to the first G groups. Such design leads to a space overhead of $O(nG)$ and a communication factor of $O(G + \log^{(G)} P)$.

In the previous sections, we adopt a communication-optimized version with $G = \log^* P$. Nevertheless, there exist space-optimized designs taking $O(ng) = O(n)$ linear space and a communication factor of $O(\log^{(g)} P)$, where g is an arbitrary positive constant.

Trading-off Communication and Batch Size. In all our bounds for operations (Theorems 4.1 and 4.3 to 4.6), we only require $S = \Omega(P \log^2 P)$. However, if we allow a batch size of $S = \Omega(P \log P \cdot C \log_C P)$, we can set the push-pull threshold to be $\Theta(C \log_C P)$, and use chunked trees rather than binary trees—combining up to $O(C)$ binary nodes into one B-tree-like node with fanout $O(C)$. Each chunk is stored on a single PIM module, and the caching strategy is applied in chunk granularity. Consequently, the search communication becomes $O(G + \log_C^{(G)} P)$ per query if the space overhead is $O(nG)$ for $1 \le G \le \log_C^* P$.

Increasing the batch size reduces the overheads of our approach. By using $C = \Theta(\log^{(\epsilon)} P)$ with arbitrary constant $\epsilon \ge 1$, we can achieve $O(1)$ communication per search and $O(n)$ space *whp*. However, in settings where queries arrive at the CPU over time, individual queries may wait longer to be processed, as a larger query batch needs to accumulate before processing starts.

Lower Bounds. We now prove that the trade-off between search communication, space and batch size in the PIM-kd-tree is ***optimal***. Our main results can be summarized into Theorem 5.1.

Theorem 5.1 (Lower Bound). *For a binary search tree containing n points and with height $\Theta(\log n)$, any PIM-based data structure that is skew-resistant for any search batch size $S = \Omega(P \log P \cdot C \log_C P)$ and supports $O(\log_C^{(G)} P)$ worst-case communication per search query (between any two arbitrary positions in the tree) must take $\Omega(nG)$ space, where $1 \le G \le \log_C^* P - \log_C^* \log_C^* P$.*

Proof. The worst-case search cost between two positions is at most double the LeafSearch cost, so we foucs on LeafSearch. We first prove the case for $S = \Theta(P \log^2 P)$, where the push-pull threshold is $O(\log P)$ and the master data chunk size is $O(1)$.

We transform the PIM LeafSearch problem into the cell-probe lower bound problem for dynamic succinct dictionaries [65]. We encode the search path for a LeafSearch query into a $\Theta(\log n)$ word, where a bit is 1 if the path takes the left child of an internal node and is 0 if the right child is taken. Then LeafSearch becomes querying for key-value pairs in a succinct dictionary.

Our group-by-group caching strategy expands in a same way as the optimal key arrangement tree in [65]. Each of the G groups of intra-group caching in the PIM aligns with a compressed auxiliary structure in the cell-probe model that require an $O(1)$ query cost (i.e., memory access in the cell-probe model [106]) to traverse. Thus, the space overhead in the PIM corresponds to the query cost in the cell-probe model. On the other hand, the dominant factor in communication $O(\log^{(G)} P)$ in PIM is the height of the lowest distributed levels without caching. In our transformation between the two problems, such lowest distributed levels in PIM align with the number of wasted bits per key in the cell-probe model to directly store uncompressed bits of information per key. Thus, the LeafSearch communication cost in the PIM Model corresponds to the space overhead in the cell-probe model.

Such a transformation guarantees that the trade-off between space and communication in the PIM Model shares the same lower bounds with the communication-space trade-off of dynamic succinct dictionaries in the cell-probe model. Thus, we directly apply Lemma 5.2 from [65] (below) to get the bound for $S = O(1)$.

For $S = P \log P \cdot C \log_C P$, the push-pull threshold and master data chunk size could be $C \log_C P$. The caching strategy is still optimal when using tree node fanout C. This gives the final results. □

Lemma 5.2 (Dynamic Succinct Dictionary [65]). *In a cell-probe model with word-size $w = \Theta(\log n)$ and $1 \le G \le \log^* P$, any dynamic succinct dictionary storing at most n keys with $O(\log^{(G)} P)$ wasted bits per key must have expected insertion, deletion or query time of at least $\Omega(G)$.*

Optimality. We conclude that our PIM-kd-tree is optimal *whp*: Our PIM-kd-tree has the same trade-off Pareto frontier as the optimal ones in Theorem 5.1, *except that* PIM-kd-tree has a *whp* term due to randomization for adversarial skew resistance.

6 APPLICATION TO CLUSTERING PROBLEMS

In this section, we showcase how to execute two widely-used clustering algorithms (density-peak clustering [84] and DBSCAN [31]) using the PIM-kd-tree and its techniques. Similar to §4, the clustering algorithms are work-efficient and PIM-balanced, and achieve nearly linear communication and space.

6.1 Density Peak Clustering

The density peak clustering (DPC) problem takes as input n points, a distance function $dist(\cdot, \cdot)$ and two user-input parameters $d_{cut} > 0$ and $\epsilon > 0$. It proceeds using the following three steps. (*i*) First, compute the density of each node x, which is the number of points in a ball centered at x and with radius d_{cut}. (*ii*) Next, for each point x, connect x to its dependent point, which is the nearest neighbor of x that has a higher density than x. (*iii*) Finally, remove all the connected edges with a distance higher than ϵ. The DPC will output a forest of trees as clusters.

A shared-memory parallel DPC [46] takes $O(n(1 + \rho_{avg}) \log n)$ average-case work and communication on a kNN-friendly dataset (see Definition 2 or [46]), where ρ_{avg} is the average cubical density

with side length $2d_{cut}$. We will show how to improve this bound using PIM systems.

Density Computation. To calculate the density of each point, we build a PIM-kd-tree on the n points in the dataset, and construct a radius search with distance d_{cut}. The radius search performs in a similar way to kNN, where we first locate the leaf node and then backtrack the kd-tree to traverse all cells that intersect with the d_{cut}-radius sphere.

Dependent Point Computation. To connect each point to its dependent point, we construct a *priority search kd-tree* [39, 46] on the PIM system. In a priority search kd-tree, each point is associated with a priority, and each internal node is associated with a value that records the highest priority among all points in its represented bounding box (i.e., its subtree). In DPC, this priority value is the density we calculated in step (*i*).

We only need to construct a *static* priority search kd-tree for DPC, so we can use Algorithm 2 for the construction. We only need to update all the priority values when we partition the points in each recursive calls of a construction, and all asymptotic bounds in Theorem 3.5 are not changed.

After the priority search kd-tree is constructed, we conduct a 1NN priority search for each point, where the backtracking only traverses the nodes with a higher priority value.

Cluster Construction. After we connect each point to its dependent point, we remove all edges that exceed distance ϵ in parallel and conduct a parallel connected components algorithm [92].

Cost Analysis. Theorem 6.1 presents the costs of our PIM-friendly DPC algorithm, showing that our design is work-efficient, PIM-offloaded, PIM-balanced, communication-reduced and space-efficient.

Theorem 6.1 (DPC costs). *For $n = \Omega(P\log^3 P)$ kNN-friendly points with average density ρ, a DPC on PIM takes $O(n(\log P + \log\log n + \rho\log^* P))$ CPU work, $O(\log^3 n)$ CPU span, $O(\frac{n}{P}(\log\frac{n}{P} + \rho\log n))$ PIM time and $O(\frac{n}{P}(1+\rho)\log^* P)$ communication time, all in expectation, and takes $O(n\log^* P)$ space.*

Proof. The construction of PIM-kd-trees takes $O(n(\log P + \log\log n))$ CPU work, $O(\log^2 P + \log n)$ CPU span, $O(\frac{n}{P}\log\frac{n}{P})$ PIM execution time and $O(\frac{n}{P}\log^* P)$ communication time, all *whp* (Theorem 3.5). For step (*i*), each range query touches $O(\rho)$ leaf nodes on average, which takes $O(n\rho\log n)$ PIM work and $O(n\rho\log^* P)$ communication. In step (*ii*), it is proved [46] that each priority search touches $O(1)$ leaf nodes. Finding connected components [92] takes an expected $O(n)$ CPU work, $O(\log^3 n)$ span and $O(\frac{n}{P})$ communication time if each vertex/edge is hashed to a random PIM module. □

6.2 DBSCAN

The DBSCAN (density-based spatial clustering of applications with noise) problem takes as input n points, a distance function $dist(\cdot,\cdot)$ and two user-input parameters $\epsilon > 0$ and $k > 0$. A point x is labeled as a ***core point*** iff $|\{x' \mid dist(x',x) \le \epsilon\}| \ge k$. DBSCAN returns a set of clusters where each core point belongs to exactly one cluster. Two core points x and y belong to the same cluster iff there exist a list of core points $x = x_1, x_2, \cdots, x_j = y$ connecting x and y s.t. $dist(x_i, x_{i+1}) \le \epsilon$ for $i = 1..j-1$. A non-core point can belong to multiple clusters that are less than ϵ distance away. A non-core point is labeled as a ***border point*** if it belongs to at least one cluster, or labeled as a ***noise point*** if it does not belong to any cluster.

Sequential 2-dimensional DBSCAN takes $O(n(k+\log n))$ work [29, 37, 41], and a shared-memory work-efficient parallel version takes $O(\log n)$ span and $O(n\log_M n)$ communication *whp*. We will improve these bounds for $D = 2$ using PIM systems, as in Theorem 6.3. High-dimensional DBSCANs ($D \ge 3$) have a lower bound of polynomial work [37], which is hard to improve even using PIM.

To implement 2d-DBSCAN on PIM, we follow a four-step execution flow: (*i*) grid computation, (*ii*) core marking, (*iii*) cell graph construction, and (*iv*) cluster construction.

Grid Computation. All points are placed into disjoint cells with side length $\epsilon/\sqrt{2}$. Each cell is stored on a random PIM module using a hash mapping. If the number of points in a cell exceeds $n/(P\log P)$, we recursively divide the cell into sub-cells until the number in each sub-cell is $O(n/(P\log P))$. We map the sub-cells to random PIM modules and also maintain the tree structure of the division.

Mark Cores. When the number of points in a cell is larger than k, then all these points are labeled as core points. Otherwise, all these points are collocated with their neighboring cells to count the number of neighbors in their ϵ-neighborhood. We use push-pull search here to always send the cell with fewer points to the cell with more points when collocating cell points.

Cell Graph Construction. If two neighboring cells have a pair of core points with distance $\le \epsilon$, then all the points in these two cells belong to the same cluster. We link an edge between the two cells, and eventually output a cell graph. We solve the problem using the unit-spherical emptiness checking (USEC) with line separation method from [101], which first performs a per-cell sort over all nodes on one axis, and builds in parallel an index of wavefronts to check the intersection within distance ϵ. Similar to core marking, we still use push-pull search in this step for load balance.

Cluster Construction. After a cell graph is constructed, a connected components algorithm is executed to collapse all connected cells into the same cluster, which gives the final clustering result.

Cost of the Sorting Step in USEC. The following lemma and proof present and analyze a PIM-friendly algorithm for performing the sorting step of USEC.

Lemma 6.2 (Sorting). *Sorting m elements on PIM together with other $n = \Omega(P\log P)$ work in a batch, takes $O(m\log P)$ CPU work, $O(\log m)$ CPU span, $O(m\log(\frac{m}{P}+1))$ PIM work and $O(m)$ communication, and is PIM-balanced whp.*

Proof. (*i*) If $m = O(n/(P\log P))$, send the m elements to one PIM module and sort there.

(*ii*) If $m = \Omega(P\log^2 P + n/(P\log P))$, then first sample $P\log P$ elements to be sorted in the CPU cache and output P ranges that have equivalent numbers of elements per range *whp*. Then assign each range to a PIM module, and send all the m elements to the corresponding range to be sorted. Searching through the sample structure takes $O(m\log P)$ work, and the sorting takes $O(m\log\frac{m}{P})$ PIM work and is balanced across the PIM modules.

(*iii*) If $m = \Omega(n/(P\log P))$ but $m = O(P\log^2 P)$, then the m elements fit in the CPU cache. We divide the m elements into multiple groups, each containing $n/(P\log P)$ elements. Each group is sent

to a random PIM module to be sorted, and then different groups are merged on the CPU. The recursive level of the CPU merging is $O(\log m) = O(\log P)$, so the CPU work is $O(m \log P)$. □

Cost Analysis. Theorem 6.3 presents the costs of our PIM-friendly DBSCAN, showing that our design is work-efficient, PIM-offloaded, PIM-balanced, communication-reduced and space-efficient.

Theorem 6.3 (DBSCAN cost). *A 2-dimensional (ϵ, k)-DBSCAN on PIM for $n = \Omega(P \log^3 P)$ points takes $O(n \log P)$ CPU work, $O(\log^3 n)$ CPU span, $O(\frac{n}{P}(k + \log \frac{n}{P}))$ PIM time and $O(\frac{n}{P})$ communication time, all whp, and takes $O(n)$ space.*

Proof. The grid computation takes $O(n)$ work and guarantees all later steps are PIM-balanced. Core marking takes $O(kn)$ work, because each cell receives at most $O(k)$ points from all its neighbors, and takes $O(n)$ communication due to push-pull search. USEC takes $O(n)$ work for wavefront building and all other substeps, except for its sorting substep, which takes $O(n \log P)$ CPU work, $O(n \log \frac{n}{P})$ PIM work and $O(n)$ communication (Lemma 6.2). All the bounds are *whp*. Because we use recursive cells with small enough sizes and push-pull search, Theorem 4.1 and lemmas 3.7 to 3.9 guarantee that all steps are PIM-balanced *whp*. □

7 DISCUSSION

A Generalized Design for Search Trees on PIM. Our optimal search tree design, incorporating log-star decomposition and intra-group dual-way caching, can be directly generalized to many types of (semi-)balanced trees, including and not limited to, B+-trees [55], skip lists [54], oct-trees [14], RC-trees [3], etc. For highly imbalanced trees, such as radix trees, the existence of the PIM-trie design [57] suggests that our design may be adaptable to such trees. Meanwhile, for other trees that require different construction/update schema (such as plane sweep trees [8, 82] or log-structured merge-trees [77]), we hypothesize that while the main structure of PIM-kd-tree remains applicable, significant modifications to the construction/update schema would be necessary.

The generalized PIM-kd-tree design applied to all these trees reduces the communication factor from $O(\log n)$, $O(\log_M n)$ or $O(\log \frac{n}{S})$ in external-memory models to a PIM-model factor of $O(\log^* P)$. At the same time, CPU work is asymptotically reduced, with the majority of the required work offloaded to the PIM modules in a work-efficient and PIM-balanced manner (even under adversarial skew). These benefits come at the cost of a nearly constant space overhead of $O(\log^* P)$.

Round Complexity. Another metric of interest is the *round complexity*, i.e., the number of bulk-synchronous rounds [54]. In the bulk-synchronous setting, an off-chip communication c_2 that depends on the results of an off-chip communication c_1 must be in a later round. In addition, the bounded CPU cache memory provides a limit to how many messages to PIM modules can be buffered at the CPU in between communication rounds. Depending on the cache size M, additional rounds may be required to flush these messages, even when there is no communication dependency.

Accordingly, we define a work-span-style analysis for rounds as follows. Define the *communication span* of an algorithm as the longest chain of off-chip communication dependencies. Then an algorithm with communication amount c and communication span s takes $\Omega(\frac{c}{M} + s)$ rounds. The analysis can be applied to both PIM and shared-memory algorithms.

For all our PIM-based algorithms, we reduce the total communication amount over the prior shared-memory algorithms (Table 1). Meanwhile, we reduce the communication span (the off-chip search path length) to $O(\log P)$ from the $O(\log n)$ required in the shared-memory algorithms. Because each of our rounds is load-balanced and can take full use of the entire cache, we achieve a round complexity of $\Theta(\frac{c}{M} + s)$ for all our algorithms.

8 CONCLUSION

This paper presents PIM-kd-tree, the first space-partitioning index designed for processing-in-memory (PIM) systems. It supports construction, LeafSearch, dynamic updates, orthogonal range query and k nearest neighbor (kNN) search, and can be applied to clustering problems such as DPC and DBSCAN. PIM-kd-tree achieves work efficiency, high load balance, low off-chip communication and nearly linear space overhead in the PIM model. We prove that its design is optimal *whp* in balancing search communication, space and batch size. Key techniques include (*i*) a log-star tree decomposition based on subtree sizes, (*ii*) an intra-group caching strategy that asymptotically reduces communication with low update and space overhead while enabling guaranteed load balance, (*iii*) an approximate counter design with low update overhead and high accuracy, serving as auxiliary metadata, and (*iv*) an efficient construction, acting as a core subroutine for operations. Future research directions include extending the optimal tree design to other tree structures in the PIM settings, and implementing a space-partitioning index on a real-world PIM system.

ACKNOWLEDGMENTS

This research was supported by NSF grants CCF-1919223, CCF-2103483, CCF-2119352, CCF-2339310, CCF-2403235, CNS-2211882 and CNS-2317194, Parallel Data Lab (PDL) Consortium (Amazon, Bloomberg, Datadog, Google, Honda, Intel, Jane Street, LayerZero, Meta, Microsoft, Oracle, Pure Storage, Salesforce, Samsung, and Western Digital), the Lee-Stanziale Ohana Endowed Fellowship, and the Michel and Kathy Doreau Fellowship.

A DEFINITION OF kNN-FRIENDLY DATASETS

Definition 2 (kNN-friendly dataset [46]). *Let ϵ_1 and ϵ_2 be positive constants. A kNN-friendly dataset has the following constraints:*

1. ***Constant Dimension**: $D = O(1)$.*
2. ***Compact Cells**: For any cell represented by a kd-tree node containing fewer than $(1+\epsilon_2)k$ points, the ratio of the longest side to the shortest side is bounded by $(1+\epsilon_1)$.*
3. ***Locally Uniform**: Suppose the dataset P is sampled from a probabilistic density function μ. μ should be constant in expectation over all samples of P within a hyper-spherical region centered at the kNN query point with radius $\leq 3R\sqrt{D}$, where R is the diagonal length of the cell corresponding to the smallest subtree that contains the query point and more than k points.*
4. ***Bounded Expansion Ratio**: For each kd-tree node containing fewer than k nodes, its sibling contains at most $(1+\epsilon_2)k$ points.*

REFERENCES

[1] 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* (San Diego, California, USA) *(SCG '03)*. Association for Computing Machinery, New York, NY, USA, 237–245. https://doi.org/10.1145/777792.777828

[2] Shahanur Alam, Chris Yakopcic, and Tarek M. Taha. 2022. Memristor Based Federated Learning for Network Security on the Edge using Processing in Memory (PIM) Computing. In *2022 International Joint Conference on Neural Networks (IJCNN)*. 1–8. https://doi.org/10.1109/IJCNN55064.2022.9891986

[3] Daniel Anderson and Guy E. Blelloch. 2024. Deterministic and Low-Span Work-Efficient Parallel Batch-Dynamic Trees. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures* (Nantes, France) *(SPAA '24)*. Association for Computing Machinery, New York, NY, USA, 247–258. https://doi.org/10.1145/3626183.3659976

[4] Md Tanvir Arafin and Zhaojun Lu. 2020. Security Challenges of Processing-In-Memory Systems. In *Proceedings of the 2020 on Great Lakes Symposium on VLSI* (Virtual Event, China) *(GLSVLSI '20)*. Association for Computing Machinery, New York, NY, USA, 229–234. https://doi.org/10.1145/3386263.3411365

[5] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. 2001. Thread Scheduling for Multiprogrammed Multiprocessors. *Theory of Computing Systems* 34, 2 (2001), 115–144.

[6] Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Y. Wu. 1998. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *J. ACM* 45, 6 (nov 1998), 891–923.

[7] Kazi Asifuzzaman, Narasinga Rao Miniskar, Aaron R. Young, Frank Liu, and Jeffrey S. Vetter. 2023. A survey on processing-in-memory techniques: Advances and challenges. *Memories - Materials, Devices, Circuits and Systems* 4 (2023), 100022. https://doi.org/10.1016/j.memori.2022.100022

[8] Mikhail J. Atallah and Michael T. Goodrich. 1986. Efficient plane sweeping in parallel. In *Proceedings of the Second Annual Symposium on Computational Geometry* (Yorktown Heights, New York, USA) *(SCG '86)*. Association for Computing Machinery, New York, NY, USA, 216–225. https://doi.org/10.1145/10515.10539

[9] Alexander Baumstark, Muhammad Attahir Jibril, and Kai-Uwe Sattler. 2023. Adaptive Query Compilation with Processing-in-Memory. In *2023 IEEE 39th International Conference on Data Engineering Workshops (ICDEW)*. 191–197. https://doi.org/10.1109/ICDEW58674.2023.00035

[10] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* 18, 9 (1975).

[11] Arthur Bernhardt, Andreas Koch, and Ilia Petrov. 2023. pimDB: From Main-Memory DBMS to Processing-In-Memory DBMS-Engines on Intelligent Memories. In *Proceedings of the 19th International Workshop on Data Management on New Hardware* (Seattle, WA, USA) *(DaMoN '23)*. Association for Computing Machinery, New York, NY, USA, 44–52. https://doi.org/10.1145/3592980.3595312

[12] Wenhao Bi, Junwen Ma, Xudong Zhu, Weixiang Wang, and An Zhang. 2022. Cloud service selection based on weighted KD tree nearest neighbor search. *Applied Soft Computing* 131 (2022), 109780. https://doi.org/10.1016/j.asoc.2022.109780

[13] Sebastian Bindick, Maik Stiebler, and Manfred Krafczyk. 2011. Fast kd-tree-based hierarchical radiosity for radiative heat transport problems. *Internat. J. Numer. Methods Engrg.* 86, 9 (2011), 1082–1100. https://doi.org/10.1002/nme.3091 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/nme.3091

[14] Guy E Blelloch and Magdalen Dobson. 2022. Parallel Nearest Neighbors in Low Dimensions with Batch Updates. In *2022 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 195–208.

[15] Guy E. Blelloch, Daniel Ferizovic, and Yihan Sun. 2016. Just Join for Parallel Ordered Sets. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 253–264.

[16] Guy E. Blelloch, Jeremy T. Fineman, Yan Gu, and Yihan Sun. 2020. Optimal Parallel Algorithms in the Binary-Forking Model. *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)* (2020), 89–102.

[17] Guy E Blelloch, Yan Gu, Julian Shun, and Yihan Sun. 2018. Parallel Write-Efficient Algorithms and Data Structures for Computational Geometry. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 235–246.

[18] Robert D. Blumofe and Charles E. Leiserson. 1998. Space-Efficient Scheduling of Multithreaded Computations. *SIAM J. on Computing* 27, 1 (1998).

[19] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling Multithreaded Computations by Work Stealing. *J. ACM* 46, 5 (1999).

[20] Yixi Cai, Wei Xu, and Fu Zhang. 2021. ikd-Tree: An Incremental K-D Tree for Robotic Applications. arXiv:2102.10808 [cs.RO] https://arxiv.org/abs/2102.10808

[21] Pedro Carrinho, Oscar Ferraz, João Dinis Ferreira, Yann Falevoz, Vitor Silva, and Gabriel Falcao. 2024. Processing Multi-Layer Perceptrons In-Memory. In *2024 IEEE Workshop on Signal Processing Systems (SiPS)*. 7–12. https://doi.org/10.1109/SiPS62058.2024.00010

[22] Liang-Chi Chen, Chien-Chung Ho, and Yuan-Hao Chang. 2023. UpPipe: A Novel Pipeline Management on In-Memory Processors for RNA-seq Quantification. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1109/DAC56929.2023.10247915

[23] Qile P Chen, Bai Xue, and J Ilja Siepmann. 2017. Using the k-d tree data structure to accelerate Monte Carlo simulations. *Journal of Chemical Theory and Computation* 13, 4 (2017), 1556–1565.

[24] Sitian Chen, Haobin Tan, Amelie Chi Zhou, Yusen Li, and Pavan Balaji. 2024. UpDLRM: Accelerating Personalized Recommendation using Real-World PIM Architecture. In *Proceedings of the 61st ACM/IEEE Design Automation Conference* (San Francisco, CA, USA) *(DAC '24)*. Association for Computing Machinery, New York, NY, USA, Article 211, 6 pages. https://doi.org/10.1145/3649329.3658266

[25] Yewang Chen, Lida Zhou, Yi Tang, Jai Puneet Singh, Nizar Bouguila, Cheng Wang, Huazhen Wang, and Jixiang Du. 2019. Fast neighbor search by using revised k-d tree. *Information Sciences* 472 (2019), 145–162. https://doi.org/10.1016/j.ins.2018.09.012

[26] Jiwon Choe, Andrew Crotty, Tali Moreshet, Maurice Herlihy, and R Iris Bahar. 2022. Hybrids: Cache-conscious concurrent data structures for near-memory processing architectures. In *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures*. 321–332.

[27] Jiwon Choe, Amy Huang, Tali Moreshet, Maurice Herlihy, and R. Iris Bahar. 2019. Concurrent Data Structures with Near-Data-Processing: an Architecture-Aware Implementation. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 297–308.

[28] Byeongjun Choi, Byungjoon Chang, and Insung Ihm. 2013. Improving Memory Space Efficiency of Kd-tree for Real-time Ray Tracing. In *Computer Graphics Forum*, Vol. 32. Wiley Online Library, 335–344.

[29] Mark de Berg, Ade Gunawan, and Marcel Roeloffzen. 2019. Faster DBSCAN and HDBSCAN in low-dimensional Euclidean spaces. *International Journal of Computational Geometry & Applications* 29, 01 (2019), 21–47.

[30] Xiaojun Dong, Yunshu Wu, Zhongqi Wang, Laxman Dhulipala, Yan Gu, and Yihan Sun. 2023. High-Performance and Flexible Parallel Algorithms for Semisort and Related Problems. In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures* (Orlando, FL, USA) *(SPAA '23)*. Association for Computing Machinery, New York, NY, USA, 341–353. https://doi.org/10.1145/3558481.3591071

[31] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. In *kdd*, Vol. 96. 226–231.

[32] Dina Fakhry, Mohamed Abdelsalam, M. Watheq El-Kharashi, and Mona Safar. 2023. An HBM3 Processing-In-Memory Architecture for Security and Data Integrity: Case Study. In *Green Sustainability: Towards Innovative Digital Transformation*, Dalia Magdi, Ahmed Abou El-Fetouh, Mohamed Mamdouh, and Amit Joshi (Eds.). Springer Nature Singapore, Singapore, 281–293.

[33] Oscar Ferraz, Gabriel Falcao, and Vitor Silva. 2024. In-Memory Bit Flipping LDPC Decoding. In *2024 32nd European Signal Processing Conference (EUSIPCO)*. 706–710. https://doi.org/10.23919/EUSIPCO63174.2024.10715253

[34] Oscar Ferraz, Yann Falevoz, Vitor Silva, and Gabriel Falcao. 2023. Unlocking the Potential of LDPC Decoders with PiM Acceleration. In *2023 57th Asilomar Conference on Signals, Systems, and Computers*. 1579–1583. https://doi.org/10.1109/IEEECONF59524.2023.10476816

[35] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. 1977. An Algorithm for Finding Best Matches in Logarithmic Expected Time. *ACM Trans. Math. Softw.* 3, 3 (Sept. 1977), 209–226.

[36] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. 1999. Cache-Oblivious Algorithms. In *IEEE Symposium on Foundations of Computer Science (FOCS)*. 285–298.

[37] Junhao Gan and Yufei Tao. 2017. On the hardness and approximation of Euclidean DBSCAN. *ACM Transactions on Database Systems (TODS)* 42, 3 (2017), 1–45.

[38] Sahar Ghoflsaz Ghinani, Jingyao Zhang, and Elaheh Sadredini. 2025. Enabling Low-Cost Secure Computing on Untrusted In-Memory Architectures. arXiv:2501.17292 [cs.CR] https://arxiv.org/abs/2501.17292

[39] Matthias Gross, Carsten Lojewski, Martin Bertram, and Hans Hagen. 2007. Fast implicit KD-trees: accelerated isosurface ray tracing and maximum intensity projection for large scalar fields. In *Proceedings of the Ninth IASTED International Conference on Computer Graphics and Imaging* (Innsbruck, Austria) *(CGIM '07)*. ACTA Press, USA, 67–74.

[40] Yan Gu, Zachary Napier, Yihan Sun, and Letong Wang. 2022. Parallel Cover Trees and their Applications. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 259–272.

[41] Ade Gunawan and M de Berg. 2013. A faster algorithm for DBSCAN. *Master's thesis* (2013).

[42] Harshita Gupta, Mayank Kabra, Juan Gómez-Luna, Konstantinos Kanellopoulos, and Onur Mutlu. 2023. Evaluating Homomorphic Operations on a Real-World Processing-In-Memory System. In *2023 IEEE International Symposium on Workload Characterization (IISWC)*. 211–215. https://doi.org/10.1109/IISWC59245.2023.00030

[43] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F. Oliveira, and Onur Mutlu. 2022. Benchmarking a New Paradigm: Experimental Analysis and Characterization of a Real Processing-in-Memory System. *IEEE*

Access 10 (2022), 52565–52608. https://doi.org/10.1109/ACCESS.2022.3174101

[44] Wenfeng Hou, Daiwei Li, Chao Xu, Haiqing Zhang, and Tianrui Li. 2018. An Advanced k Nearest Neighbor Classification Algorithm Based on KD-tree. In *2018 IEEE International Conference of Safety Produce Informatization (IICSPI)*. 902–905. https://doi.org/10.1109/IICSPI.2018.8690508

[45] Yifan Hua, Shengan Zheng, Weihan Kong, Cong Zhou, Kaixin Huang, Ruoyan Ma, and Linpeng Huang. 2024. RADAR: A Skew-Resistant and Hotness-Aware Ordered Index Design for Processing-in-Memory Systems. *IEEE Transactions on Parallel and Distributed Systems* 35, 9 (2024), 1598–1614. https://doi.org/10.1109/TPDS.2024.3424853

[46] Yihao Huang, Shangdi Yu, and Julian Shun. 2023. Faster Parallel Exact Density Peaks Clustering. arXiv:2305.11335 [cs.DC] https://arxiv.org/abs/2305.11335

[47] Yuan Huang, Zhiqin Zhao, Conghui Qi, Zaiping Nie, and Qing Huo Liu. 2018. Fast Point-Based KD-Tree Construction Method for Hybrid High Frequency Method in Electromagnetic Scattering. *IEEE Access* 6 (2018), 38348–38355. https://doi.org/10.1109/ACCESS.2018.2853659

[48] Bongjoon Hyun, Taehun Kim, Dongjae Lee, and Minsoo Rhu. 2024. Pathfinding Future PIM Architectures by Demystifying a Commercial PIM Technology. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 263–279. https://doi.org/10.1109/HPCA57654.2024.00029

[49] Intel. 2025. Intel In-Memory Analytics Accelerator (Intel IAA). https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/in-memory-analytics-accelerator.html. Accessed February 2025.

[50] Joe Jeddeloh and Brent Keeth. 2012. Hybrid memory cube new DRAM architecture increases density and performance. In *2012 Symposium on VLSI Technology (VLSIT)*. 87–88. https://doi.org/10.1109/VLSIT.2012.6242474

[51] Muhammad Attahir Jibril, Hani Al-Sayeh, and Kai-Uwe Sattler. 2024. Accelerating Aggregation Using a Real Processing-in-Memory System. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. 3920–3932. https://doi.org/10.1109/ICDE60146.2024.00300

[52] Lu Jie, Chen Hongchang, Sun Penghao, Hu Tao, and Zhang Zhen. 2021. OrderSketch: An unbiased and fast sketch for frequency estimation of data streams. *Computer Networks* 201 (2021), 108563.

[53] Jaemin Jo, Jinwook Seo, and Jean-Daniel Fekete. 2017. A progressive k-d tree for approximate k-nearest neighbors. In *2017 IEEE Workshop on Data Systems for Interactive Analysis (DSIA)*. 1–5. https://doi.org/10.1109/DSIA.2017.8339084

[54] Hongbo Kang, Phillip B Gibbons, Guy E Blelloch, Laxman Dhulipala, Yan Gu, and Charles McGuffey. 2021. The Processing-in-Memory Model. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*. 295–306.

[55] Hongbo Kang, Yiwei Zhao, Guy E. Blelloch, Laxman Dhulipala, Yan Gu, Charles McGuffey, and Phillip B. Gibbons. 2022. PIM-Tree: A Skew-Resistant Index for Processing-in-Memory. *Proc. VLDB Endow.* 16, 4 (dec 2022), 946–958. https://doi.org/10.14778/3574245.3574275

[56] Hongbo Kang, Yiwei Zhao, Guy E. Blelloch, Laxman Dhulipala, Yan Gu, Charles McGuffey, and Phillip B. Gibbons. 2022. PIM-tree: A Skew-resistant Index for Processing-in-Memory. *arXiv preprint* (2022). https://doi.org/10.48550/ARXIV.2211.10516

[57] Hongbo Kang, Yiwei Zhao, Guy E. Blelloch, Laxman Dhulipala, Yan Gu, Charles McGuffey, and Phillip B. Gibbons. 2023. PIM-Trie: A Skew-Resistant Trie for Processing-in-Memory. In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures* (Orlando, FL, USA) *(SPAA '23)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3558481.3591070

[58] Yoon-Sig Kang, Jae-Ho Nah, Woo-Chan Park, and Sung-Bong Yang. 2013. gkDtree: A group-based parallel update kd-tree for interactive ray tracing. *Journal of Systems Architecture* 59, 3 (2013), 166–175.

[59] Hyoungjoo Kim, Yiwei Zhao, Andrew Pavlo, and Phillip B. Gibbons. 2025. No Cap, This Memory Slaps: Breaking Through the Memory Wall of Transactional Database Systems with Processing-in-Memory. *Proc. VLDB Endow.* (2025).

[60] Dominique Lavenier, Jean-Francois Roy, and David Furodet. 2016. DNA mapping using Processor-in-Memory architecture. In *2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. 1429–1435. https://doi.org/10.1109/BIBM.2016.7822732

[61] Dongjae Lee, Bongjoon Hyun, Taehun Kim, and Minsoo Rhu. 2024. PIM-MMU: A Memory Management Unit for Accelerating Data Transfers in Commercial PIM Systems. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 627–642. https://doi.org/10.1109/MICRO61859.2024.00053

[62] Haoyu Li, Qizhi Chen, Yixin Zhang, Tong Yang, and Bin Cui. 2022. Stingy sketch: a sketch framework for accurate and fast frequency estimation. *Proceedings of the VLDB Endowment* 15, 7 (2022), 1426–1438.

[63] Jizhou Li, Zikun Li, Yifei Xu, Shiqi Jiang, Tong Yang, Bin Cui, Yafei Dai, and Gong Zhang. 2020. WavingSketch: An Unbiased and Generic Sketch for Finding Top-k Items in Data Streams. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Virtual Event, CA, USA) *(KDD '20)*. Association for Computing Machinery, New York, NY, USA, 1574–1584. https://doi.org/10.1145/3394486.3403208

[64] Shihua Li, Jingxian Wang, Zuqin Liang, and Lian Su. 2016. Tree point clouds registration using an improved ICP algorithm based on kd-tree. In *2016 IEEE International Geoscience and Remote Sensing Symposium (IGARSS)*. 4545–4548. https://doi.org/10.1109/IGARSS.2016.7730186

[65] Tianxiao Li, Jingxun Liang, Huacheng Yu, and Renfei Zhou. 2023. Tight Cell-Probe Lower Bounds for Dynamic Succinct Dictionaries. In *2023 IEEE 64th Annual Symposium on Foundations of Computer Science (FOCS)*. 1842–1862. https://doi.org/10.1109/FOCS57990.2023.00112

[66] Chaemin Lim, Suhyun Lee, Jinwoo Choi, Jounghoo Lee, Seongyeon Park, Hanjun Kim, Jinho Lee, and Youngsok Kim. 2023. Design and Analysis of a Processing-in-DIMM Join Algorithm: A Case Study with UPMEM DIMMs. *Proc. ACM Manag. Data* 1, 2, Article 113 (June 2023), 27 pages. https://doi.org/10.1145/3589258

[67] Xingyu Liu, Yangdong Deng, Yufei Ni, and Zonghui Li. 2015. FastTree: A hardware KD-tree construction acceleration engine for real-time ray tracing. In *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1595–1598.

[68] Zhiyu Liu, Irina Calciu, Maurice Herlihy, and Onur Mutlu. 2017. Concurrent Data Structures for Near-memory Computing. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 235–245.

[69] Andrew McGregor, Sofya Vorotnikova, and Hoa T. Vu. 2016. Better Algorithms for Counting Triangles in Data Streams. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (San Francisco, California, USA) *(PODS '16)*. Association for Computing Machinery, New York, NY, USA, 401–411. https://doi.org/10.1145/2902251.2902283

[70] Ziyang Men, Zheqi Shen, Yan Gu, and Yihan Sun. 2025. Pkd-tree: Parallel kd-tree with Batch Updates. In *ACM International Conference on Management of Data (SIGMOD)*.

[71] Meven Mognol, Dominique Lavenier, and Julien Legriel. 2024. Parallelization of the Banded Needleman & Wunsch Algorithm on UPMEM PiM Architecture for Long DNA Sequence Alignment. In *Proceedings of the 53rd International Conference on Parallel Processing* (Gotland, Sweden) *(ICPP '24)*. Association for Computing Machinery, New York, NY, USA, 1062–1071. https://doi.org/10.1145/3673038.3673094

[72] Robert Morris. 1978. Counting large numbers of events in small registers. *Commun. ACM* 21, 10 (1978), 840–842.

[73] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. 2023. *A Modern Primer on Processing in Memory*. Springer Nature Singapore, Singapore, 171–243. https://doi.org/10.1007/978-981-16-7487-7_7

[74] Onur Mutlu, Ataberk Olgun, Geraldo F. Oliveira, and Ismail E. Yuksel. 2024. Memory-Centric Computing: Recent Advances in Processing-in-DRAM (Invited). In *2024 IEEE International Electron Devices Meeting (IEDM)*. 1–4. https://doi.org/10.1109/IEDM50854.2024.10873410

[75] Mpoki Mwaisela. 2024. PhD Forum: Efficient Privacy-Preserving Processing via Memory-Centric Computing. In *2024 43rd International Symposium on Reliable Distributed Systems (SRDS)*. 322–325. https://doi.org/10.1109/SRDS64841.2024.00039

[76] Mohammed Otair. 2013. Approximate k-nearest neighbour based spatial clustering using k-d tree. arXiv:1303.1951 [cs.DB] https://arxiv.org/abs/1303.1951

[77] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33 (1996), 351–385.

[78] Reid Pinkham, Shuqing Zeng, and Zhengya Zhang. 2020. QuickNN: Memory and Performance Optimization of k-d Tree Based Nearest Neighbor Search for 3D Point Clouds. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 180–192. https://doi.org/10.1109/HPCA47549.2020.00024

[79] Octavian Procopiuc, Pankaj K Agarwal, Lars Arge, and Jeffrey Scott Vitter. 2003. Bkd-tree: A dynamic scalable kd-tree. In *Advances in Spatial and Temporal Databases: 8th International Symposium, SSTD 2003, Santorini Island, Greece, July 2003. Proceedings 8*. Springer, 46–65.

[80] Sanguthevar Rajasekaran. 1991. Randomized algorithms for packet routing on the mesh. *Advances in Parallel Algorithms* (1991).

[81] Parikshit Ram and Kaushik Sinha. 2019. Revisiting kd-tree for Nearest Neighbor Search. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Anchorage, AK, USA) *(KDD '19)*. Association for Computing Machinery, New York, NY, USA, 1378–1388. https://doi.org/10.1145/3292500.3330875

[82] John H Reif and Sandeep Sen. 1992. Optimal randomized parallel algorithms for computational geometry. *Algorithmica* 7, 1 (1992), 91–117.

[83] Steve Rhyner, Haocong Luo, Juan Gómez-Luna, Mohammad Sadrosadati, Jiawei Jiang, Ataberk Olgun, Harshita Gupta, Ce Zhang, and Onur Mutlu. 2024. PIM-Opt: Demystifying Distributed Optimization Algorithms on a Real-World Processing-In-Memory System. In *Proceedings of the 2024 International Conference on Parallel Architectures and Compilation Techniques* (Long Beach, CA, USA) *(PACT '24)*. Association for Computing Machinery, New York, NY, USA, 201–218. https://doi.org/10.1145/3656019.3676947

[84] Alex Rodriguez and Alessandro Laio. 2014. Clustering by fast search and find of density peaks. *science* 344, 6191 (2014), 1492–1496.

[85] W. Saftly, M. Baes, and P. Camps. 2014. Hierarchical octree and k-d tree grids for 3D radiative transfer simulations. *A&A* 561 (2014), A77. https://doi.org/10.1051/0004-6361/201322593

[86] Samsung. 2025. Samsung PIM Technology. https://semiconductor.samsung.com/technologies/memory/pim/. Accessed February 2025.

[87] Peter Sanders. 1996. On the Competitive Analysis of Randomized Static Load Balancing. In *Workshop on Randomized Parallel Algorithms (RANDOM)*.

[88] Johannes Schauer and Andreas Nüchter. 2015. Collision detection between point clouds using an efficient k-d tree implementation. *Advanced Engineering Informatics* 29, 3 (2015), 440–458. https://doi.org/10.1016/j.aei.2015.03.007

[89] Jason Sewall, Jatin Chhugani, Changkyu Kim, Nadathur Satish, and Pradeep Dubey. 2011. PALM: Parallel architecture-friendly latch-free modifications to B+ trees on many-core processors. *Proceedings of the VLDB Endowment* 4, 11 (2011), 795–806.

[90] Yunxiao Shan, Shu Li, Fuxiang Li, Yuxin Cui, Shuai Li, Ming Zhou, and Xiang Li. 2022. A density peaks clustering algorithm with sparse search and Kd tree. *IEEE Access* 10 (2022), 74883–74901.

[91] Shunchen Shi, Xueqi Li, Zhaowu Pan, Peiheng Zhang, and Ninghui Sun. 2024. CoPIM: A Collaborative Scheduling Framework for Commodity Processing-in-memory Systems. In *2024 IEEE 42nd International Conference on Computer Design (ICCD)*. 44–51. https://doi.org/10.1109/ICCD63220.2024.00018

[92] Julian Shun, Laxman Dhulipala, and Guy E. Blelloch. 2014. A Simple and Practical Linear-work Parallel Algorithm for Connectivity. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 143–153.

[93] Guy L Steele Jr and Jean-Baptiste Tristan. 2016. Adding approximate counters. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 1–12.

[94] Guy L Steele Jr and Jean-Baptiste Tristan. 2018. Method and system for latent dirichlet allocation computation using approximate counters. US Patent 10,147,044.

[95] Harold S. Stone. 1970. A Logic-in-Memory Computer. *IEEE Trans. Comput.* C-19, 1 (1970), 73–78.

[96] Dufy Teguia, Jiaxuan Chen, Stella Bitchebe, Oana Balmau, and Alain Tchana. 2024. vPIM: Processing-in-Memory Virtualization. In *Proceedings of the 25th International Middleware Conference* (Hong Kong, Hong Kong) *(Middleware '24)*. Association for Computing Machinery, New York, NY, USA, 417–430. https://doi.org/10.1145/3652892.3700782

[97] Vijay R Tiwari. 2023. Developments in KD tree and KNN searches. *International Journal of Computer Applications* 975 (2023), 8887.

[98] Thomas Tseng, Laxman Dhulipala, and Guy E. Blelloch. 2019. Batch-parallel Euler Tour trees. In *SIAM Meeting on Algorithm Engineering and Experiments (ALENEX)*. 92–106.

[99] UPMEM. 2025. UPMEM Technology. https://www.upmem.com/technology/. Accessed February 2025.

[100] Leslie G Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (1990), 103–111.

[101] Yiqiu Wang, Yan Gu, and Julian Shun. 2020. Theoretically-efficient and practical parallel DBSCAN. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2555–2571.

[102] Yiqiu Wang, Shangdi Yu, Laxman Dhulipala, Yan Gu, and Julian Shun. 2022. ParGeo: a library for parallel computational geometry. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Seoul, Republic of Korea) *(PPoPP '22)*. Association for Computing Machinery, New York, NY, USA, 450–452. https://doi.org/10.1145/3503221.3508429

[103] Chunxia Xiao and Meng Liu. 2010. Efficient mean-shift clustering using gaussian kd-tree. In *Computer Graphics Forum*, Vol. 29. Wiley Online Library, 2065–2073.

[104] Qingjun Xiao, Zhiying Tang, and Shigang Chen. 2020. Universal online sketch for tracking heavy hitters and estimating moments of data streams. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 974–983.

[105] Jingyi Xu, Sehoon Kim, Borivoje Nikolic, and Yakun Sophia Shao. 2021. Memory-Efficient Hardware Performance Counters with Approximate-Counting Algorithms. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 226–228. https://doi.org/10.1109/ISPASS51385.2021.00041

[106] Andrew Chi-Chih Yao. 1981. Should tables be sorted? *Journal of the ACM (JACM)* 28, 3 (1981), 615–628.

[107] Wei Zeng and Theo Gevers. 2018. 3dcontextnet: Kd tree guided hierarchical learning of point clouds using local and global contextual cues. In *Proceedings of the European Conference on Computer Vision (ECCV) Workshops*. 0–0.

[108] Yiwei Zhao, Jinhui Chen, Sai Qian Zhang, Syed Shakib Sarwar, Kleber Hugo Stangherlin, Jorge Tomas Gomez, Jae-Sun Seo, Barbara De Salvo, Chiao Liu, Phillip B. Gibbons, and Ziyun Li. 2025. H4H: Hybrid Convolution-Transformer Architecture Search for NPU-CIM Heterogeneous Systems for AR/VR Applications. In *Proceedings of the 30th Asia and South Pacific Design Automation Conference* (Tokyo, Japan) *(ASPDAC '25)*. Association for Computing Machinery, New York, NY, USA, 1133–1141. https://doi.org/10.1145/3658617.3697627

[109] Yiwei Zhao, Ziyun Li, Win-San Khwa, Xiaoyu Sun, Sai Qian Zhang, Syed Shakib Sarwar, Kleber Hugo Stangherlin, Yi-Lun Lu, Jorge Tomas Gomez, Jae-Sun Seo, et al. 2024. Neural Architecture Search of Hybrid Models for NPU-CIM Heterogeneous AR/VR Devices. *arXiv preprint arXiv:2410.08326* (2024).

[110] Yue Zhao, Yunhai Wang, Jian Zhang, Chi-Wing Fu, Mingliang Xu, and Dominik Moritz. 2021. KD-Box: Line-segment-based KD-tree for interactive exploration of large-scale time-series data. *IEEE Transactions on Visualization and Computer Graphics* 28, 1 (2021), 890–900.

[111] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. 2008. Real-time KD-tree construction on graphics hardware. *ACM Trans. Graph.* 27, 5, Article 126 (Dec. 2008), 11 pages. https://doi.org/10.1145/1409060.1409079