R-Tree: A Hardware Implementation

Xiang Xiao, Tuo Shi, Pranav Vaidya and Jaehwan John Lee
Department of Electrical and Computer Engineering
Indiana University Purdue University Indianapolis

Abstract—R-tree data structures are widely used in spatial databases to store, manage and manipulate spatial information. As the data volume of such databases is typically very large, the query operation on R-tree data structure has an important impact on the performance of spatial databases. To boost the performance of R-tree search operations, we propose a new parallel R-tree search algorithm, which utilizes an adjacency matrix representation for an R-tree and performs the search using binary arithmetic among matrix elements. When the algorithm is implemented in hardware, we demonstrate through our simulation that the run-time complexity of the new algorithm is only bounded by the height of an R-tree. Furthermore, we find that a program with the proposed algorithm in hardware is 8.27X faster than the same program with its software counterpart in solving an example search problem. In the future, more research will be conducted to adapt the proposed algorithm with divide-conquer paradigm to handle large spatial data sets.

Index Terms: R-tree, Spatial Search, Parallel Algorithm, Hardware Implementation, Database.

I. INTRODUCTION

Modern spatial related applications such as Geographic Information systems (GIS), VLSI design, network applications and multimedia databases have imposed new challenges in spatial query processing. The reason is that non-spatial indexing cannot handle features such as distance of points and whether points fall within certain area of interest or not. Common spatial index approaches include Grid [13], Quadtree [10], UB-tree [14] and R-tree [11]. Among them, the original R-tree, a hierarchical index-structure derived from the B-tree, was first introduced by Gutmann in 1984 [11]. R-tree data structure, a preferred method of spatial data indexing, is one of the keys to address such challenges. The structure was designed to efficiently perform intersection queries of two-dimensional (2D) spatial objects such that a spatial search may require visiting only a small number of nodes. The R-tree offers one of the most promising solutions to store 2D spatial objects in their primitive forms while maintaining versatility of its data structure.

Used to handle indexing (X,Y) coordinates of geographical data, each node of an R-tree corresponds to a Minimum Bounding Rectangle (MBR) (also known as bounding box). Figure 1 shows an example of an R-tree and corresponding bounding boxes. From Figure 1, we see that R-tree data structure splits space with hierarchically nested and possibly overlapped MBRs. Therefore, each non-leaf node in a higher level is formed to enclose multiple child rectangles from a lower level. At the bottom level, each rectangle is formed to enclose a bounding box of data object (e.g., $R_8$ through $R_{15}$). Each node in a non-leaf level is allowed to have a number of children between lower-bound (say $m$) and upper-bound (say $M$) values. For instance, in Figure 1, $M$ is 3 and $m$ is 2. If any node has more than $M$ children, it is called an overflowing node and needs to be split. If any node has fewer than $m$ children, the R-tree needs to be condensed [11]. Typically, $R(m, M)$, $m \leq \lceil M/2 \rceil$ is used to represent a particular R-tree. Note that there can be multiple R-tree solutions for one spatial data set according to R-tree variants as described in Section III. 

![Figure 1](image_url)

Figure 1. An example of R-tree and corresponding bounding boxes.

The search operation on R-tree data structure, one of operations most frequently performed in spatial databases, has a critical impact on the performance of spatial databases. Therefore, we consider R-tree search in our study, investigate how to parallelize a search algorithm and implement it in hardware to gain speedup in search operations. In this paper, we propose a new R-tree search algorithm that uses a matrix representation of an R-tree and conducts the search through binary arithmetic among matrix elements in parallel. We have demonstrated that when the algorithm is implemented in hardware, the run-time complexity of the algorithm is bounded only by the height of an R-tree. Moreover, we compare the execution time (in clock cycles) of the algorithm implemented...
in hardware with its software version and find that for the example search problem discussed in this paper, the search with the hardware version is 18.1X faster than that with the software version when the execution time of application programming interface is included.

The rest of this paper is organized as follows: Section II illustrates the terms used in this paper. Section III reviews some R-tree variants and related hardware implementations. Section IV presents the proposed parallel R-tree search algorithm, and Section V describes its hardware implementation. Section VI discusses the simulation of the proposed algorithm and comparison between the algorithm in hardware and its software version. Section VII concludes this paper.

II. TERMS AND DEFINITIONS

In this section we use Figure 1 to describe the terms used in this paper.

1) Minimum Bounding Rectangle (MBR): An MBR, also known as a bounding box, is a rectangle that encloses an R-tree node and its children nodes if it is a non-leaf node with minimum area. For example, \( R_1 \) through \( R_{19} \) in Figure 1(a) are MBRs.

2) Node: Each node corresponds to an MBR. For example, \( R_1 \) through \( R_{19} \) in Figure 1(b) are R-tree nodes. There are three types of R-tree nodes. In Figure 1(b), \( R_1 \) and \( R_2 \) are root nodes (i.e., nodes in the root); \( R_3 \) through \( R_7 \) are intermediate nodes; and \( R_8 \) through \( R_{19} \) are leaf nodes.

3) Intermediate Node: An intermediate node is any node (except root) of an R-tree that has children nodes and thus is not a leaf node.

4) Data object: Data objects are spatial data elements and contained in leaf nodes of an R-tree (e.g., \( R_8 \) through \( R_{19} \)).

5) Height of an R-tree: The height of an R-tree is the number of hierarchical levels in an R-tree. For example, the height of the R-tree shown in Figure 1 is 3.

6) Full node: A full node is a node with the maximum number (M) of children nodes.

III. RELATED WORK

There are many R-tree variants, which are built in either static or dynamic manner. Static methods re-build the entire R-tree when a new data object is added. In contrast, dynamic methods support adding and removing nodes on the fly. When a new data object is added to an existing R-tree, dynamic methods use two major techniques to split overflowing nodes during insertion. The primary goal of the first technique is to minimize both coverage of the sum of the areas of bounding boxes and overlap with sibling nodes. An example of node split is given in Figure 2. Figure 2(a) shows that a new node is inserted in a full node with the maximum number (M=8) of children nodes. Figure 2(b) shows a split solution that minimizes coverage of the sum of the areas of bounding boxes A and B. Figure 2(c) shows a split solution that minimizes overlap between bounding boxes A and B. As can be seen from Figure 2(b) and Figure 2(c), it is typical that the goals to minimize coverage and to minimize overlap are contradictory. Thus, many heuristics for node split have been developed. The second technique is the extension of \( B^+ \)-tree to d-dimension (\( d \geq 2 \)) and makes use of the centroids of bounding boxes to order data objects. In the following, we summarize some of prominent R-tree variants proposed in the past:

1) \( R^+ \)-tree: \( R^+ \)-tree is a compromise between R-tree and K-D-B tree [15]. K-D-B tree is a data structure that splits multidimensional spaces like an adaptive K-D tree [5], but balances the resulting tree like a B-tree. \( R^+ \)-tree avoids overlapping of intermediate nodes by inserting an object into multiple leaves if necessary. \( R^+ \)-tree differs from R-tree in that: (i) nodes are not guaranteed to be at least half filled; (ii) the entries of any intermediate node do not overlap; and (iii) an object ID may be stored in more than one intermediate node. Because nodes are not overlapped with each other, query performance benefits by following a single path and thus visiting fewer nodes than R-tree. Since rectangles are duplicated, the size of an \( R^+ \)-tree can be larger than that of an R-tree built on the same data set. Construction and update operations of \( R^+ \)-tree are more complex than those of R-tree and other R-tree variants.

2) R*-tree: Beckmann et al. [4] introduced the R*-tree, which has more complex criteria for the distribution of bounding boxes. However, it is more efficient in insertion and space utilization than R-tree. To minimize both overlap and coverage, two criteria are typically used: (i) the overlap between the MBRs of intermediate nodes is minimized, and (ii) the perimeter of each MBR of intermediate nodes is minimized. They also propose...
a force-reinsert technique to reduce the possibility of splitting a full node. When a node overflows, instead of immediately splitting, first attempt is made to see if some of the objects in the node could possibly be more suited to be in another node. For all $M+1$ entries in an overflowing node, distances between the centroids of their MBRs and the centroid of the MBR of the node are calculated. Then, the entries are sorted in descending order of the distances. The first $p$ entries in the sorted list are removed from the current node and reinserted in the tree. A node is only split if it is found to overflow after reinsertion takes place. Therefore, such redistributions may prevent more expensive split operations, and also storage utilization is improved.

3) Hilbert Method: Kamel et al. [12] proposed an improved R-tree, called Hilbert R-tree, which uses a Hilbert space filling curve for grouping data objects and orders objects on the basis of the Peano-Hilbert numbers (e.g., [16]) corresponding to the centroids of MBRs. Every node has a well-defined set of sibling nodes, and thus, deferred splitting can be used. Objects are stored in a $B^+\text{-tree}$. The Hilbert space filling curve has been shown to give a degree of space utilization as high as is desired.

4) Priority R-tree (PR-tree): Present by [1], PR-tree is the first R-tree variant that answers a window query (i.e., query box) by accessing $O((N/B)^{1-1/d} + T/B)$ blocks, where $N$ is the number of $d$-dimensional objects stored, $B$ is the number of objects per block, and $T$ is the number of objects whose bounding boxes intersect the query window. Moreover, PR-tree is a linear space structure and asymptotically optimal in terms of query performance. Experiments show that the PR-tree performs similar to the best known R-tree variants on real-life and relatively nicely distributed data, but outperforms them significantly on more extreme data [1].

In terms of hardware implementations of R-tree, its variants, spatial selections and/or joins, several have been proposed in the past. Sun et al. presented hardware acceleration for spatial selections and joins using efficient rendering and searching capabilities of modern graphics hardware [3], [17] and reported that their techniques can provide up to 5.9X speedup over software only systems. Bandi et al. introduced another hardware acceleration method utilizing content addressable memories for database systems and showed such method can provide up to two orders of magnitude speedup against a software implementation [2]. Bogdan et al. [6] applied such hardware acceleration to bioinformatics area for processing of mass spectrometric data for proteomics using an FPGA and also showed a similar speedup to [2]. We also believe that there exists high potential in devising hardware acceleration approaches for various database algorithms, which is one of our research directions. However, our direction is somewhat different than others in that we exploit hardware-oriented parallel algorithmic approaches derived from existing promising software algorithms and thus focus on reducing runtime complexity. To the best of our knowledge, we have not found such similar work to ours. Thus, the proposed work in this paper is the first attempt to this direction.

IV. A NEW PARALLEL R-TREE SEARCH ALGORITHM

In this section, we first introduce our R-tree data structure and then describe the details of our parallel R-tree search algorithm. Lastly, we explain how the search algorithm operates on the given example shown in Figure 1.

A. Data structure used in our R-tree search algorithm

Adjacency matrices are widely used to represent graphs and trees. We also use an adjacency matrix to represent the structure of an R-tree. Assuming that an R-tree contains $n$ nodes (say $R_1, R_2, \ldots, R_n$), our adjacency matrix representation of the R-tree is defined as follows:

$$R_{tree}[i][j]_{n \times n} = \begin{cases} 1 & \text{if } R_i \text{ is the parent node of } R_j, \\ 0 & \text{otherwise,} \end{cases}$$

where $n$ is the number of nodes in the R-tree. Table 1 shows the adjacency matrix representing the R-tree shown in Figure 1(b).

<table>
<thead>
<tr>
<th></th>
<th>$R_1$</th>
<th>$R_2$</th>
<th>$R_3$</th>
<th>$R_4$</th>
<th>$R_5$</th>
<th>$R_6$</th>
<th>$R_7$</th>
<th>$R_8$</th>
<th>$R_9$</th>
<th>$R_{10}$</th>
<th>$R_{11}$</th>
<th>$R_{12}$</th>
<th>$R_{13}$</th>
<th>$R_{14}$</th>
<th>$R_{15}$</th>
<th>$R_{16}$</th>
<th>$R_{17}$</th>
<th>$R_{18}$</th>
<th>$R_{19}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_1$</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$R_2$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$R_3$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$R_4$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$R_5$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$R_6$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$R_7$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$R_8$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$R_9$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$R_{10}$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$R_{11}$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$R_{12}$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$R_{13}$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$R_{14}$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$R_{15}$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$R_{16}$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$R_{17}$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$R_{18}$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$R_{19}$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 1. An adjacency matrix representing the R-tree in Figure 1(b).
**Algorithm 1: Parallel R-tree Search Algorithm**

```
OVERLAP([box-a-coor[1:4]], box-b-coor[1:4]) // Decide whether or not box A is overlapped with box B
2 then overlap-x ← 0
3 else overlap-x ← 1 // Two boxes are overlapped along X-coordinate.
5 then overlap-y ← 0
6 else overlap-y ← 1 // Two boxes are overlapped along Y-coordinate.
7 return overlap ← overlap-x ∧ overlap-y // Two boxes are overlapped if they are overlapped along both X and Y coordinates.

R-TREE-SEARCH(tree-root[1:n], query-box-coor[1:4])
8 for all i where i ← 1 to n // in parallel for all nodes, resulting in O(1) computation time
9 do global-overlap-results[i] ← OVERLAP(Coor[i]), query-box-coor[1:4]) // Find all nodes with their bounding
10 // boxes overlapped with the query box.
11 current-search-nodes[] ← tree-root[]
12 while current-search-nodes[] contains at least one non-zero element
13 do current-overlap-nodes[] ← current-search-nodes[] ∧ global-overlap-results[]
14 // Determine whether or not any of the nodes being visited is overlapped with the query box.
15 for all i where i ← 1 to n // in parallel for all nodes, resulting in O(1) computation time
16 do if (current-overlap-nodes[i] = 1) and (Rtree[i]) contains all zeros // R_i has a leaf node with its bounding
17 then Record R_i as one of the qualifying leaf nodes // parent node overlaps with the query box.
18 for all j where j ← 1 to n // in parallel for all nodes, resulting in O(1) computation time
19 do current-search-nodes[j] = (Rtree[i]) ∧ current-overlap-nodes[i]) // Find all children nodes whose
20 // parent node overlaps with the query box in the current iteration.
21 return
```

* ∧ and ∨ denote bitwise AND and OR, respectively.

In addition to the structures just introduced, the coordinates of bounding boxes also need to be stored in order to find MBRs overlapped with a query box. Using (X,Y) coordinates, each bounding box, as a rectangle, can be uniquely represented by the coordinates of its corner points as shown in Figure 3.

![Figure 3. An example of two overlapped bounding boxes.](image)

We use a vector consisting of four elements in the format of [x1 y1 x2 y2] to represent a bounding box, where x1 < x2 and y1 < y2. Hence, the example shown in Figure 1(a) needs a matrix of dimension 19 × 4 to store the coordinates of all bounding boxes. Thus, coordinates matrix Coor[] is defined as follows:

\[
Coor[i][j]_{n×4} = \begin{cases} 
    \text{coordinate } x1 \text{ of the bounding box of } R_i, & \text{for } j = 1, \\
    \text{coordinate } x2 \text{ of the bounding box of } R_i, & \text{for } j = 2, \\
    \text{coordinate } y1 \text{ of the bounding box of } R_i, & \text{for } j = 3, \\
    \text{coordinate } y2 \text{ of the bounding box of } R_i, & \text{for } j = 4, 
\end{cases}
\]

where n is the number of nodes in an R-tree.

After we have the coordinates of all the MBRs, we can determine whether or not any bounding box is overlapped with a query box. For any two overlapped bounding boxes, they are both overlapped along X-coordinate as well as Y-coordinate as shown in Figure 3. More detail of how we find the overlapped nodes is revealed when we discuss the R-tree search algorithm in the next subsection.

**B. The details of the algorithm**

Now we are ready to introduce a new parallel search algorithm for an R-tree. The purpose of the search algorithm is to find all leaf nodes whose bounding boxes overlap a search rectangle (i.e., a query box). For example, a query box (denoted by R_q) that is overlapped with a few bounding boxes is shown in Figure 1(a). Given the query box, the search starts from the root node. For every non-leaf node, whether or not it overlaps with the query box has to be decided. If yes, all children nodes of such a non-leaf node have to be searched iteratively. When a leaf node is reached, its bounding box (i.e., rectangle) is tested against the query box, and fetched into the result set if it intersects the query box. Searching is traditionally done in a recursive manner until all nodes have been traversed.

Before we present the pseudo code of the search algorithm, let us first introduce some notations. In this article, matrix[], matrix[i][] and matrix[][j] refer to as “all elements in the matrix,” “all elements of row i in the matrix,” and “all elements of column j in the matrix,” respectively. Next, the pseudo code of the parallel R-tree search algorithm is presented in Algorithm 1. Explanation of the algorithm in detail is given in the next subsection.

**C. Explanation of the algorithm with a simple example**

To explain how the search algorithm works, we use the bounding boxes and their corresponding R-tree in Figure 1. We
show how the proposed algorithm finds all leaf nodes whose bounding boxes are overlapped with the query box (i.e., $R_q$ in Figure 1(a)). We assume that the coordinates of $R_1$ to $R_{19}$ and query box $R_q$ are set as shown in Table 2.

Given two nodes $R_3$ and $R_2$ in the root and the coordinates of query box $R_q$, the algorithm starts searching from the tree root. Before the search starts, it is checked for the bounding box of each node ($R_1$ through $R_{19}$) whether or not it is overlapped with query box $R_q$ by calling sub-function OVERLAP (lines 8 to 9). Taking $R_1$ as an example, to check whether or not $R_1$’s bounding box is overlapped with $R_q$, we compare $R_1$’s coordinates (i.e., [9 43 0 37]) with $R_q$’s (i.e., [19 24 24 33]). We can see from the fact that $x_1(R_1) < x_1(R_q) < x_2(R_q) < x_2(R_1)$ (i.e., $9 < 19 < 24 < 43$) and $y_1(R_1) < y_1(R_q) < y_2(R_q) < y_2(R_1)$ (i.e., $0 < 24 < 33 < 37$), $R_q$ is enclosed by $R_1$. Thus, global-overlap-results[1] is set to 1. On the other hand, taking $R_{19}$ for instance, the coordinates of its bounding box are [48 53 35 39]. Hence, we have $x_1(R_{19}) < x_2(R_{19}) < x_1(R_1) < x_2(R_1)$ (i.e., $19 < 24 < 48 < 53$) and $y_1(R_{19}) < y_2(R_{19}) < y_1(R_1) < y_2(R_1)$ (i.e., $24 < 33 < 35 < 39$), which means that two boxes are not overlapped along either X-coordinate or Y-coordinate. As a result, we conclude that the bounding box of $R_{19}$ is not overlapped with query box $R_q$, and thus global-overlap-results[19] is set to 0. The other elements in global-overlap-results[] can be computed in the same way. Since there is no dependency among the computation of elements in global-overlap-results[], we perform all the computation in parallel in hardware, resulting in O(1) run-time. The values of the elements in global-overlap-results[] are listed in Table 3. This means that the bounding boxes of $R_1$, $R_2$, $R_4$, $R_6$, $R_{12}$ and $R_{16}$ are overlapped with query box $R_q$.

Table 3. The elements in vector global-overlap-results[].

<table>
<thead>
<tr>
<th>$R_1$</th>
<th>$R_2$</th>
<th>$R_3$</th>
<th>$R_4$</th>
<th>$R_5$</th>
<th>$R_6$</th>
<th>$R_7$</th>
<th>$R_8$</th>
<th>$R_9$</th>
<th>$R_{10}$</th>
<th>$R_{11}$</th>
<th>$R_{12}$</th>
<th>$R_{13}$</th>
<th>$R_{14}$</th>
<th>$R_{15}$</th>
<th>$R_{16}$</th>
<th>$R_{17}$</th>
<th>$R_{18}$</th>
<th>$R_{19}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

After vector global-overlap-results[] is computed, the R-tree search starts from the root. One of the inputs to R-TREE-SEARCH, vector tree-root[1:19] is shown in Table 4, which signifies that 1st and 2nd elements are the nodes in the root of the R-tree. Thus, two nodes $R_1$ and $R_2$ are visited (line 10). Next, the search is carried out in the while loop (lines 11 through 17). In the rest of this sub-section, we go through each iteration of the while loop until the loop terminates.

Table 4. The elements in vector tree-root[].

<table>
<thead>
<tr>
<th>$R_1$</th>
<th>$R_2$</th>
<th>$R_3$</th>
<th>$R_4$</th>
<th>$R_5$</th>
<th>$R_6$</th>
<th>$R_7$</th>
<th>$R_8$</th>
<th>$R_9$</th>
<th>$R_{10}$</th>
<th>$R_{11}$</th>
<th>$R_{12}$</th>
<th>$R_{13}$</th>
<th>$R_{14}$</th>
<th>$R_{15}$</th>
<th>$R_{16}$</th>
<th>$R_{17}$</th>
<th>$R_{18}$</th>
<th>$R_{19}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>43</td>
<td>23</td>
<td>37</td>
<td>31</td>
<td>20</td>
<td>17</td>
<td>14</td>
<td>11</td>
<td>25</td>
<td>18</td>
<td>38</td>
<td>35</td>
<td>4</td>
<td>11</td>
<td>45</td>
<td>47</td>
<td>48</td>
<td>19</td>
</tr>
</tbody>
</table>

At the beginning of the first iteration, current-search-node[] is [1100000000000000000] (i.e., equal to vector tree-root[]).

Thus, the loop condition in line 11 is true, and the algorithm moves on to line 12. In line 12, we compute which nodes currently being visited are overlapped with the query box, and store the result in current-overlap-nodes[]. current-overlap-nodes[] is calculated as follows:

\[
current-overlap-nodes[] = current-search-nodes[] \land global-overlap-results[]
\]

where $\land$ denotes bitwise AND.

From the result, we know that the bounding boxes of both $R_1$ and $R_2$ are overlapped with the query box. Then, in lines 13 to 15, we check if overlapped nodes $R_1$ and $R_2$ are leaf nodes by examining if $Rtree[1][]$ and $Rtree[2][]$ contain all zeros. If $Rtree[1][]$ (Rtree[2][]) contains all zeros, $R_1$ (R_2) has no child nodes. That is, $R_1$ (R_2) is a leaf node. Since neither is, we have not found any qualifying leaf node yet. Next, in lines 16 and 17, we find all children nodes of both $R_1$ and $R_2$, and these children nodes are searched in the next iteration. Thus, for each node in the R-tree, we now need to decide whether or not it is a child node of either $R_1$ or $R_2$. For the purpose of illustration, we show how to check whether or not $R_3$ is a child node of either $R_1$ or $R_2$, which is done by computing current-search-nodes[3] as follows:

\[
current-search-nodes[3] = (Rtree[1][1] \land current-overlap-nodes[1]) \lor (Rtree[2][2] \land current-overlap-nodes[2]) \ldots \lor (Rtree[19][19] \land current-overlap-nodes[19])
\]

\[
= (\{1 \land 1\} \lor \{0 \land 1\} \lor \ldots \lor \{0 \land 0\})
\]

\[
= 1
\]

where $\lor$ denotes bitwise OR. As seen from the above equation, current-search-nodes[3] is 1, meaning that $R_3$ is a child node of either $R_1$ or $R_2$. At the end of the first iteration, we have updated vector current-search-nodes[]. The new values in vector current-search-nodes[] are shown in Table 5. These new values indicate that $R_3$ through $R_7$ are children nodes of either $R_1$ or $R_2$. Hence, they are visited in the second iteration.

Table 5. Values of vector current-search-nodes[] after the first iteration.

<table>
<thead>
<tr>
<th>$R_1$</th>
<th>$R_2$</th>
<th>$R_3$</th>
<th>$R_4$</th>
<th>$R_5$</th>
<th>$R_6$</th>
<th>$R_7$</th>
<th>$R_8$</th>
<th>$R_9$</th>
<th>$R_{10}$</th>
<th>$R_{11}$</th>
<th>$R_{12}$</th>
<th>$R_{13}$</th>
<th>$R_{14}$</th>
<th>$R_{15}$</th>
<th>$R_{16}$</th>
<th>$R_{17}$</th>
<th>$R_{18}$</th>
<th>$R_{19}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

At the beginning of the second iteration, the loop condition remains true since vector current-search-nodes[] contains five 1s. Similar computations as in the first iteration are performed in the second iteration. In line 12, we calculate vector current-overlap-nodes[] as follows:

\[
current-overlap-nodes[] = current-search-nodes[] \land global-overlap-results[]
\]

\[
= [00000000000000000] \land [11010100000100010000]
\]

\[
= [00000000000000000]
\]
From the new values of vector \( \text{current-overlap-nodes}[] \), we know that bounding boxes \( R_4 \) (a child of \( R_1 \)) and \( R_6 \) (a child of \( R_2 \)) are overlapped with query box \( R_q \). However, neither of them is a leaf node. Thus, the children nodes of both \( R_4 \) and \( R_6 \) are calculated and stored in vector \( \text{current-search-nodes}[] \). The new values of vector \( \text{current-search-nodes}[] \) at the end of the second iteration are shown in Table 6. These values indicate that \( R_{11}, R_{12}, R_{15} \) and \( R_{16} \) are children nodes of either \( R_4 \) or \( R_6 \) and are going to be visited in the third iteration.

Table 6. Values of vector \( \text{current-search-nodes}[] \) after the second iteration.

<table>
<thead>
<tr>
<th>( i )</th>
<th>( 0 )</th>
<th>( 1 )</th>
<th>( 2 )</th>
<th>( 3 )</th>
<th>( 4 )</th>
<th>( 5 )</th>
<th>( 6 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{current-search-nodes}(i) )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
</tr>
</tbody>
</table>

Now the \textbf{while} loop continues with its third iteration because vector \( \text{current-search-nodes}[] \) consists of four 1s. Then in line 12, vector \( \text{current-overlap-nodes}[] \) is calculated as follows:

\[
\text{current-overlap-nodes}[] = \text{current-search-nodes}[] \land \text{global-overlap-results}[]
\]

From the result, we know that bounding boxes \( R_{12} \) and \( R_{16} \) are overlapped with query box \( R_q \). Since both \( R_{12} \) and \( R_{16} \) are leaf nodes, they are fetched into the result set. Next in lines 16 to 17, new values of vector \( \text{current-search-nodes}[] \) are calculated. Because both \( R_{12} \) and \( R_{16} \) are leaf nodes, neither of them has any child node. Thus, the values of vector \( \text{current-search-nodes}[] \) become all zeros at the end of the third iteration as shown in Table 7.

Table 7. Values of vector \( \text{current-search-nodes}[] \) after the third iteration.

<table>
<thead>
<tr>
<th>( i )</th>
<th>( 0 )</th>
<th>( 1 )</th>
<th>( 2 )</th>
<th>( 3 )</th>
<th>( 4 )</th>
<th>( 5 )</th>
<th>( 6 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{current-search-nodes}(i) )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
</tr>
</tbody>
</table>

Since vector \( \text{current-search-nodes}[] \) now contains all zeros, the \textbf{while} loop terminates. This indicates that all nodes whose bounding boxes are overlapped with the query box have been visited and all qualified leaf nodes have been found. Therefore, the R-tree search algorithm terminates for this example. The result of this search indicates that the bounding boxes of two leaf nodes, \( R_{12} \) and \( R_{16} \), are overlapped with query box \( R_q \). As also can be seen, the number of iterations that the \textbf{while} loop has executed is equal to the height of the R-tree (in our example, the R-tree has three layers) because the algorithm searches one entire tree layer per iteration.

V. HARDWARE IMPLEMENTATION

A diagram of hardware implementation of the proposed parallel R-tree search algorithm is shown in Figure 4. On the left side of Figure 4, the “Coordinates” block represents dozens of registers that store the values of coordinates (i.e., matrix \( \text{Coor}[] \)) of the R-tree bounding boxes and the query box. On the right side, the “R-tree” block represents the registers that store the values of adjacency matrix \( \text{Rtree} \). Each “RT” cell in the figure contains an element in matrix \( \text{Rtree} \). Elements in vectors \( \text{global-overlap-results}[] \), \( \text{current-overlap-nodes}[] \) and \( \text{current-search-nodes}[] \) are stored in the cells denoted with “global overlap results,” “current overlap nodes” and “current search nodes,” respectively. In addition, there is a finite state machine (not shown) that sequences the steps inside the \textbf{while} loop of Algorithm 1 (lines 12 through 17) and terminates the \textbf{while} loop when it reaches the leaf level.

We implement the described R-Tree-Search hardware unit in Verilog HDL and use the Mentor Graphics ASIC Design Kit 3.0 [7] with the TSMC .25 \( \mu \)m standard cell library [19] to synthesize the unit. The area of a synthesized R-Tree-Search unit is 4919 units, each of which is equivalent to a minimum-sized two-input NAND gate in the library. The simulation of such hardware unit performing the search example described in Section IV-C is discussed in the next section.

VI. EXPERIMENTS AND DISCUSSION

A. Simulation of R-tree hardware implementation

To validate the correctness of the proposed parallel R-tree search algorithm, we simulate our hardware implementation using ModelSim v6.2 [9] and performs the search example described in Section IV-C. Through the simulation, we demonstrate that our implementation finishes the search process in three iterations (total three clock cycles), which is determined by the number of iterations that the \textbf{while} loop (lines 11 through 17) executes. It is because the search along different tree branches is performed in parallel. Thus, all nodes with overlapped bounding boxes in the same tree layer are searched simultaneously in hardware in one iteration. Since the R-tree discussed in our example has three layers, the search algorithm finishes in three iterations, each in one clock cycle.

B. Comparison between hardware and software R-tree search algorithms

To investigate the benefit of implementing an R-tree search algorithm in hardware, we compare the number of clock cycles that the proposed algorithm in hardware needs with that of its corresponding software implementation in solving the same search problem. The pseudo code of the R-tree search algorithm in its software version is presented as follows. Note that the maximum number of sub-nodes in a node in Figure 1 is 3 (i.e., \( M=3 \)).

\[
\text{SF-RTREE-SEARCH}(\text{node}, \text{querybox})
\]

1. if \( \text{node} \) is overlapped with \( \text{querybox} \)
2. then
if node has no children
    then include this node into the result set
else
    SF-RTREE-SEARCH(node→left-child, querybox)
    SF-RTREE-SEARCH(node→mid-child, querybox)
    SF-RTREE-SEARCH(node→right-child, querybox)
return

To ensure a fair comparison, we construct a simulated computer system using Mentor Graphics Seamless [8], which can run both the hardware and software versions of the R-tree search algorithm. The system contains one Motorola MPC755 processor and SRAM memory of 256MB. To measure the performance of the search in hardware, we integrate the hardware unit described in Section V within the simulated system so that the hardware unit can be controlled through a program. The program is written in C language and able to write values into “Coordinates” and “R-tree” registers shown in Figure 4 (to fill in the coordinates and construct tree structures) and trigger the search algorithm in hardware. The recursive software implementation of the R-tree search algorithm follows the pseudo code of SF-RTREE-SEARCH, and is also written in C language. Both programs are compiled using a PowerPC-GCC cross-compiler. We use Atalanta Real-Time Operating System (RTOS) version 0.3 [18] to execute the programs on the processor.

We execute both programs to solve the search problem described in Section IV-C and count the number of clock cycles that each program spends on searching and related operations. Specifically, both programs perform three operations: loading MBR coordinates, building R-tree adjacency matrix and conducting the search algorithm. The hardware search takes 494, 342 and 30 clock cycles for the three operations, respectively. In contrast, the software search takes 5510, 1972 and 542 cycles, respectively. The results show that for problem sizes that can be easily accommodated by the hardware, performance gain of the hardware search algorithm as compared to software version is quite acceptable. Comparing solely the time of search, the hardware version is 18.1X faster than the software version. However, while software can be conveniently modified to deal with large data sets, it is more advisable to handle large data sets in hardware using the divide-conquer approach instead of stretching hardware capacity. Thus, there are costs to divide a large problem into smaller sub-problems, manage the sub-problems and merge the results from sub-problems into a complete solution.

We have also considered the other two optimization opportunities: (i) the rows in the adjacency matrix of an R-tree whose elements are all zeros are not used during the search and thus the “zero” lower block can be removed, and (ii) the OVERLAP procedure of Algorithm 1 needs to be optimized to reduce area overhead. In the future, more research will be conducted to study on how to properly handle problems whose sizes exceed hardware capacity to maintain good overall performance of the search algorithm.

VII. CONCLUSION

R-tree is crucial data structure used to manage information in spatial databases. The efficiency on R-tree operations is of vital importance to the performance of spatial databases. In this paper, we propose a parallel R-tree search algorithm in hardware. When the size of the search problem fits in the capacity of the hardware, the run-time complexity of the proposed algorithm is bounded by the height of an R-tree. In addition, our simulation shows that 8.27X performance gain can be achieved in a program with the proposed search algorithm in hardware against that with a software version. Furthermore, to adapt the proposed algorithm to solve large problems, we also design the algorithm in such a way that it is ready to work under the divide-conquer approach. In the future, more research will focus on how to reduce area overhead and integrate the proposed algorithm with the divide-conquer approach while maintaining good performance.

VIII. ACKNOWLEDGMENT

We would like to thank Mentor Graphics for providing ModelSim and Seamless simulation software, Georgia Tech for providing Atalanta RTOS and Sun Microsystems for providing Solaris Unix machines.

REFERENCES