R-Tree: A Hardware Implementation

Xiang Xiao, Tuo Shi, Pranav Vaidya and Jaehwan John Lee

ECE Department, Purdue School of Engineering and Technology

Indiana University-Purdue University Indianapolis

Abstract

R-tree data structures are widely used in spatial databases to store, manage and manipulate spatial information. Because 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 the proposed algorithm in hardware is 30 times faster than 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.

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 the distance of points and whether points fall within certain area of interest or not. Common spatial index approaches include Grid [10], Quadtree [7], UB-tree [11] and R-tree [8]. Among them, the original R-tree, a hierarchical index-structure derived from the B-tree, was first introduced by Gutmann in 1984 [8]. 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.

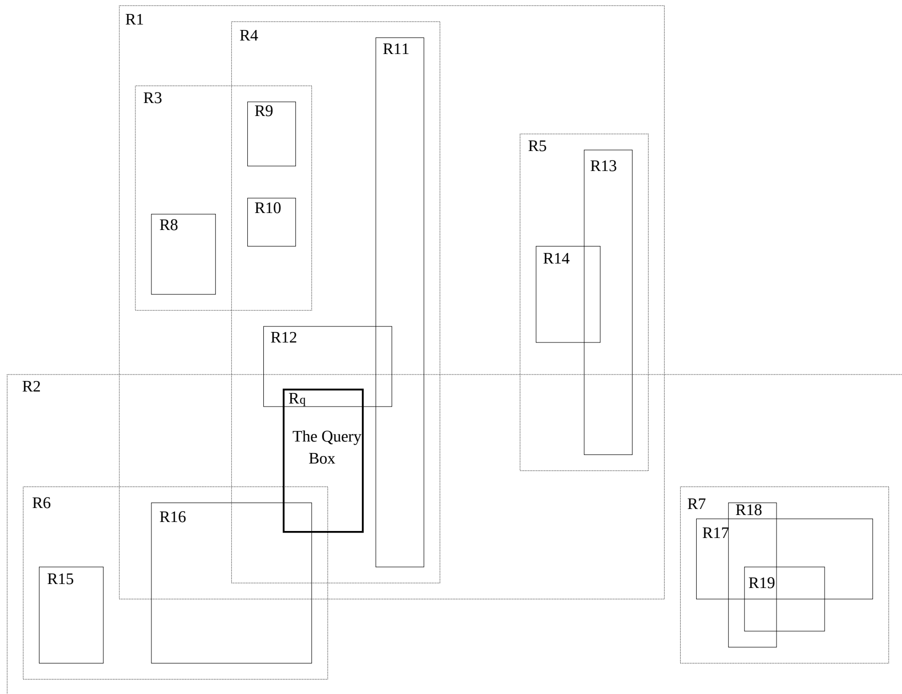
Used to handle indexing (X,Y) coordinates of geographical data, each node of an R-tree corresponds to a Minimum Bounding Rectangle (MBR) (also called bounding box). Figure 1 shows an example of an R-tree and corresponding bounding boxes. From Figure 1, we see that R-tree data structure splits space with hierarchically nested and possibly overlapping 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 (leaf-level), each rectangle is formed to enclose a bounding box of data object (e.g., R_8 through R_{19}). 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 [8]. Typically, $R(m, M),\ m \leq \lceil M/2 \rceil$ is used to represent a particular R-tree. The root node in an R-tree has at least two entries unless it is the only node. Notice that there can be multiple R-tree solutions for one spatial data set according to R-tree variants as described in Section III.

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 report, 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. We have demonstrated that when the algorithm is implemented in hardware, its run-time complexity 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 report, the hardware version is almost 30 times faster than the software version.

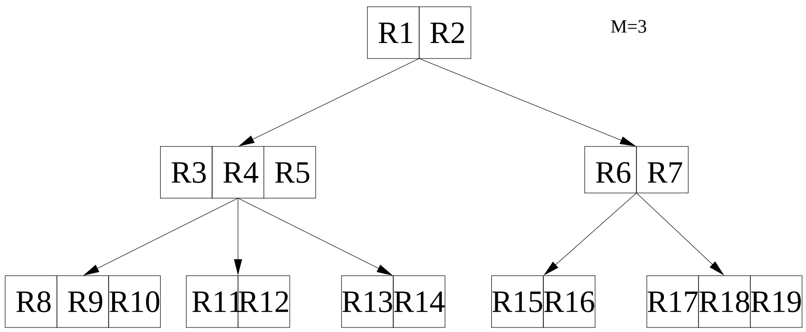
The rest of this report is organized as follows: Section II illustrates the terms used in this paper. Section III reviews some R-tree variants and their respective operations. Section IV presents the proposed parallel R-tree search algorithm. Section V describes the hardware implementation of the proposed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

II. Terms and Definitions

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



(a) Bounding boxes.



(b) R-tree.

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

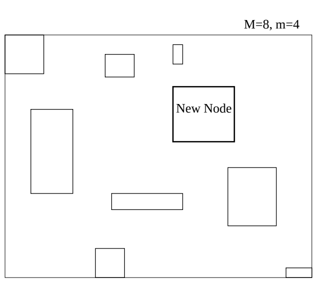
1) Minimum Bounding Rectangle (MBR): An MBR, also known as a bounding box, is a rectangle that encloses an R-tree node and its children nodes if it is a non-leaf node with minimum area. For example, R_1 through R_{19} in Figure 1(a) are MBRs.
2) Node: Each node corresponds to an MBR. For example, R_1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

is not a leaf node.

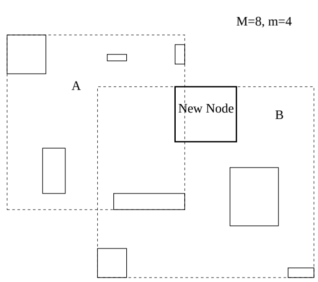
4) Data object: Data objects are spatial data elements and contained in leaf nodes of an R-tree (e.g., R_8 through R_{19}).
5) Height of an R-tree: The height of an R-tree is the number of hierarchical levels in an R-tree. For example, the height of the R-tree shown in Figure 1 is 3.
6) Full node: A full node is a node with the maximum number (M) of children nodes.

III. Related Work

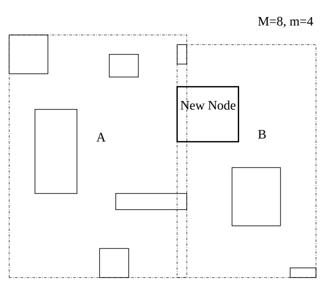
There are many R-tree variants, which are built in either static or dynamic manner. Static methods rebuild the entire R-tree when a new data object is added. In contrast, dynamic methods support adding and removing nodes on the fly. When a new data object is added to an existing R-tree, dynamic methods 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 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 such update approaches as well as the original R-tree algorithm having been proposed in the past:



(a) Insert a new sub-node in a full node.



(b) One solution to minimize coverage.



(c) One solution to minimize overlap.

Figure 2. An example of node split to minimize coverage and/or overlap.

1) Exhaustive search: When a node of an R-tree overflows, the exhaustive search tries all possible partitionings and selects the one that minimizes the sum of areas of two nodes split from an overflowing node [8]. Thus, the complexity of the algorithm is an order of 2^{M-1} where M stands for the maximum number of entries in a node, which makes it impractical for large data volume.
2) Quadratic approach: The quadratic approach, when splitting an overflowing node into two new nodes, chooses two sub-nodes which would waste the most area if they were put in the same node, to form the first entry of each node split. An example of quadratic approach is given in Figure 3, where sub-nodes 1 and 2 are chosen to form the first entries of two new nodes, A and B (bolded rectangles). Sub-nodes 1 and 2 would waste the most area if they were put in the same group (the rectangle drawn in dashed line in Figure 3). This is computed by subtracting the sum of the areas of the two bounding boxes from the area of the dashed rectangle covering sub-nodes 1 and 2 (i.e., $area(\text{dashed rectangle}) - area(1) - area(2)$). Then, the algorithm calculates the enlargement of the MBR of each new node needed to accommodate the rest of bounding boxes. This process is recursively applied to the remaining bounding boxes [8]. This method takes quadratic time $O(M^2)$ with respect to the maximum number of entries in a node.

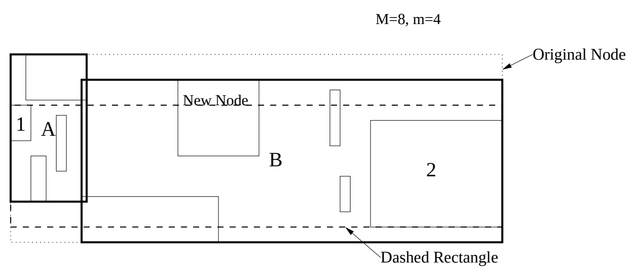


Figure 3. An example of quadratic approach.

3) Linear approach: Along each dimension (i.e., X or Y coordinate), linear split first finds two bounding boxes with the maximum normalized separation and then split along this axis. An example of linear approach is given in Figure 4, where normalized separation along X coordinate ($x1/x0$) is larger than normalized separation along Y coordinate ($y1/y0$). Therefore, the node is split along X-coordinate and the remaining bounding boxes in the node are assigned to the splitting nodes with the least enlargement [8]. This technique takes linear time $O(M)$ with respect to the maximum number of entries in a node.
4) R^+ tree: R^+ tree is a compromise between R-tree and K-D-B tree [12]. K-D-B tree is a data

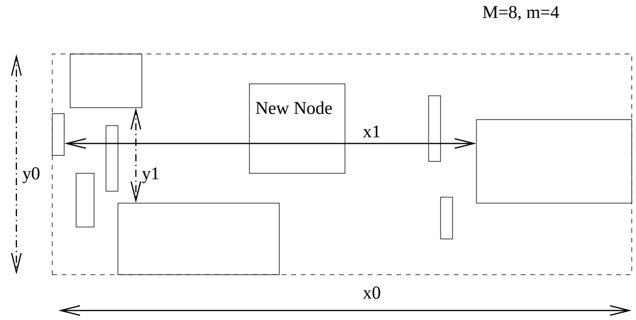


Figure 4. An example of linear approach.

structure that splits multidimensional spaces like an adaptive K-D tree [3], 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.

5) R*-tree: Beckmann et al. [2] 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 that has the maximum number of entries (M). 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.

6) Hilbert Method: Kamel et al. [9] 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., [13]) 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^+-tree. The Hilbert space filling curve has been shown to give a degree of space utilization as high as is desired.
7) 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].

IV. A 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 ($Rtree$) of an R-tree is defined as follows:

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

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

We also define a *tree-root* vector of an R-tree as follows:

Table 1. An adjacency matrix representing the R-tree in Figure 1(b).

	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	R_{11}	R_{12}	R_{13}	R_{14}	R_{15}	R_{16}	R_{17}	R_{18}	R_{19}
R_1	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R_2	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
R_3	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0
R_4	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0
R_5	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0
R_6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0
R_7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1
R_8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R_9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R_{10}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R_{11}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R_{12}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R_{13}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R_{14}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R_{15}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R_{16}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R_{17}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R_{18}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R_{19}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

$$\textit{tree-root}[i]_n = \begin{cases} 1 & \text{if } R_i \text{ is a node in the root of an R-tree,} \\ 0 & \text{otherwise,} \end{cases}$$

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

In addition to the structures just introduced, the coordinates of its bounding boxes also need to be stored in order to find overlapped MBRs 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 5.

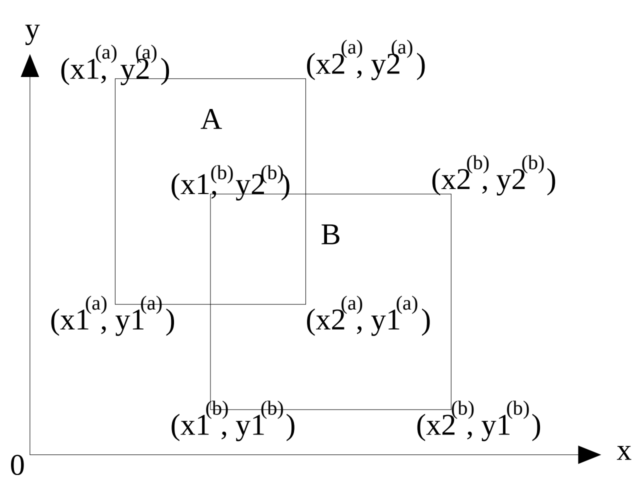


Figure 5. An example of two overlapped bounding boxes.

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

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

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

After we have the coordinates of all the MBRs, we can find out 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 5. 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, it has to be decided whether or not it overlaps with the query box. 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.

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
if (box-a-coor[2] < box-b-coor[1]) or (box-b-coor[2] < box-a-coor[1])
// Two boxes are not overlapped along X-coordinate.
   then overlap-x ← 0
else overlap-x ← 1 // Two boxes are overlapped along X-coordinate.
if (box-a-coor[4] < box-b-coor[3]) or (box-b-coor[4] < box-a-coor[3])
   then overlap-y ← 0 // Two boxes are not overlapped along Y-coordinate.
else overlap-y ← 1 // Two boxes are overlapped along Y-coordinate.
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])
for i ← 1 to n // in parallel
   do global-overlap-results[i] ← OVERLAP(Coor[i][], query-box-coor[]) // Find all nodes with their
                                       // bounding boxes overlapped with the query box.
current-search-nodes[] ← tree-root[]
while current-search-nodes[] contains at least one non-zero element
   do current-overlap-nodes[] ← current-search-nodes[] ∧ global-overlap-results[]
   // Find out whether or not any of the nodes being visited is overlapped with the query box.
      for i ← 1 to n // in parallel
         do if (current-overlap-nodes[i] = 1) and (R_i is a leaf node)
            then Record R_i as one of the qualifying leaf nodes // We find a leaf node with its bounding
                                       // box overlapped with the query box.
      for j ← 1 to n // in parallel
         do current-search-nodes[j] ← ∨_{1≤i≤n}(Rtree[i][j] ∧ current-overlap-nodes[i]) // Find all
                                       // children nodes whose parent nodes are found to have
                                       // overlaps with the query box in the current iteration.
return
```

* $\wedge$ and $\vee$ denote bitwise AND and OR, respectively.

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.

Table 2. Coordinates of R_1 to R_{19} and query box R_q shown in Figure 1(a).

	Coordinates																			
	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	R_{11}	R_{12}	R_{13}	R_{14}	R_{15}	R_{16}	R_{17}	R_{18}	R_{19}	R_q
x1	9	2	10	16	34	3	44	11	17	17	25	18	38	35	4	11	45	47	48	19
x2	43	58	21	29	42	22	57	15	20	20	28	26	41	39	8	21	56	50	53	24
y1	0	23	5	1	8	30	30	13	6	12	2	20	9	15	35	31	32	31	35	24
y2	37	43	19	36	29	42	41	18	10	15	35	25	28	21	41	41	37	40	39	33

Given two nodes R_1 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 $x1^{(R_1)} < x1^{(R_q)} < x2^{(R_q)} < x2^{(R_1)}$ (i.e., $9 < 19 < 24 < 43$) and $y1^{(R_1)} < y1^{(R_q)} < y2^{(R_q)} < y2^{(R_1)}$ (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 $x1^{(R_q)} < x2^{(R_q)} < x1^{(R_1)} < x2^{(R_1)}$ (i.e., $19 < 24 < 48 < 53$) and $y1^{(R_q)} < y2^{(R_q)} < y1^{(R_1)} < y2^{(R_1)}$ (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 know 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. 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.

After vector *global-overlap-results[]* is computed, two nodes R_1 and R_2 in the root are firstly visited. As one of the inputs to **R-TREE-SEARCH**, vector *tree-root[1:19]*, defined in Table 4, shows that 1^{st} and 2^{nd} elements are the nodes in the root of the R-tree. Next, the search is carried out in the **while**

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

R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	R_{11}	R_{12}	R_{13}	R_{14}	R_{15}	R_{16}	R_{17}	R_{18}	R_{19}
1	1	0	1	0	1	0	0	0	0	0	1	0	0	0	1	0	0	0

loop (lines 11 through 17). In the rest of this sub-section, we go through each iteration of the **while** loop until the loop ends.

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

R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	R_{11}	R_{12}	R_{13}	R_{14}	R_{15}	R_{16}	R_{17}	R_{18}	R_{19}
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

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 find out 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:

$$\begin{aligned} \textit{current-overlap-nodes[]} &= \textit{current-search-nodes[]} \wedge \textit{global-overlap-results[]} \\ &= [1100000000000000000] \wedge [1101010000010001000] \\ &= [1100000000000000000] \end{aligned}$$

where $\wedge$ 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. 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 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:

$$\begin{aligned}
\textit{current-search-nodes[3]} &= (\textit{Rtree[1][3]} \wedge \textit{current-overlap-nodes[1]}) \\
&\quad \vee(\textit{Rtree[2][3]} \wedge \textit{current-overlap-nodes[2]}) \\
&\quad \ldots \\
&\quad \vee(\textit{Rtree[19][3]} \wedge \textit{current-overlap-nodes[19]}) \\
&= (1 \wedge 1) \vee (0 \wedge 1) \vee \ldots \vee (0 \wedge 0) \\
&= 1
\end{aligned}$$

where $\vee$ denotes bitwise OR. As seen from 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.

R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	R_{11}	R_{12}	R_{13}	R_{14}	R_{15}	R_{16}	R_{17}	R_{18}	R_{19}
0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0

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:

$$\begin{aligned}
\textit{current-overlap-nodes[]} &= \textit{current-search-nodes[]} \wedge \textit{global-overlap-results[]} \\
&= [0011111000000000000] \wedge [1101010000010001000] \\
&= [0001010000000000000]
\end{aligned}$$

From the new values of vector *current-overlap-nodes[]*, we know that the bounding boxes of R_4 (child of R_1) and R_6 (child of R_2) are overlapped with query box R_q. However, neither of them are leaf nodes. Thus, the children nodes of both R_4 and R_6 are calculated and stored in vector *current-search-nodes[]*. The new values of vector *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 *current-search-nodes[]* after the second iteration.

R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	R_{11}	R_{12}	R_{13}	R_{14}	R_{15}	R_{16}	R_{17}	R_{18}	R_{19}
0	0	0	0	0	0	0	0	0	0	1	1	0	0	1	1	0	0	0

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

$$\begin{aligned} \textit{current-overlap-nodes[]} &= \textit{current-search-nodes[]} \wedge \textit{global-overlap-results[]} \\ &= [0000000000110011000] \wedge [1101010000010001000] \\ &= [0000000000010001000] \end{aligned}$$

From the result, we know that the bounding boxes of R_{12} and R_{16} are overlapped with query box R_q. Since R_{12} and R_{16} are both leaf nodes, they are fetched into the result set. Next in lines 16 to 17, new values of vector *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 *current-search-nodes[]* become all zeros at the end of the third iteration as shown in Table 7.

Table 7. Values of vector *current-search-nodes[]* after the third iteration.

R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	R_{11}	R_{12}	R_{13}	R_{14}	R_{15}	R_{16}	R_{17}	R_{18}	R_{19}
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Since vector *current-search-nodes[]* now contains all zeros, the **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 **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 6. On the left side of Figure 6, the "Coordinates" block represents dozens of registers that store the values of coordinates (i.e., *x1, x2, y1* and *y2*) of the R-tree bounding boxes and the query box (values in matrix *Coor[]*). On the right side, the "R-tree" block represents the registers that store the values of adjacency matrix *Rtree*. Each "RT" cell in the figure contains an element in matrix *Rtree*. Elements in vectors *global-overlap-results[], current-overlap-nodes[]* and *current-search-nodes* are stored in the cells marked with "global overlap results", "current overlap nodes" and "current search nodes", respectively. We use the Mentor Graphics ASIC Design Kit 3.0 [4] with the TSMC .25 μm standard cell library [15] to synthesize the described R-Tree-Search hardware unit. The area of a synthesized R-Tree-Search unit is 199 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 next section.

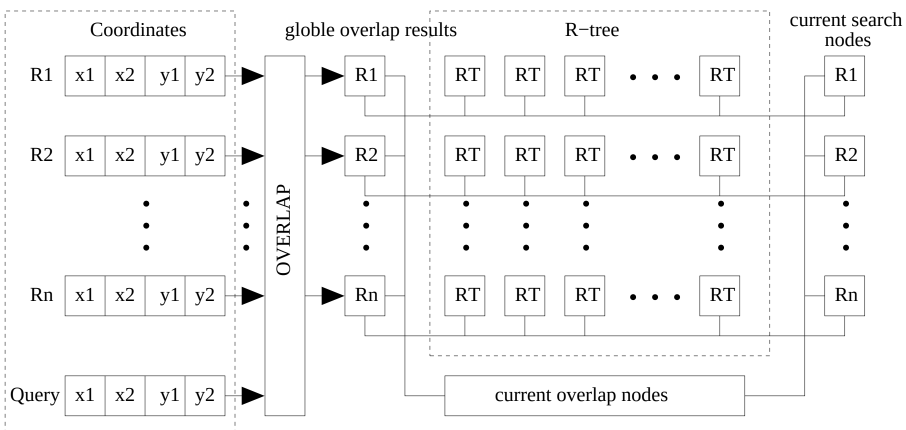


Figure 6. Diagram of hardware implementation of the parallel R-tree search algorithm.

VI. Experiments and Discussion

A. Simulation of hardware implementation

To validate the correctness of the proposed parallel R-tree search algorithm, we implement it in Verilog HDL and simulate our implementation using ModelSim v6.2 [6]. Our implementation follows the hardware diagram depicted in Figure 6, and performs the search example described in Section IV-C.

Through the simulation, we demonstrate that our implementation finishes the search process described in Section IV-C in three clock cycles, which is determined by the number of iterations that the **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 (Section IV-C) has three layers, the search algorithm finishes in three cycles.

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

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).

```
SF-RTREE-SEARCH(node, querybox)
if node has overlaps with querybox
   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)
```

To ensure a fair comparison, we construct a simulated computer system using Mentor Graphics Seamless [5], which can run both the hardware and software version 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 6 (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 [14] 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 for each program from the moment it is started till it ends. According to our simulation, the program that controls the search algorithm in hardware takes 103 clock cycles to finish the search while the software search program takes 3,393 clock cycles to finish the search. These results show that for a problem whose size can be accommodated by hardware capacity, performance gain compared to software version of the search algorithm is decent. 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 algorithm needs to be optimized to reduce area overhead. In the future, more research will be conducted to study about how to properly handle problems whose sizes exceed hardware capacity to maintain good performance of the search algorithm overall.

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 report, 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 performance gain of such a search algorithm in hardware can be 30 folds faster than the same algorithm implemented in software. 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 RSFG internal grant that enabled the research performed here. We also 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

[1] Lars Arge, Mark de Berg, Herman J. Haverkort, and Ke Yi. The Priority R-tree: A practically efficient and worst-case-optimal R-tree.

[2] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD Conference*, pages 322–331, 1990.

[3] Paul E. Black. Adaptive K-D tree. Dictionary of Algorithms and Data Structures. NIST.

[4] Mentor Graphics Corp. Asic design kit. http://www.mentor.com/company/higher ed/asic/index.cfm, visited Dec. 2007.

[5] Mentor Graphics Corp. Hardware/software co-verification: Seamless. http://www.mentor.com/seamless, visited in Jan. 2008.

[6] Mentor Graphics Corp. Modelsim. http://www.mentor.com/modelsim, visited in Jan. 2008.

[7] Raphael A. Finkel and Jon Louis Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Inf.*, 4:1–9, 1974.

[8] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD Conference*, pages 47–57, 1984.

[9] Ibrahim Kamel and Christos Faloutsos. On packing R-trees. In *CIKM*, pages 490–499, 1993.

[10] P.Rigaux, M. Scholl, and A. Voisard. *Spatial Databases - with application to GIS*. Morgan Kaufmann, San Francisco, 2002.

[11] Frank Ramsak, Volker Markl, Robert Fenk, Martin Zirkel, Klaus Elhardt, and Rudolf Bayer. Integrating the UB-tree into a database system kernel. In *VLDB*, pages 263–272, 2000.

[12] John T. Robinson. The K-D-B-tree: A search structure for large multidimensional dynamic indexes. In *SIGMOD '81: Proceedings of the 1981 ACM SIGMOD international conference on Management of data*, pages 10–18, New York, NY, USA, 1981. ACM.

[13] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990.

[14] D. Sun, D. Blough, and V. Mooney. Atalanta: A new multiprocessor RTOS kernel for system-on-a-chip applications. Technical Report GIT-CC-02-19, College of Computing, Georgia Tech, 2002.

[15] TSMC. Taiwan Semiconductor Manufacturing Co. Ltd. http://www.tsmc.com/, visited Dec. 2007.