Parallel R-tree–based Spatial Query Processing on a Commercial Processing-in-Memory System

Tasmia Jannat[1], Michael Gowanlock[2], Satish Puri[1]
[1]Department of Computer Science
Missouri University of Science and Technology, Rolla, MO, USA
[2]School of Informatics, Computing, and Cyber Systems
Northern Arizona University, Flagstaff, AZ, USA
tjv7w@mst.edu, michael.gowanlock@nau.edu, satish.puri@mst.edu

***Abstract*—The growing volume of data in scientific domains has made spatial query processing increasingly challenging due to high data transfer costs across the memory hierarchy and limited memory bandwidth. To address these bottlenecks and reduce the energy consumed on data movement, this work explores Processing-in-Memory (PIM) systems by executing range queries directly inside memory chips. Unlike prior PIM studies centered on linear scans or hash-based queries, this work is the first to map R-tree range queries onto commercial PIM hardware. The proposed broadcast-based method constructs the R-tree bottom-up on the CPU, broadcasts top levels to UPMEM DPUs (DRAM Processing Units) for global filtering, and distributes lower levels for parallel batched queries in a CPU–DPU system. We evaluate our approach on two real spatial datasets Sports (999K rectangles) and Lakes (8.4M rectangles) and assess scalability using a synthetic dataset with up to 16M rectangles and 3.9M queries, on a commercial UPMEM PIM system with up to 2,540 DPUs. Across all datasets, broadcast-based execution consistently outperforms subtree partitioning by preventing communication from dominating execution. On the Lakes dataset, strong scaling from 512 to 2,540 DPUs reduces kernel time from 64.9 s to 17.6 s, yielding up to 3.66× kernel and 2.70× end-to-end speedup relative to the CPU R-tree search on the same system. The PIM kernel also consumes approximately 3.4× less energy than the corresponding CPU search (e.g., 59.6 kJ vs. 167.0 kJ on Lakes), demonstrating scalable and energy-efficient hierarchical spatial range queries.**

***Index Terms*—Range Search, Processing-in-Memory (PIM), High-Performance Computing (HPC), Spatial Query Processing, R-tree Structure**

I. Introduction

HPC systems face significant challenges in handling the exponential increase in data, especially in applications such as Geographic Information Systems (GIS) and spatial databases, which require efficient spatial query processing. Traditional computing architectures struggle because they face the memory wall problem [1], the widening gap between processor speed and memory access, which hinders performance for data intensive tasks. This has led to interest in Processing-in-Memory (PIM) architectures that bring computation closer to the data, reducing bottlenecks, and improving both speed and energy efficiency. This limitation is especially evident in spatial query processing. Many scientific and geospatial workloads issue large numbers of range queries over rectangles and polygons. While R-tree indexing reduces the number of candidate objects examined, the cost of accessing index nodes and data records from memory frequently dominates both performance and energy consumption. PIM offers a direct solution by executing computations near data. Among available platforms, UPMEM DPUs [2], [3] are notable for scale, programmability, and availability in server configurations, which makes them attractive for experiment driven systems work.

Most prior PIM evaluations focus on scans and hash based operators [4], [5], [6]. In contrast, little is known about how a hierarchical spatial index such as the R-tree behaves on real PIM hardware, especially when the index is built on the host CPU and traversed across thousands of near memory cores with explicit host to device transfers and small local memories.

This paper addresses this gap by presenting the first design and implementation of R-tree range query on a commercial PIM platform. We introduce a broadcast-based execution strategy that serializes and broadcasts compact upper-level R-tree headers to all DPUs for global pruning, while partitioning leaf nodes across DPUs for parallel processing. Queries are broadcast in batches, and each DPU applies upper-level filtering using the locally available R-tree headers before scanning its assigned leaf nodes, thereby preserving hierarchical pruning while minimizing communication and redundant data movement.

We evaluate our design on real and synthetic spatial datasets using a commercial UPMEM PIM system with thousands of DPUs. The results show that the proposed broadcast-based execution consistently outperforms subtree-based partitioning by preventing communication from dominating execution. Compared to CPU-based spatial search on the same system, our approach achieves substantial performance improvements and significantly lower energy consumption during the query phase. These findings indicate that hierarchical spatial query processing can be executed efficiently on PIM architectures when index layout and communications are carefully aligned with hardware constraints. We make the following contributions:

1) **A heterogeneous CPU–PIM design for hierarchical spatial query processing.** We develop a CPU–DPU execution pipeline that constructs and serializes an R-tree on the CPU, broadcasts the upper levels to all DPUs,

and distributes the leaf level in contiguous breadth-first partitions across ranks.

2) **First R-tree implementation on commercial Processing-In-Memory architecture.** To our knowledge, this is the first study to map R-tree traversal onto a real PIM platform. The design broadcasts the top-level tree structure to all DPUs and distributes leaf partitions across DPUs to expose parallelism.
3) **An empirical study of performance and energy on real workloads.** We evaluate our design on two real spatial datasets (Sports and Lakes) and a large synthetic dataset with up to 16M rectangles and 3.99M queries, using up to 2,540 DPUs. Our study analyzes kernel-level and end-to-end performance, demonstrating up to 3.6× kernel speedup and 2.7× end-to-end speedup over the CPU search phase on the same system, along with up to 3.4× lower energy consumption for the PIM kernel.

Overall, this work shows that hierarchical spatial query processing can be effectively supported on commercial PIM systems through hardware-aware index organization and communication design. By presenting the first implementation and evaluation of R-tree–based spatial query processing on a real PIM platform, the proposed approach highlights the potential of PIM architectures for accelerating irregular, data-intensive workloads beyond linear scans and hash-based operators.

II. Background and Motivation

A. PIM System Architecture and Model

PIM architectures integrate simple processing elements directly within memory chips, enabling computation to be performed close to data and reducing costly data movement across the memory hierarchy [7]. Figure 1 shows the UPMEM processing in memory system that has been used in our work. Each DIMM contains 16 PIM chips, and each chip integrates 8 DRAM Processing Units (DPUs) running at 400 MHz, resulting in 128 DPUs and 8 GB of DRAM capacity per DIMM. At server scale, a 20-DIMM configuration provides up to 2,560 DPUs and 160 GB of memory, enabling thousands of near-memory cores to operate in parallel.

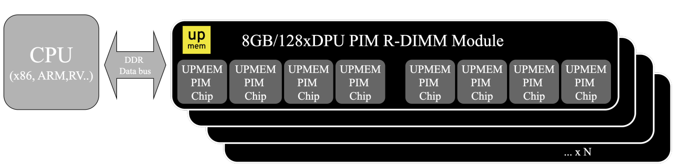

Fig. 1: Processing-in-Memory (PIM) System Organization [8]. DPU stands for DRAM Processing Unit.

Each DPU integrates a lightweight processing core and a hierarchical memory system consisting of MRAM, WRAM, and instruction memory (IRAM). MRAM provides 64 MB of DRAM-based storage per DPU and is optimized for bulk accesses, making it suitable for large data structures such as serialized index nodes and leaf records. WRAM is a small (64 KB), low-latency memory shared among tasklets and is used for control metadata, query buffers, and intermediate results. IRAM stores DPU instructions and is managed by the runtime. DPUs support multiple lightweight hardware threads (tasklets); in the UPMEM PIM v1B platform used in this work, each DPU supports up to 16 tasklets, enabling intra-DPU parallelism under tight WRAM constraints.

DPUs do not support direct inter-DPU communication; all data exchange occurs via the host using explicit bulk transfers. Consequently, applications follow a bulk-synchronous parallel (BSP) execution model [8], which favors communication-efficient designs that exploit local MRAM access and carefully manage limited WRAM.

B. R-tree

Spatial databases and GIS engines answer range-based queries, joins, and overlays on rectangles and polygons, which require indexing methods that can efficiently prune large portions of search space [9]–[13]. Introduced by Guttman [14], the R-tree is a height-balanced spatial index that organizes objects into a hierarchy of minimum bounding rectangles (MBRs). Leaf nodes store object MBRs, while internal nodes store MBRs that bound their children, enabling hierarchical pruning during query evaluation. Although node overlap can lead to multiple traversal paths, R-trees remain widely used in GIS, spatial databases, and scientific workloads due to their flexibility and effective support for spatial range queries. For static datasets, bulk-loading improves R-tree construction and query performance. One of the most widely adopted bulk-loading algorithms is the Sort-Tile-Recursive (STR) method proposed by Leutenegger et al. [15]. STR builds a packed R-tree bottom-up by sorting objects along one dimension, partitioning them into slices, and grouping them into leaf nodes with fixed fanout; higher levels are formed recursively from child MBRs. STR achieves high space utilization and reduced overlap compared to incremental insertion, improving query pruning. In this work, we adopt STR bulk-loading due to its low overlap, high space utilization, and predictable level-wise structure, which are well suited for parallel spatial query processing.

The parallel R-tree data structure has been well explored in the literature for multi-cores [16], GPUs [17]–[21] and distributed memory environments [22]–[25]. However, porting existing designs to the PIM system is not straightforward due to architectural differences between the traditional processor-centric and PIM paradigms.

C. Prior Works: PIM-Friendly Indexes

Most of the existing work on PIM focuses on data-parallel operators with regular access patterns, including table scans [3], [4], selection and aggregation [26], and hash-based joins [5]. Prior studies on commercial PIM platforms, particularly UPMEM, show that such operators benefit from high internal memory bandwidth and massive parallelism across DPUs [3], [5]. Similar trends have also been reported in earlier near-data processing systems targeting analytics workloads [26]–[28].

Indexing support on PIM has received comparatively less attention. Existing PIM-friendly index designs primarily target key–value workloads and rely on simplified data structures such as hashing or skip lists to avoid deep pointer traversal and irregular control flow, as exemplified by PIM-Tree [6]. More recently, Kim et al. proposed OLTPim, an end-to-end OLTP DBMS for commercial UPMEM systems that offloads pointer-chasing components such as indexes and version chains to PIM, using hash-partitioned local B+trees and batched execution to amortize offload overheads [29]. A recent work has also explored space-partitioning indexes for multidimensional points on PIM, such as PIM-zd-tree [30], which leverages space-filling curve ideas to linearize multidimensional data and reduce traversal complexity. In contrast, R-trees perform hierarchical pruning over MBRs (bounding rectangles) and require multi-level traversal with data-dependent branching, which makes them structurally different from the above PIM index designs. Moreover, R-tree range queries are output-sensitive: query performance depends not only on the input size and traversal depth, but also on the number of overlapping objects produced as output, which is not known a priori. It presents additional implementation challenges to handle variable size output. The proposed design handles a variety of output sizes across different queries and since the memory allocation is different in PIM, our implementation takes care of output-sensitive nature of the data structure. To the best of our knowledge, we are not aware of any existing work that implements and evaluates R-tree range-query processing on commercial UPMEM PIM hardware; this work addresses this gap with a broadcast-based execution strategy that keeps host–DPU communication from dominating end-to-end execution.

III. Methodology

This section presents the three execution approaches considered in our study: a multi-threaded CPU implementation (CPU Baseline), a subtree-based PIM design used as the baseline (Baseline PIM R-tree), and the proposed broadcast-based PIM design (Broadcast PIM R-tree).

A. Multi-threaded CPU Baseline

We compare PIM-based execution against a multi-threaded CPU baseline that performs R-tree range queries entirely in host memory. To ensure a fair comparison, the CPU baseline uses the *same R-tree structure* as the proposed broadcast-based PIM approach, constructed on the host with identical bulk-loading parameters. This ensures that performance differences arise from the execution model rather than from index structure.

R-tree construction is performed sequentially, as it is a one-time preprocessing cost, while query processing is parallelized across CPU threads. Each query traverses the R-tree using bounding-box filtering followed by exact rectangle intersection tests, matching the semantics of the PIM kernel.

We implement a custom multi-threaded search engine using POSIX threads, as publicly available R-tree libraries do not provide scalable, thread-parallel query execution with explicit control over scheduling and traversal behavior. Parallel query processing uses dynamic, chunk-based scheduling to mitigate load imbalance caused by spatial skew and query selectivity. The R-tree is read-only during query execution, eliminating synchronization within the traversal. Algorithm 1 outlines the parallel CPU query processing strategy.

Algorithm 1 Parallel CPU R-tree Range Query Processing (CPU Baseline)

Require: R-tree root R, query set $Q[0 : : : N \quad 1]$, threads T, chunk size C
Ensure: Result array $results[0 : : : N \quad 1]$

```
Initialize shared atomic index idx    0
for all threads t = 0 to T    1 in parallel do
    while true do
        start    atomic_fetch_and_add(idx; C)
        if start    N then break
        end if
        end    min(start + C; N)
        for i = start to end    1 do
            results[i]    SEARCHR-TREE(R; Q[i])
        end for
    end while
end for
return results
```

B. Subtree-based Baseline PIM R-tree

As a baseline, we implement a subtree-based PIM R-tree in which each DPU is assigned an independent R-tree subtree and processes queries locally. This design aligns with PIM systems, where DPUs operate on DPU-local MRAM address spaces without inter-DPU communication. We use this baseline to quantify the cost of subtree partitioning and host–DPU data movement.

a) Fanout-Constrained R-tree Construction: To enforce a one-to-one mapping between subtrees and DPUs, we construct the R-tree on the host using a custom top-down bulk-loading strategy that explicitly constrains the root fanout to the number of DPUs. Traditional R-tree constructions are not suitable for this as Guttman's insertion-based R-tree yields data-dependent fanout, while STR bulk loading builds the tree bottom-up without controlling the number of top-level subtrees. Algorithm 2 outlines the construction, which recursively partitions rectangles using x- and y-center ordering to form spatially coherent groups while capping the root fanout. Although not a classical STR construction, this approach adopts STR-style spatial ordering to improve locality, reduce overlap, and balance subtree sizes.

b) Subtree Serialization and Execution: Figure 2 illustrates the baseline workflow. After construction, the R-tree is partitioned at the root, and each level-1 subtree is serialized into a contiguous representation and transferred to its assigned DPU. Each DPU stores its subtree in MRAM and evaluates all queries locally using multiple tasklets.