Performance of Data-Parallel Spatial Operations*

Erik G. Hoel†
Geography Division
Bureau of the Census
Washington, D.C. 20233
hoel@cs.umd.edu

Hanan Samet
Computer Science Department
Center for Automation Research
Institute for Advanced Computer Sciences
University of Maryland
College Park, Maryland 20742
hjs@cs.umd.edu

Abstract

The performance of data-parallel algorithms for spatial operations using data-parallel variants of the bucket PMR quadtree, R-tree, and R^+-tree spatial data structures is compared. The studied operations are data structure build, polygonization, and spatial join in an application domain consisting of planar line segment data. The algorithms are implemented using the scan model of parallel computation on the hypercube architecture of the Connection Machine. The results of experiments reveal that the bucket PMR quadtree outperforms both the R-tree and R^+-tree. This is primarily because the bucket PMR quadtree yields a regular disjoint decomposition of space while the R-tree and R^+-tree do not. The regular disjoint decomposition increases the potential for interprocessor communication and parallelism in the bucket PMR quadtree, thereby enabling the execution times to decrease relative to those needed by the R-tree and R^+-tree.

1 Introduction

Parallel database systems have been the subject of increasing attention. This is due in part to the advent of highly parallel architectures, adoption of the relational model, and challenges posed by object-oriented systems [10, 19]. Much of the parallel database research has focused on multi-attribute declustering techniques (such as Bubba's extended range declustering [5] and multi-attribute grid declustering [13]), data placement [8], and intra-operator parallelization [9]. Topics such as algorithms for manipulating relations containing highly skewed attribute values, and parallel spatial data structures and algorithms remain open.

Prior research in the spatial domain has been limited to quadtrees and R-trees, with different goals. The quadtree research [3] was conducted under the data-parallel SAM model of computation, and its goal was the development of algorithms to operate on the data structure in parallel. The R-tree research has concentrated on the development of algorithms for building data-parallel R-trees and polygonization [16], as well as spatial joins for both data-parallel bucket PMR quadtrees and data-parallel R-trees [17]. This work, and the results we report here, differs significantly from other approaches in that we make use of many processors to execute the spatial queries rather than merely store the data on parallel disks while operating with a single cpu (e.g., [18]).

Our emphasis is on the performance of spatial operations in a data-parallel environment when the data is represented using hierarchical spatial data structures [21, 22]. Our approach is similar in spirit to an earlier study [15] in that the same data structures are examined (i.e., the R-tree, PMR quadtree, and the R^+-tree). The difference is that here we test operations requiring a significant amount of computation so that using parallelism may be attractive. Thus we do not study point operations such as finding the nearest line to a point as in [15]. Instead, we examine more complex operations such as data structure creation, polygonization, and spatial join.

In this paper our sample spatial database is one that contains collections of line segments (i.e., maps) corresponding to features such as roads, railway lines, boundaries of political and economic units, utility data, etc. Data structure creation is the time neces-

*This work was supported in part by the National Science Foundation under Grant IRI-92-16970.

†Also with the Center for Automation Research at the University of Maryland.

**Proceedings of the 20th VLDB Conference
Santiago, Chile, 1994**

sary to build the data structure for a particular map. This is an important issue as when the data structure is used for just one query, it may not be worthwhile to expend much effort in its construction. Polygonization is the process of determining all closed polygons formed by a collection of planar line segments. For example, it can be used to find the boundaries of all countries in the world. Both data structure creation and polygonalization involve just one data set.

In contrast, the spatial join involves two data sets. It is one of the most common operations in spatial databases. This term is usually used in conjunction with a relational database management system [11]. In that context, a join is said to combine entities from two data sets into a single set for every pair of elements in the two sets that satisfy a particular condition. These conditions usually involve specified attributes that are common to the two sets. In the spatial variant of the join, the condition is interpreted as being satisfied (i.e., two elements are joined) when the elements of the pair cover some part of the space that is identical. In the sequential domain, this problem has been studied algorithmically and empirically for the R-tree [6], while in the data-parallel domain it has only been studied in an algorithmic context [17].

We examine a variant of the spatial join that seeks to find all line segments that lie within a given distance of line segments of another type (the line segments need not be contiguous). This is the spatial analog of a range query (also termed a window) in a conventional database where the query region is not limited to a rectangle. It is also known as a corridor or a buffer zone in GIS, or image dilation in image processing. As an example, suppose that we have one map corresponding to the roads in the United States and another map corresponding to the border of Colorado and we want to determine all roads that lie within 10 miles of the border of Colorado.

In this paper we focus on representations that sort the data with respect to the space that it occupies. This results in speeding up operations involving search. The effect of the sort is to decompose the space from which the data is drawn (e.g., the two-dimensional space containing the lines) into regions called *buckets*. One approach known as an R-tree [14] buckets the data based on the concept of a minimum bounding (or enclosing) rectangle. In this case, lines are grouped (hopefully by proximity) into hierarchies, and then stored in another structure such as a B-tree [7]. The drawback of the R-tree is that it does not result in a disjoint decomposition of space — that is, the bounding rectangles corresponding to different lines may overlap. Equivalently, a line may be spatially contained in several bounding rectangles, yet it is only associated with one bounding rectangle. This means that a spatial query may often require several bounding rectangles to be checked before ascertaining the presence or absence of a particular line.

The non-disjointness of the R-tree is overcome by a decomposition of space into disjoint cells. In this case, each line is decomposed into disjoint sublines such that each of the sublines is associated with a different cell. There are a number of variants of this approach. They differ in the degree of regularity imposed by their underlying decomposition rules and by the way in which the cells are aggregated. The price paid for the disjointness is that in order to determine the area covered by a particular line, we have to retrieve all the cells that it occupies. Here we study two methods: the R^+-tree [12] and a variant of the PMR quadtree [20].

The R^+-tree partitions the lines into arbitrary sublines having disjoint bounding rectangles which are grouped in another structure such as a B-tree. The partition and the subsequent groupings are such that the bounding rectangles are disjoint at each level of the structure. The drawback of the R^+-tree is that the decomposition is data-dependent. This makes it difficult to perform tasks that require composition of different operations and data sets (e.g., set-theoretic operations such as overlay). In contrast, the PMR quadtree is based on a regular decomposition. The space containing the lines is recursively decomposed into four equal area blocks on the basis of the number of lines that it contains. We use a variant termed a ***bucket PMR quadtree*** that decomposes the space whenever it contains more than b lines (b is termed the ***bucket capacity***). The decomposition process can be implemented by a tree structure. It is useful for set-theoretic operations as the partitions of the two data sets occur in the same positions.

As mentioned above, R-trees and R^+-trees are closely related to B-trees. An R-tree or R^+-tree of order (m, M) has the property that each node in the tree, with the exception of the root, contains between $m \leq \lceil M/2 \rceil$ and M entries. The root node has at least 2 entries unless it itself is a leaf node. Thus we see that the node capacity M in the R-tree and R^+-tree plays the same role as the bucket capacity in the bucket PMR quadtree. We will make use of this analogy in our discussion where, at times, the terms will be used interchangeably.

The problem with using the R-tree and R^+-tree data structures to perform a spatial join is that they do not contain any information to help us in determining which bounding rectangles in one map overlap with bounding rectangles in the other map. This means that little of the search space can be pruned while performing the operations. The difficulty is that although

the R-tree and R^+ tree's main utility is to enable the user to distinguish easily between occupied and unoccupied regions in a particular map, they do not provide a means of correlating the contents of one map with another map. Unfortunately, this is exactly the ability that is needed to implement spatial join algorithms efficiently. As we will see, this places the data-parallel R-tree and R^+-tree at a considerable disadvantage in comparison to the data-parallel bucket PMR quadtree as it reduces the potential for interprocess communication thereby resulting in greater execution times for the data-parallel R-tree and R^+-tree.

We use the scan model of parallel computation [4]. The scan model is defined in terms of a collection of primitive operations that can operate on arbitrarily long vectors (single dimensional arrays) of data. Three types of primitives (elementwise, permutation, and scan) are used to produce result vectors of equal length. A *scan* operation [4, 3] takes an associative operator $\bigoplus$, a vector $[a_0, a_1, \cdots, a_{n-1}]$, and returns the vector $[a_0, (a_0 \bigoplus a_1), \cdots, (a_0 \bigoplus a_1 \bigoplus \cdots \bigoplus a_{n-1})]$. The scan model considers all primitive operations (including scans) as taking unit time on a hypercube architecture. This allows sorting operations to be performed in $O(\log n)$ time.

The rest of this paper is organized as follows. Section 2 gives the construction and polygonization algorithms for the data-parallel bucket PMR quadtree. The data-parallel R-tree algorithms are not presented here as they can be found elsewhere [16, 17]. Section 3 is concerned with the data-parallel R^+-tree and contains a description of the construction, polygonalization, and spatial join algorithms. Section 4 compares the three data-parallel data structures in terms of performance data for the specified operations on a Thinking Machines CM-5 parallel computer. Section 5 contains concluding remarks as well as a discussion of topics for future research. In our discussion of the various data structures, in the interest of brevity, we will drop the qualifier *data parallel* unless the distinction needs to be emphasized in the case of a potential for misunderstanding a claim.

2 Bucket PMR Quadtrees

In this section we discuss the implementation of the bucket PMR quadtree algorithms for the spatial operations that we examined. We give the construction and polygonization algorithms as they have not been formally presented before. See [17] for the bucket PMR quadtree spatial join algorithm.

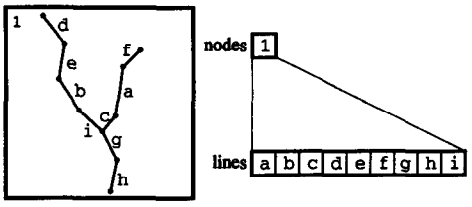

Figure 1: Initial bucket PMR quadtree processor assignments.

2.1 Bucket PMR Quadtree Construction

A bucket PMR quadtree is built as follows. Initially, a single processor is assigned to each line in the data set, and one processor to the resultant bucket PMR quadtree as depicted for the sample data set in Figure 1. Using a downward scan operation, the number of lines associated with the single node processor (9 in the example) is determined and then passed to the node processor. If the number of lines associated with the node processor exceeds the bucket capacity (2 in our example), then the node must be split into four subnodes and each of the lines must be regrouped, according to the nodes it intersects.

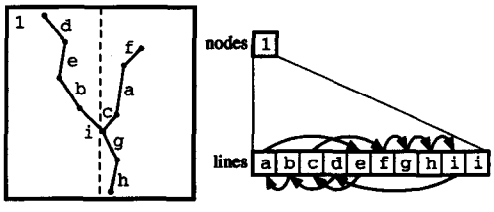

Figure 2: Result of applying the un-shuffle operation to the x coordinate value of the center of the block associated with the node processor.

The splitting occurs in two stages. The regrouping is applied after each split and is achieved with an *unshuffle* operation [3] (where two intermixed types are rearranged into two disjoint groups termed *segments* via two monotonic mappings) which is used to concentrate those line processors together into two new segments, each of which corresponds to all of the line processors lying either in whole or in part to the left and right of the x coordinate value of the center of the block associated with the node processor. The result of this un-shuffle operation is depicted in Figure 2. This is achieved by monotonically shifting to the left (right) all line processors with a midpoint less (greater) than the split coordinate value. Note that a line may span two or even three nodes, thus requiring the line to be duplicated or even triplicated and hence either one or two additional processors in the line processor set are allocated for it (termed *cloning* [3]). For example, consider line i in the process of subdividing the first node

in Figure 2. Here we see that line i intersects both the left and right halves of the root node.

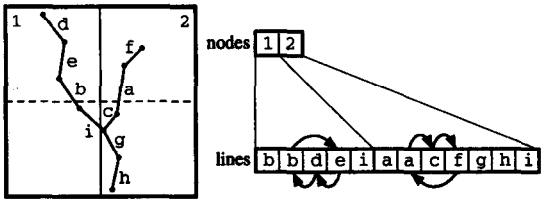

Figure 3: Result of applying the un-shuffle operation to the y coordinate value of the center of the block associated with the node processor.

The second stage applies the un-shuffle to the resulting two segments, thereby creating two sets of two segments each of which corresponds to all of the line processors which lie either in whole or in part below and above the y coordinate value of the center of the block associated with the node processor. The result of this un-shuffle operation is depicted in Figure 3.

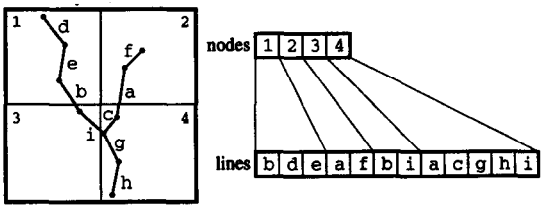

Figure 4: Result of the first node subdivision, line duplication and un-shuffling.

Continuing with this iterative process, each line segment group determines the number of lines it contains, and then communicates the count to the associated node processor. For example, in Figure 4, the first line segment group transmits a count of three to node 1, the second line segment group transmits a count of two to node 2, etc. Each of the node processors then determines whether or not the transmitted line count exceeds the bucket capacity. If the bucket capacity is exceeded, the node will subdivide, and the associated lines will be regrouped according to which of the resulting subnodes they intersect. For example, in Figure 4, the NW and SE nodes will subdivide.

This iterative subdivision process continues until all nodes in the bucket PMR quadtree have a line count less than or equal to the bucket capacity, or the maximal resolution of the quadtree has been reached (i.e., a node of size 1×1). Note that in the degenerate case, even at the maximal resolution of the quadtree, it is possible that the number of lines associated with a node exceeds the bucket capacity. For practical splitting thresholds (i.e., 8 and above), this situation is exceedingly rare and will not cause any algorithmic difficulties provided that the bucket PMR quadtree algorithms do not assume an upper bound on the number of lines associated with a given node.

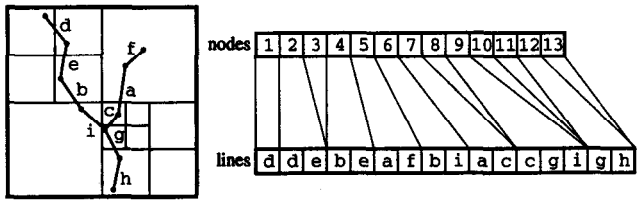

Figure 5: Result of the bucket PMR quadtree build process.

The result of the third and final subdivision for our example data set is shown in Figure 5. Note that one of the quadtree nodes (node 9) still has its bucket capacity exceeded. To facilitate the discussion of the algorithms, this node will not be further subdivided. The bucket PMR quadtree building operation takes $O(\log n)$ time, where each of the $O(\log n)$ subdivision stages requires $O(1)$ computations (a constant number of scans and re-shuffles).

2.2 Bucket PMR Quadtree Polygonization

Polygonization proceeds as follows. Identify each polygon uniquely by the bordering line with the lexicographically minimum identifier (i.e., line number) and the side on which the polygon borders the line. Polygonization can be achieved without a spatial data structure. Basically, the lines can be sorted according to their identifier in $O(\log n)$ time. Next, each line, in sorted sequence, transmits its endpoint coordinates, line identifier, and current left and right polygon identifiers to all following lines via a sequence of $O(n)$ scan operations. Each line can independently determine the identifiers of the left and right polygons. The drawback is that it is an $O(n)$ operation with a large number of scans. Data-parallel variants of spatial data structures such as the bucket PMR quadtree, as well as the R-tree and R$^+$-tree can reduce the number of global scan operations (i.e., a scan across the entire processor set) by instead relying upon segmented scans executed in parallel.

Given a bucket PMR quadtree, the polygonization process begins by constructing a partial winged-edge representation [1] (an association between the incident line segments forming the minimal and maximal angles at each endpoint of each segment). This representation enables us to determine all edges that comprise a face (i.e., polygon) and all edges that meet at a vertex in time proportional to the number of edges. In constructing the partial winged-edge representation, the endpoints of each line in a node are broadcast to

all other lines in the node through a series of scans. By *broadcast* we mean the process of transmitting a constant value from a single processor to all other processors in the same node via a scan operation (i.e., the vector $[a_0, a_0, \ldots, a_0]$). Locally, each line processor maintains the minimal and maximal angles formed at each endpoint as well as the identities of the corresponding lines. Once the broadcasts are done, each line processor locally assigns an initial polygon identifier for the bordering polygon on the left and right side (moving from source to destination endpoint).

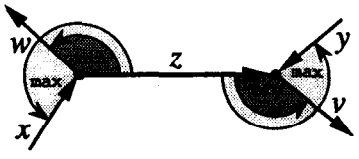

Figure 6: Selecting the initial polygon identifiers.

In Figure 6, the left polygon identifier for line segment z is selected from the minimum identifiers of the source endpoint minimal angle (w_R, where w is the line identifier and R denotes the right side of w), the destination endpoint maximal angle (y_R), and the line identifier itself (z_L). For the right polygon identifier, select the minimum identifier among the source endpoint maximum angle (x_R), the destination endpoint minimal angle (v_R), and the line identifier (z_R). In Figure 6, line z is assigned w_R as the initial left polygon identifier, and v_R as the right polygon identifier.

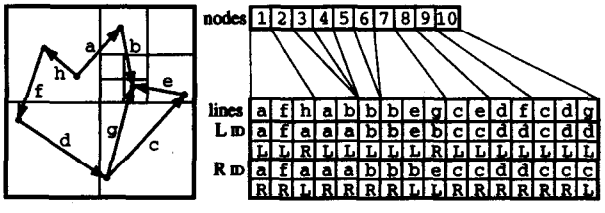

Figure 7: Initial polygon assignments.

Figure 7 shows the initial polygon assignment for the depicted example where the left and right polygon identifiers are contained in L_{ID} and R_{ID}, respectively.

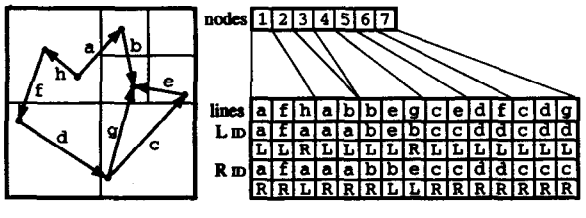

Figure 8: Polygon assignments after the first round of leaf node merging.

Starting at the leaf level, sibling nodes are then merged together into their parent nodes (i.e., in Figure 7, leaf nodes 4–7 are merged together, resulting in leaf node 4 in Figure 8). All the lines in the merged sibling leaf nodes are sorted, and any duplicate lines are marked. In Figure 7, the merging of sibling leaf nodes 4–7 will result in one pair of duplicate lines (line b) as there is a line b in nodes 5 and 7. In order to ensure that each duplicate line has consistent polygon identifiers as well as correct winged-edge representations, each duplicate line has its endpoints and polygon identifiers broadcast to the other duplicate lines in the merged node. If any of the duplicates' polygon identifiers are updated, the identifier updates must also then be broadcast among all other lines in the merged nodes. By *update*, we mean assigning a lexicographically smaller polygon identifier. For instance in Figure 8 the merging of sibling leaf nodes 2–5 will result in two pairs of duplicate lines (i.e., lines b and e). With the duplicate line b in the merged node, initially one instance has left and right polygon identifiers a_L and a_R, and the second instance has polygon identifiers b_L and b_R. The left and right polygon identifiers of the second instance of line b are updated from b_L to a_L, and b_R to a_R respectively.

When the second instance of line b is updated, the two identifier updates are then broadcast to all other lines in the merged node. For each other line in the merged node, if the transmitted polygon identifier update matches either of its current left or right polygon identifiers (i.e., the b_L to a_L update matches any line's left or right polygon identifier having value b_L) the line's polygon identifier is changed to a_L in order to reflect the broadcast update and the lexicographically smaller identifier.

Similarly, the duplicate line e results in two additional identifier updates — that is, c_R to b_L, and e_L to c_L. Actually, line e's b_L was previously updated to a_L during line b update broadcasts.

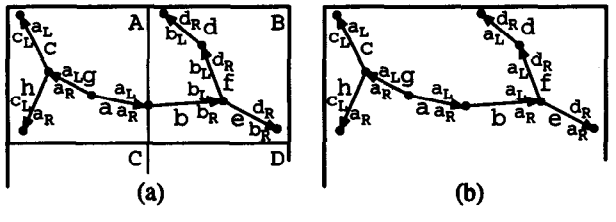

Figure 9: (a) Example of two leaf nodes A and B merging (the contents of sibling nodes C and D are not shown), and (b) the result of the merge operation.

Finally, when merging four sibling nodes together, any line whose endpoint falls on the shared node border (i.e., lines a and b in Figure 9a), must also have their endpoints and polygon identifiers broadcast

among the merged nodes. Consider the example in Figure 9a where four sibling nodes labeled A–D are being merged (for sake of clarity, the contents of nodes C and D are not shown). There are no duplicate lines in the merging nodes, but lines a and b have an endpoint that intersects the common node border. The endpoint coordinates and polygon identifiers of these two lines are broadcast among the merged lines, and any appropriate winged-edge updates are made (i.e., the source endpoint of line b is updated to reflect the incidence of line a). For all lines whose winged-edge representations are updated, the polygon identifiers are checked for possible updates. Figure 9b shows the resulting polygon identifiers.

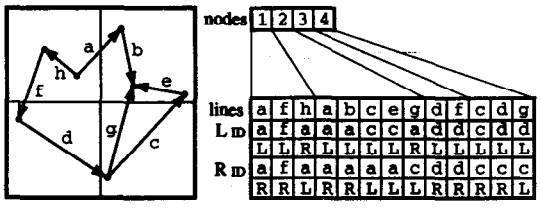

Figure 10: Polygon assignments after the second round of leaf node merging.

The merging and updating process continues up the entire bucket PMR quadtree until all lines are contained in a single node and all necessary broadcasts have been made (as shown in Figures 10 and 11, with the final assigned polygon identifiers circled).

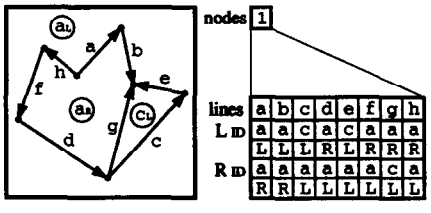

Figure 11: Completion of the polygonization operation.

The bucket PMR quadtree's spatial sort greatly limits the amount of inter-segment communication necessary as compared with a non-spatially sorted dataset where all lines would have to communicate their endpoints and polygon identifiers to all others.

3 R$^+$-trees

The R$^+$-tree algorithms are similar to those for the R-tree. The principal difference is in the amount of work needed in the data structure building phase to ensure a good node split. Below we give a brief outline of the algorithms for the various operations.

3.1 R$^+$-tree Construction

The R$^+$-tree construction algorithm is similar to that of the R-tree with a few additional modifications. Initially, one processor is assigned to each line of the data set, and one processor to the resultant R$^+$-tree. Within the line processor set, a downward scan operation is performed on the line processor set to determine the number of lines associated with the single R$^+$-tree node processor. The number of lines in the segment is then passed to the single R$^+$-tree node processor. If the number of lines in the segment exceeds the node capacity M, then the R$^+$-tree root node must be split into two leaf nodes and a root node. The two new leaf nodes are inserted into the R$^+$-tree node processor set, with the root node updated to reflect the two new children.

The R$^+$-tree node splitting algorithm first sorts all lines in the node according to the left edge of their bounding boxes. For each node split whose result satisfies a pre-established minimal node occupancy level of m/M lines in the two resulting nodes, the coordinate value of the left edge is broadcast to each of the lines in the node being split. Each line in parallel clips itself against the split coordinate value. The clip results in either one (the line does not intersect the split coordinate value) or two lines (the line intersects the split coordinate value). Each resulting line determines in which of the two new nodes it is contained. The definition of an R$^+$-tree requires that each node at a given level of the tree is disjoint from all other nodes. In order to ensure this disjoint decomposition, some lines will have to be split across multiple nodes in the final decomposition. This situation also arises in the bucket PMR quadtree. Once each line determines the node in which node it lies, a sequence of scan operations is used to determine the bounding box that contains the lines in the two new nodes. Finally, the perimeter of the two resulting bounding boxes is computed.

The splitting process continues for each of the legal node splits and split axes. Once all legal node splits have been determined and the resulting node perimeters are computed, the split axis and coordinate value that correspond to the minimal perimeter of the two resulting nodes is selected as the final node split value. In the event of a tie, some other metric such as the split with the minimal bounding box areas may be employed. After choosing the splitting axis and the coordinate value, an un-shuffle operation concentrates those line processors together into two new nodes, each of which corresponds to one of the two R$^+$-tree leaf node processors.

The insertion algorithm proceeds iteratively as described above, with each node determining the number

of lines it contains, and transmitting the count to the associated R^+-tree node processor. If the number of lines in the node exceeds M, then the node (and corresponding R^+-tree node processor) are split. Note that the leaf node subdivision process may result in processors that correspond to internal nodes in the R^+-tree being forced to split when the number of their children (e.g., leaf nodes) exceeds the node capacity. These internal node splits may possibly propagate up to the root node of the R^+-tree (and are referred to as upward splits).

An additional complication in the node splitting process arises if the splitting of an internal node forces the splitting of some of the descendents (both nodes and lines) of the splitting internal node. Unlike the R-tree which does not enforce a disjoint decomposition, an upward internal node split may result in the selection of a split axis and a coordinate value that intersects the descendents of the splitting node. The disjoint decomposition requires that any intersecting descendents (nodes or lines) must also be split. Splitting the descendents of a node is termed a downward split. The process terminates when all nodes in the node processor set have at most M child processors (either internal R^+-tree nodes or line processors).

3.2 R^+-tree Polygonization

The R^+-tree polygonization algorithm is very similar to that for the R-tree [16]. Because the R^+-tree employs a disjoint decomposition, a single line may reside in more than one leaf node (similar to the bucket PMR quadtree). In order to handle this difference with respect to the R-tree, the polygonization algorithm must be changed somewhat during the node merging phase.

Rather than marking all lines that intersect any of the overlapping regions formed by the bounding boxes of the nodes that are merging (as there are none with a disjoint decomposition), the update procedure follows the technique described in the bucket PMR quadtree polygonization algorithm in Section 2.2. All the lines in the merged sibling node are first sorted according to identifier, and all duplicate lines are marked for rebroadcasting among the lines in the merged nodes. This enables the correct updating of duplicate lines in the merged nodes. The duplicate node rebroadcasting operation is used to update the winged-edge representations of all duplicate lines and maintain consistency. During the update, we note any polygon identifiers that must also be updated (i.e., among duplicate lines, if one line has polygon identifiers that are less than the polygon identifiers of the second line). In addition, all lines whose endpoint falls on a common node border are marked for the rebroadcast of their endpoint coordinates in order to update the winged-edge representations and polygon identifiers of any line that may share an endpoint but lie in another node.

If any line has its polygon identifiers updated during the first round of rebroadcasting, then the polygon identifier update must be communicated in a second round of broadcasting to all other lines in the merged node. Locally, if the transmitted polygon update matches either the left or right polygon identifiers of the local line, then the local polygon identifier is updated to reflect the polygon identifiers that have been broadcast.

As is the case with the bucket PMR quadtree and R-tree polygonization algorithms, the merging and updating process continues up the entire R^+-tree until all lines are contained in a single node and all necessary broadcasts have been made.

3.3 R^+-tree Spatial Join

The R^+-tree spatial join algorithm is identical to the one used with the R-tree [17], with one small modification at the end of processing. Because the disjoint decomposition of the R^+-tree may cause some lines to be split across multiple leaf nodes, it may be the case that a line in the source map is only within a given distance of a portion of a line that has been split in the target map. Thus, some of the pieces of a particular line in the original target map may be marked as within a given distance, while other portions are not marked. In order to resolve this inconsistency among portions of lines that correspond to the same line in the original target map, once all intersection determinations are completed, the pieces of the target lines are sorted according to identifier. This results in all pieces of a line in the original target map occupying a contiguous space in the linear ordering of processors. An upward and a downward scan operation can be used to resolve any inconsistencies, resulting in all target lines being properly marked.

4 Performance Comparison

The performance of the three spatial structures in the data-parallel environment is compared using the Bureau of the Census TIGER/Line File map of Prince Georges County, MD (containing approximately 35000 line segments). Our data-parallel algorithms assume that the entire data structure resides in main memory of the Thinking Machines CM-5 (32 processors, 1 GB RAM). Thus measurements of I/O performance are meaningless in this context (the development of disk-based data-parallel analogs to the described algorithms is a subject for future research).

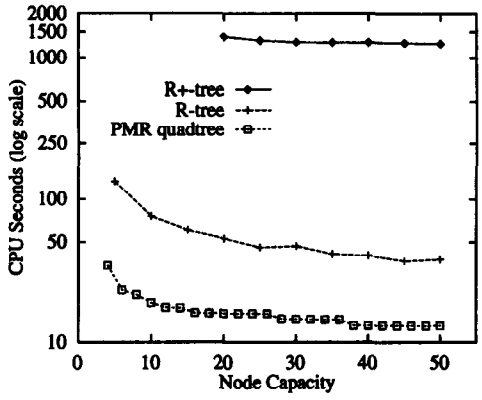

Figure 12: Build times for the three data structures for the map of Prince Georges County, MD (35000 lines).

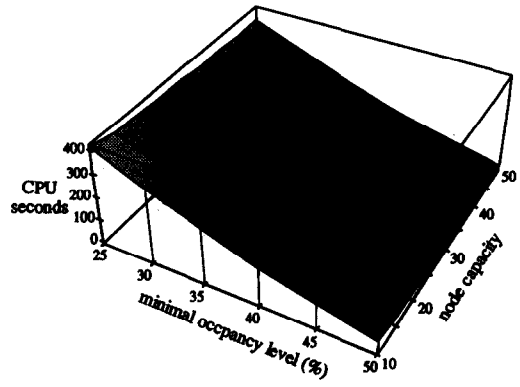

Figure 13: Execution time in seconds for the R^+-tree build algorithm for the map of Fredricksburg, VA (1700 lines).

4.1 Data Structure Build Performance

Figure 12 presents the build times for the three data structures for node capacities ranging from 5 to 50. The R^+-tree was built with a 49.5% minimal occupancy level (see the discussion below). From the figure, all three structures exhibit decreasing build times as the node capacities increase. This behavior is due to the decreased amount of spatial sorting that takes place with the increased node sizes. The three data structures exhibit analogous behavior in the sequential environment [15]. It is also apparent that the bucket PMR quadtree is approximately 3–4 times faster than the R-tree for similar node capacities. The relative difference in build performance is attributable to the use of a regular decomposition in the case of the bucket PMR quadtree which makes it very easy to split an overflowing node as there is just one choice. In contrast, the R-tree and the R^+-tree make use of irregular decomposition which requires testing a possibly large numbers of split axis/coordinate pairs in determining a locally optimal node split.

Figure 13 shows the build times for the R^+-tree of the Fredericksburg, VA map containing approximately 1700 line segments. In addition to varying the node capacity between 10 and 50, we also varied the minimal occupancy levels between 25% and 50% (as a point of reference, the best performance for an R-tree, termed an R^*-tree [2], has been observed to be 30% and is the one that we use in our experiments). When splitting a node, a minimal occupancy level of k% ensures that each of the two resulting nodes is at least k% full. Hence, when the minimal occupancy level is raised, fewer split axis/coordinate pairs are tested when choosing the best split. This results in increasing the speed of of the build process as can be seen in Figure 13. As is the case in the Prince Georges map, in Figure 12, increasing the node capacity also results in decreased build times.

It is important to note that although Figure 13 represents a map that is approximately 5% of the size of that in Figure 12 (i.e., 1700 lines versus 35000 lines), the R^+-tree takes 198.85 seconds to build while the R-tree (using a node capacity of 50 and a minimal occupancy level of 30%) for the same map requires 37.78 seconds to build and the bucket PMR quadtree requires just 12.97 seconds. We found that despite the R-tree and R^+-tree being quite similar in structure, the R^+-tree takes approximately 2 orders of magnitude longer to build per line segment in the dataset. This difference is attributable to a combination of the use of the scan model and the fact that the R-tree does not employ a disjoint decomposition of space (thus preventing the children of a splitting node from themselves splitting), making it possible to determine the locally optimum node split with a constant number (approximately 10) of upward and downward scan operations. In contrast, the node splitting process in the R^+-tree, with its disjoint decomposition of space, is an iterative process where the number of iterations is directly proportional to the number of items in the node that is being split. This testing for splits means that a large number of clipping operations must to be performed as we need to determine which part (or parts) of the line is associated with the two nodes resulting from the split.

Note that although the bucket PMR quadtree (with its disjoint decomposition) also requires line clipping, each line is clipped in parallel a maximum of 4 times the height of the tree. Also the fact that the bucket PMR quadtree employs a regular decomposition means that when a node is split, there are effectively only two candidate split axis/coordinate pairs.

It is interesting to observe that the R^+-trees that we built for the Prince Georges map used a minimal

node capacity	R-tree time	R-tree scans	R^+-tree time	R^+-tree scans
25	37.2	865	1309.3	28212
30	35.6	823	1274.6	27545
35	33.5	739	1268.0	27305
40	30.4	654	1269.2	27187
45	29.5	614	1261.3	27040
50	28.5	614	1246.6	26691

Table 1: Data structure build statistics for the R-tree and R+-tree both using a 49.5% minimal occupancy level for the Prince Georges map.

occupancy level of 49.5% (resulting in approximately 3000 line clips) and a node capacity varying between 25 and 50. This took between 1309.30 seconds and 1246.6 seconds as shown in Table 1. The analogous R-tree (employing the same node capacities and minimal occupancy levels), took between 37.2 seconds and 28.5 seconds. Note that if we would have used an R^+-tree with a minimal occupancy level of 30% (as in the R-tree), these numbers would have been at least one order of magnitude higher. Unfortunately, due to hardware and time limitations we were not able to perform these tests.

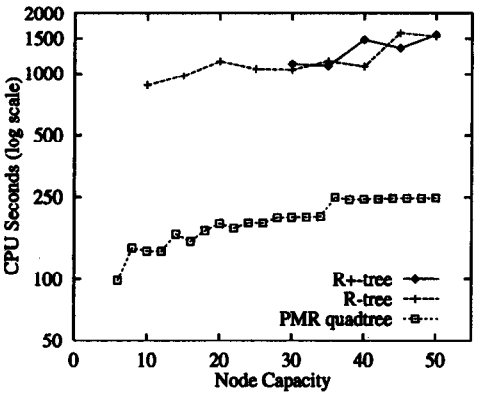

Figure 14: Polygonization times for the three structures.

4.2 Polygonization Performance

Figure 14 shows the execution times for map polygonization for each of the three spatial data structures using the Prince Georges maps built in Section 4.1. Due to the performance inefficiencies of the R^+-tree, a minimal occupancy level of 49.5% was employed, while the R-tree used the standard 30% level. From the figure it is clear that the bucket PMR quadtree offers significant performance advantages over both the R-tree and the R^+-tree. The difference is roughly one order of magnitude. It is attributable primarily to the considerable amount of time that the R-tree and the R^+-tree must spend in determining which nodes are intersecting (or adjoining in the case of the R^+-tree) when merging sibling nodes. For the bucket PMR quadtree, this computation is immediate as a result of regular decomposition. In addition, at each stage of the polygonization process, the R-tree and R^+-tree merge many more nodes/lines together (i.e., a node occupancy of n implies a fanout of n), while for the bucket PMR quadtree four nodes are merged together at each stage of the computation. Essentially, the bucket PMR quadtree performs a larger number (equal to the height of the tree) of smaller node merges (with respect to the number of nodes being merged) than the R-tree and the R^+-tree.

4.3 Spatial Join Performance

The key issue in the performance of the bucket PMR quadtree vis-a-vis the R-tree and the R^+-tree is the use of regular decomposition. Thus since the data-parallel algorithms for the R-tree and the R^+-tree are so similar, we only conducted limited tests on the R^+-tree. The performance of the R^+-tree will be worse than that of the R-tree because of the use of disjoint decomposition in addition to being irregular. Thus lines are broken into smaller portions resulting in correspondingly more leaf nodes. This leads to an increase in the intersection lists between source and target nodes and implies greater execution times.

In the interest of obtaining a better understanding of the R-tree spatial join operation, we tested both a top-down and bottom-up algorithm, while only a bottom-up algorithm was tested for the bucket PMR quadtree as this is the most logical approach to implement the operation. Similarly, we only tested the top-down algorithm for the R^+-tree. For additional comparison purposes, a brute-force solution that does not employ any spatial decomposition (i.e., each source line is broadcasted to each target line) was implemented as well. Note that the execution time of this brute-force approach is independent of the spatial join condition (i.e., the distance within which the desired lines are found).

For each of the spatial joins, the set of lines corresponding to railroads in the Prince Georges map (334 line segments) was chosen as the source map, while the set of lines corresponding to the road network in the Prince Georges map (28514 line segments in contrast to a total of 35000 line segments in the original map which includes all of the linear features rather than just the roads) was chosen as the target map. In this

case, the spatial join query is one that seeks to determine which roads are within a specified distance of a railroad line. The distance (i.e., radius of expansion) varied between 0 and 50 where the map was normalized on a scale of 16384 × 16384. In addition, the bucket capacity for the bucket PMR quadtree varied between 8 and 32, while the node capacity ranged between 10 and 50 for the R-tree and 25 to 50 for the R^+-tree.

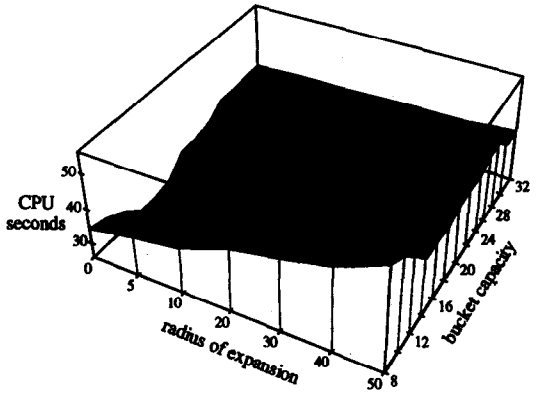

Figure 15: Execution time in seconds for the bucket PMR quadtree spatial join algorithm.

Figure 15 presents the cpu times for the bucket PMR quadtree spatial join operation as a function of the radius of expansion and the bucket capacity. We observe that for this map the execution time is at its minimum for a bucket capacity of roughly 14 to 16. As the radius of expansion increases toward 50, these bucket capacities continue to exhibit good performance although the advantage is not as great.

Two basic forces work against each other as the radius of expansion and bucket capacity increase. First, with a larger radius of expansion, fewer source lines are removed from consideration as we iterate at levels successively closer to that of the root node, thus resulting in more source line to target line endpoint transmissions. Second, as the bucket capacity increases for a fixed radius of expansion, we have fewer nodes but of larger capacity. The lessened node count results in a quadtree of shallower depth (which results in fewer iterations of the algorithm), but each iteration takes longer as more source line segments need to transmit their endpoint coordinates to the target lines.

Figure 16 shows the cpu times for the top-down R-tree spatial join as a function of the radius of expansion and the node capacity. Note that R-trees with smaller node capacities (i.e., 10 or 15) exhibit execution times that are considerably less than for larger node capacities (i.e., 45 or 50). The reason for this substantial difference in performance is that smaller node capacities result in a finer decomposition of space. In particular, each of the smaller source nodes intersects a

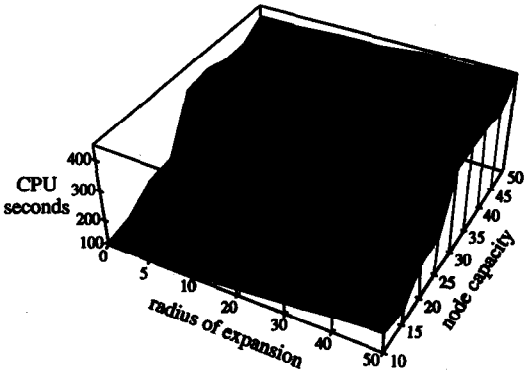

Figure 16: Execution time in seconds for the top-down R-tree spatial join algorithm.

smaller number of target nodes. With this finer granularity, there is increased opportunity for parallel communication when broadcasting the source lines to the appropriate target nodes.

Not surprisingly, the execution times for a fixed node capacity tend to increase as the radius of expansion increases. Similar to what was observed with the bucket PMR quadtree, the increased radius of expansion results in a greater number of source/target node intersections as the region around each source node that has a potential of being within the given distance of a target node is larger.

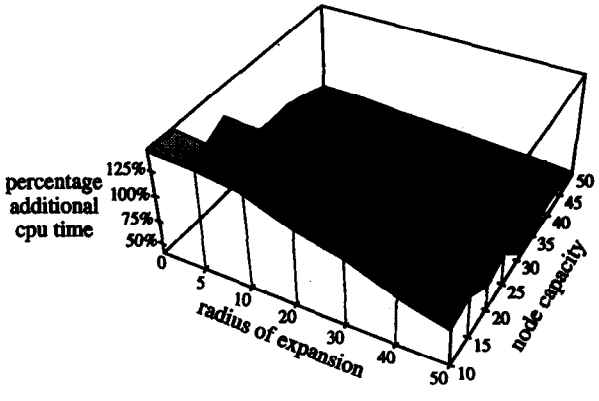

Figure 17: Percentage of additional execution time required by the leaf node intersection determination phase of the bottom-up R-tree spatial join algorithm relative to the top-down algorithm.

Figure 17 shows the percentage of additional execution time required by the node intersection phase of the bottom-up R-tree spatial join algorithm relative to the node intersection phase of the top-down R-tree spatial join algorithm. For the given node capacities and radii of expansion, the bottom-up procedure requires between 40–135% more cpu time to determine all node intersections. It should be clear that the top-down

algorithm (which makes full use of the R-tree decomposition) offers significant performance advantages as compared with the simpler bottom-up algorithm. The advantage of the top-down algorithm was pronounced when the node capacities were smallest (i.e., 10–25) and the corresponding tree height was greatest. Moreover, the top-down algorithm performed relatively better with a small radius of expansion. Unfortunately, the node intersection determination phase of the spatial join operation only consumes 2–25% of the entire algorithm (with the greatest fraction occurring when the node capacity and radius of expansion are small).

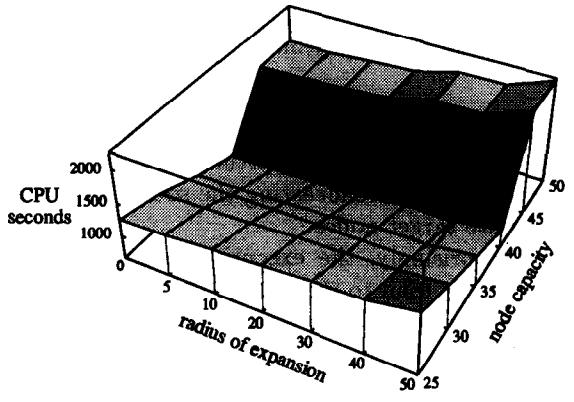

Figure 18: Execution time in seconds for the R^+-tree spatial join algorithm.

Figure 18 presents the cpu times for the R^+-tree spatial join operation. We observe that for this map, as the node capacity increases, the execution time falls (due to fewer leaf nodes); but then rises considerably for node capacities 45 and 50. This is due to the number of leaf nodes in the source map decreasing (thereby becoming larger, thus intersecting more target nodes and creating more communication conflicts). Note that execution times are larger than those for the corresponding R-tree (see Figure 16) as the disjoint decomposition results in about twice as many leaf nodes, thus increasing the amount of source to target node communication (as well as increasing the size of the intersection lists [17]). Finally, as is observed with the PMR quadtree and the R-tree, as the radius of expansion increases toward 50, the execution time increases.

When comparing the execution times of the bucket PMR quadtree and top-down R-tree and R^+-tree spatial join algorithms, it is apparent that the bucket PMR quadtree offers significant performance advantages. For example, consider Table 2 which lists the cpu times for the Prince Georges map's for the three data structures (each with a node capacity of 25) for a variety of source map expansions. For each of the listed expansions, the R-tree takes approximately 5–6 times longer than the corresponding bucket PMR quadtree,

expansion radius	CPU seconds PMR	R-tree	R^+-tree
0	34.01	203.95	1256.03
5	34.59	205.15	1289.43
10	34.94	205.92	1324.43
20	36.98	219.61	1362.80
30	37.59	227.10	1498.38
40	39.29	235.13	1444.19
50	42.96	238.43	1575.86

Table 2: Spatial join execution times for the three data structures for node capacity 25.

and the R^+-tree takes approximately 6 times longer than the corresponding R-tree. This performance advantage is primarily because the bucket PMR quadtree makes use of a regular disjoint decomposition of space which, in a data parallel environment, facilitates increased amounts of parallel communication between source and target maps in comparison to the R-tree and R^+-tree. This drawback of the R-tree and R^+-tree cannot be overcome by using classical R-tree improvements such as the R*-tree [2].

Our final comparison was designed to answer the question of whether using a spatial decomposition method is worthwhile. This was achieved by making use of a true brute-force approach where a spatial decomposition is not employed (i.e., each source line broadcasts to each target line). It proved superior to both R-tree algorithms in terms of the execution time required. The brute-force approach for the Prince Georges map required 54.95 cpu seconds, *regardless* of the radius of expansion. In contrast, the top-down R-tree spatial join algorithm required a minimum of 118.79 seconds for all combinations of node capacity and radius of expansion, while the bottom-up R-tree required a minimum of 151.26 seconds. On the other hand, our bucket PMR quadtree spatial join algorithms proved superior to the brute-force approach in all but one combination of splitting threshold and radius of expansion (the data parallel bucket PMR quadtree for the Prince Georges map required between 26.41 and 55.16 seconds).

Of course, we must bear in mind that these execution times are for two map spatial joins. If we were to implement single map versions of the queries (i.e., given a single map containing line segments representing both roads and railways being distinguished by appropriate attribute flags), the performance of the R-tree and R^+-tree would increase considerably; perhaps even to a level comparable to that displayed by the bucket PMR quadtree. Single map spatial join

algorithms are a topic for future research.

5 Concluding Remarks

Data-parallel algorithms for data structure construction, polygonization, and computing a spatial join for the bucket PMR quadtree, R-tree, and R^+-tree spatial data structures have been presented. Tests were conducted for each algorithm which revealed better performance for the bucket PMR quadtree. The main reason for this behavior is the fact that the bucket PMR quadtree yields a regular disjoint decomposition of space while this is not the case for the R-tree or the R^+-tree. Interestingly, for the spatial join, a brute-force approach that does not employ a spatial decomposition proved superior to both of our R-tree and R^+-tree implementations. This further emphasizes the penalty incurred by using either non-disjoint or irregular decompositions in the parallel domain.

References

[1] B. Baumgart, Winged-edge polyhedron representation, STAN-CS-320, Computer Science Dept., Stanford Univ., 1972.

[2] N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger, The R*-tree: an efficient and robust access method for points and rectangles, *Proc. of the SIGMOD Conf.*, Atlantic City, NJ, June 1990, 322–331.

[3] T. Bestul, Parallel paradigms and practices for spatial data, Ph.D. dissertation, CS-TR-2897, Computer Science Dept., Univ. of Maryland, April 1992.

[4] G. E. Blelloch, Scans as primitive parallel operations, *Proc. of the Intl. Conf. on Parallel Processing*, St. Charles, IL, August 1987, 355–362.

[5] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez, Prototyping Bubba, a highly parallel database system, *IEEE Trans. on Knowledge and Data Engineering*, 2, 1, March 1990, 4–25.

[6] T. Brinkhoff, H. P. Kriegel, and B. Seeger, Efficient processing of spatial joins using R-trees, *Proc. of the SIGMOD Conf.*, Washington, DC, June 1993, 237–246.

[7] D. Comer, The ubiquitous B-tree, *ACM Comp. Surveys 11*, 2 (June 1979), 121–137.

[8] G. Copeland, W. Alexander, E. Boughter, and T. Keller, Data placement in Bubba, *Proc. of the SIGMOD Conf.*, Chicago, June 1988, 99–108.

[9] D. J. DeWitt et. al., GAMMA - a high performance dataflow database machine, *Proc. of the VLDB Conf.*, Tokyo, August 1986., 228–237.

[10] D. J. DeWitt and J. Gray, Parallel database systems: the future of database processing or a passing fad?, *SIGMOD Record*, 19, 4 (December 1990), 104–112.

[11] R. Elmasri and S. B. Navathe, *Fundamentals of Database Systems*, Benjamin/Cummings, Redwood City, CA, 1989.

[12] C. Faloutsos, T. Sellis, and N. Roussopoulos, Analysis of object oriented spatial access methods, *Proc. of the SIGMOD Conf.*, San Francisco, May 1987, 426–439.

[13] S. Ghandeharizadeh, D. DeWitt, and W. Qureshi, A performance analysis of alternative multi-attribute declustering strategies, *Proc. of the SIGMOD Conf.*, San Diego, June 1992, 29–38.

[14] A. Guttman, R-trees: a dynamic index structure for spatial searching, *Proc. of the SIGMOD Conf.*, Boston, June 1984, 47–57.

[15] E. G. Hoel and H. Samet, A qualitative comparison study of data structures for large line segment databases, *Proc. of the SIGMOD Conf.*, San Diego, June 1992, 205–214.

[16] E. G. Hoel and H. Samet, Data-parallel R-tree algorithms, *Proc. of the 22nd Intl. Conf. on Parallel Processing*, St. Charles, IL, August 1993, 49–53.

[17] E. G. Hoel and H. Samet, Data-parallel spatial join algorithms, *Proc. of the 23rd Intl. Conf. on Parallel Processing*, St. Charles, IL, August 1994.

[18] I. Kamel and C. Faloutsos, Parallel R-trees, *Proc. of the SIGMOD Conf.*, San Diego, June 1992, 195–204.

[19] W. Kim, Research directions in object-oriented database systems, *Proc. of the PODS Conf.*, Nashville, April 1990, 1–15.

[20] R. C. Nelson and H. Samet, A consistent hierarchical representation for vector data, *Computer Graphics 20*, 4 (August 1986), 197–206.

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

[22] H. Samet, *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*, Addison-Wesley, Reading, MA, 1990.