CAR-TR-704 IRI-92-16970
CS-TR-3230 February 1994

Algorithms for Data-Parallel Spatial Operations

Erik G. Hoel[1]
Hanan Samet[2]

[1,2]Computer Science Department
[2]Institute for Advanced Computer Sciences
[1,2]Center for Automation Research
University of Maryland
College Park, MD 20742

Abstract

Efficient data-parallel algorithms for three common spatial data structures (the bucket PMR quadtree, R-tree, and R^+-tree) are presented. The domain consists of planar line segment data (i.e., Bureau of the Census TIGER/Line files). Parallel algorithms for building the data-parallel spatial structures, as well as determining the closed polygons formed by the line segments, map intersection, and a spatial range query are described. The performance of data-parallel algorithms for spatial operations is also compared. 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.

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

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 [DeWi90, Kim90]. Much of the parallel database research has focused on multi-attribute declustering techniques (such as Bubba's extended range declustering [Bora90] and multi-attribute grid declustering [Ghan92]), data placement [Cope88], and intra-operator parallelization [DeWi86]. Topics such as algorithms for manipulating relations containing highly skewed attribute values, and parallel spatial data structures and algorithms, remain open.

1.1 Prior Research in the Parallel Spatial Domain

Prior research in the spatial domain has been limited to quadtrees, k-d trees, and R-trees, with different approaches (i.e., image-space or object-space [Fole90]) and goals. The object-space approaches assign one processor per spatial object, while the image-space approach assigns one processor per region of space.

The quadtree research has primarily focussed on area (raster) data and region quadtrees. Much research has concentrated on algorithms for building (either in a top-down or bottom-up manner) both pointer-based and linear region quadtrees [Hung89, Dehn91, Ibar93]. Other efforts have focussed on developing neighbor finding techniques [Nand88, Hung89] as well as extracting region properties and performing set theoretic queries [Bhas88, Kasi88, Hung89, Dehn91]. Some of the work has employed proprietary parallel architectures (e.g., two-dimensional shuffle exchange network [Mei86], or DRAFT [Mart86]), mesh-connected computers [Hung89], or different programming languages (e.g., Concurrent Prolog [Edel85]), while the majority has dealt with hypercube architectures. Bestul [Best92] extended the quadtree research under the data-parallel SAM (for Scan-And-Monotonic-mapping) model of parallel computation. In addition to dealing with linear region quadtrees in the data-parallel [Hill86] context, algorithms were developed by Bestul for building and manipulating (e.g., set theoretic spatial queries) PR quadtrees [Oren82, Ande83, Rose83] and PM quadtrees [Webb84, Same85b].

The k-d tree [Bent75] research was limited but resulted in a description of an important algorithm for building the data structure for a collection of points using Blelloch's scan model of computation [Blel89b].

Some of the parallel R-tree research has focussed on algorithms for single cpu-multiple parallel

disk systems [Kame92]. Our R-tree research has concentrated on the development of algorithms for building data-parallel R-trees and polygonization [Hoel93], as well as spatial joins [Hoel94a, Hoel94b]. It differs significantly from the former approach 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., [Kame92]). Interestingly, the partitioning of data across parallel disks can be considered an image-space approach where one processor (or disk) is assigned to each region.

1.2 This Research

Our emphasis is on the performance of spatial operations in a data-parallel environment when the data is represented using hierarchical spatial data structures [Same90a, Same90b]. Our approach is similar in spirit to an earlier study [Hoel92] in that the same data structures are examined (i.e., the R-tree, the R^+-tree, and the PMR quadtree). 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 [Hoel92]. Instead, we examine more complex operations such as data structure creation, polygonization, and spatial join. It should be noted that the data-parallel algorithms are assumed to be main memory resident. Our application environment, a minimally configured Thinking Machines CM-5 with 32 processors, contains 1 GB of main memory. The adaptation of these main memory algorithms to disk-based variants is a subject of our ongoing research.

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 necessary to build the data structure for a particular map. This is an important issue, because 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 the 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 [Rote91, Günt93]. This term is usually used in conjunction with a relational database management system [Elma89]. 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 with polygonal data both algorithmically and empirically for the R-tree [Brin93], while in the data-parallel domain it has been studied with line segment data both algorithmically and empirically [Hoel94a, Hoel94b].

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; however, in our analog 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 Maryland and we want to determine all roads that lie within 10 miles of the border of Maryland.

1.3 Spatial Decompositions

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 [Gutt84] 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 [Come79]. 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. The reason is that each line is

decomposed into as many pieces (termed *q-edges*) as there are cells through which it passes. Here we study two methods: the R^+-tree [Falo87] and a variant of the PMR quadtree [Nels86].

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 two 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.

1.4 Spatial Joins

There are a number of possible ways to implement a spatial join. In order to compare the three representations we use algorithms that take comparable advantage of the different spatial decompositions. We have chosen a bottom-up approach with the data-parallel PMR quadtree, as the quadtree structure naturally lends itself to this style of implementation (i.e., non-overlapping regions with split axes in registration). In contrast, for the data-parallel R-tree and R^+-tree, a top-down algorithm (see [Brin93] for a similar algorithm in the sequential domain) is preferable, as it can take advantage of the spatial decomposition by determining early which nodes cannot possibly intersect. In the interest of providing a comparison we also implemented a bottom-up algorithm for the R-tree, as well as a very simple data-parallel algorithm that does not employ any spatial sorting and thus checks every line segment against every other line segment for satisfaction of the join condition.

Ideally, we would like to take advantage of the decomposition of the underlying space from which the lines are drawn and avoid comparing lines that cannot possibly intersect. This is quite easy when using the data-parallel bucket PMR quadtree as it provides a sort of the underlying space and a partition into disjoint blocks. Moreover, since the data-parallel bucket PMR quadtree is based on a regular decomposition, it is easy to identify blocks in the two maps that correspond to the same parts of the underlying space, thereby avoiding having to check for the intersection of lines that cannot possibly intersect. However, this is not possible for the data-parallel R-tree and the R^+-tree as they do not make use of regular decomposition. Furthermore, in the case of the data-parallel R-tree, the bounding rectangles are not disjoint. Thus the best we can hope for is to use the fact that some of the bounding rectangles do not intersect thereby avoiding the need to test their constituent bounding rectangles or line segments for possible intersection.

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 main utility of the R-tree and R^+ tree 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 those of 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.

The rest of this paper is organized as follows. Section 2 describes three useful models of parallel computation: PRAM, Scan, and SAM. Section 3 gives the construction algorithms for the data-parallel bucket PMR quadtree, R-tree, and R^+-tree. Section 4 contains a description of the polygonalization and spatial join algorithms for each of the three data-parallel structures. Section 5 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 6 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 a case where there is a potential for misunderstanding a claim.

2 Models of Parallel Computation

In this section we describe three models of parallel computation: PRAM, Scan, and SAM. In the process we elaborate on their suitability for operations on spatial data structures. As we will see, the scan model is the most appropriate. For the scan model we also describe the types of primitive operations, as they will be used in the description of the various algorithms.

2.1 PRAMs

An N processor Parallel Random Access Machine (or PRAM) consists of a collection of processors $P_1, P_2, \ldots, P_N$ and a global shared memory [Kuck77, Leig92]. Figure 1 contains a simple representation of an N processor PRAM. Each of the N processors can read or write from any location within the shared memory at each step of the computation.

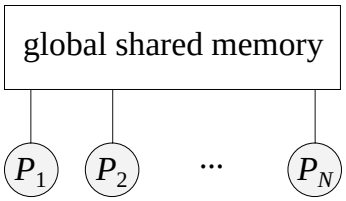


Figure 1: Simple figure representing N processors connected to a common shared memory in the PRAM model.

PRAMs are commonly classified according to concurrent access capabilities of the global shared memory. The most restrictive of the models is the exclusive-read, exclusive-write (EREW) PRAM. At any stage of the computation, only one processor is allowed to either read from or write to a specific memory location in the global shared memory. If we relax the exclusive read constraint and allow multiple processors to simultaneously read from a specific memory location, we obtain the concurrent-read, exclusive write (CREW) PRAM. Finally, if the exclusive write constraint is similarly relaxed, we obtain the concurrent-read, concurrent-write (CRCW) PRAM.

The PRAM model of parallel computation frees the user from the tedious details of actually implementing a parallel algorithm on a parallel machine. The programmer does not need to worry about the processor interconnection topology and communication conflicts. Unfortunately, large shared memory parallel computers are difficult to implement, and shared-nothing machines appear to be more scalable and well-suited to developing parallel machines with large numbers of processors [DeWi92]. It is possible, however, to emulate PRAMs on large shared-nothing machines (e.g., hypercubes [Leig92]) such as the CM-5, but with unknown performance penalties.

2.2 Scan Model

The scan model of parallel computation [Blel88, Blel89a] 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 [Schw80] takes an associative operator $\bigoplus$, a vector $[a_0, a_1, \ldots, a_{n-1}]$, and returns the vector $[a_0, (a_0 \bigoplus a_1), \ldots, (a_0 \bigoplus a_1 \bigoplus \cdots \bigoplus a_{n-1})]$. Such a scan is said to be in the upward direction. 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.

2.2.1 Scanwise Operations

In addition to being classified as either upward or downward, scan operations may be segmented. A segmented scan may be thought of as multiple parallel scans, where each operates independently on a segment of contiguous processors. Segment groups are commonly delimited by a segment bit, where a value of 1 denotes the first processor in the segment. For example, in Figure 2, there are four segment groups, corresponding to segments of size 3, 4, 2, and 3.

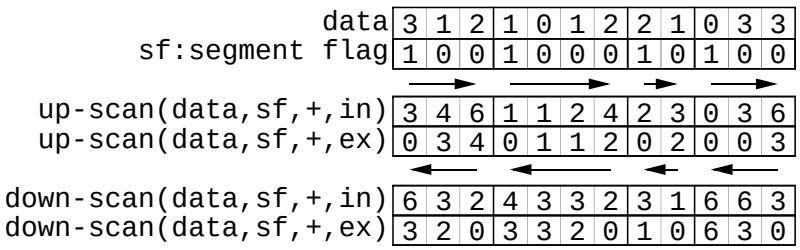


Figure 2: Example segmented scans for both the upward and downward directions (as well as inclusive and exclusive).

Finally, scan operations may be further classified as being either inclusive or exclusive. For example, an upward *inclusive* scan operation using the associative operator $\bigoplus$ returns the vector $[a_0, (a_0 \bigoplus a_1), \ldots, (a_0 \bigoplus a_1 \bigoplus \cdots \bigoplus a_{n-1})]$, while an upward exclusive scan returns the vector $[0, a_0, \ldots, (a_0 \bigoplus a_1 \bigoplus \cdots \bigoplus a_{n-2})]$. Various combinations of segmented scans, where $\bigoplus$ is bound to the addition operator, are shown in Figure 2.

2.2.2 Elementwise Operations

An *elementwise* primitive is an operation that takes two vectors of equal length and produces an answer vector, also of equal length. The i^{th} element in the answer vector is the result of the application of an arithmetic or logical primitive to the i^{th} elements of the input vectors. In Figure 3, an example elementwise addition operation is shown. `A` and `B` correspond to the two input vectors, and `ew(+,A,B)` denotes the answer vector.

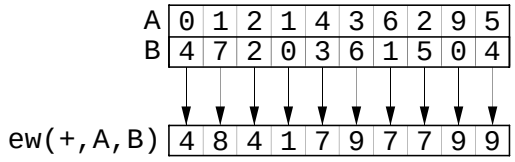


Figure 3: Example highlighting an elementwise addition operation.

2.2.3 Permutations

A *permutation* primitive takes two vectors, the data vector and an index vector, and rearranges (permutes) each element of the data vector to the position specified by the index vector. Note that the permutation must be one-to-one; two or more data elements may not share the same index vector value. Figure 4 provides an example permutation operation. `A` is the data vector, `index` is the index vector, and `permute(A,index)` denotes the answer vector.

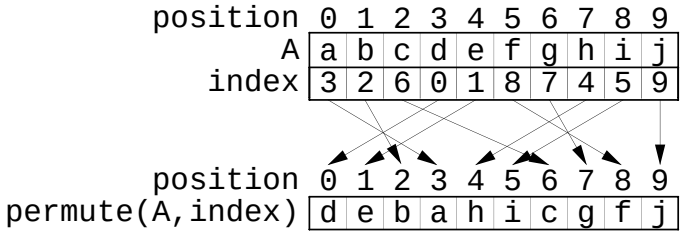


Figure 4: Example of a permutation.

2.3 SAM Model

A similar but more restrictive model of parallel computation, the SAM (Scan-And-Monotonic-mapping) model of parallel computation [Best92], may be defined by one or more linearly ordered sets of processors which allow element-wise and scan-wise operations to be performed. Instead of the permutation operation, the SAM model only allows the performance of *monotonic mappings*

both within and between each linearly ordered set of processors. A monotonic mapping is defined as one in which the destination processor indices are a monotonically increasing or monotonically decreasing function of the source processor indices. For example, consider the situation depicted in Figure 5 where the source processors are contained in processor set `A`, and the destination processors are located in processor set `B`. Figure 5a is a valid monotonic mapping, while the mapping in Figure 5b is not a monotonic mapping (as `f` comes before `c` in the linear ordering).

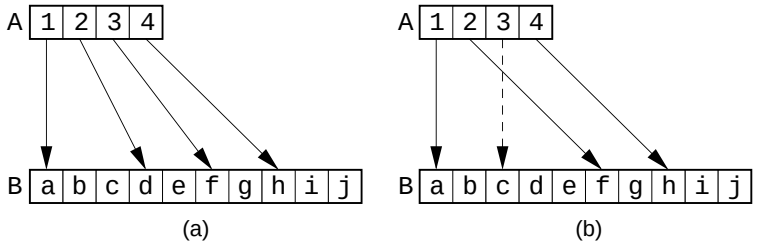


Figure 5: (a) An example monotonic mapping between two sets of processors, and (b) a similar mapping which is not monotonic.

Being more restrictive than the scan-model by requiring monotonic mappings, the SAM model also considers scan operations as taking unit time, thus allowing sorting operations to be performed in $O(\log n)$. The SAM model was deemed inappropriate for our research as it is unable to efficiently facilitate the manipulation of R-trees. This is due to the difficulties involved in maintaining monotonic mappings between two different R-trees when performing spatial queries such as map intersection (see [Hoel94a, Hoel94b] for more details). Note however that our algorithm for building data-parallel R-trees as described in Section 3.2 does not violate the more restrictive SAM model. Bucket PMR quadtrees, with their regular disjoint decompositions, are a structure for which the SAM model is well-suited.

Because of the bucket PMR quadtree's regular decomposition, a unique linear ordering may readily be obtained given a particular linear ordering methodology such as a Peano curve [Pean90]. As will be shown later, the R-tree, with its irregular decomposition, does not have a unique linear ordering. When performing operations on two maps with non-unique linear orderings, the maintenance of the monotonic mappings becomes expensive due to the necessary processor reorderings in the data-parallel environment. For example, consider the situation depicted in Figure 6 where two sets of processors (set (`A`,`B`) and set (`C`,`D`)) correspond to the overlapping regions in Figure 6a. Suppose each processor in one group must communicate with each intersecting processor in the other group (i.e., `A` with `C` and `D`, and `B` with `C` and `D`). For the first round of communication shown

in Figure 6b, a monotonic mapping may be maintained. The second round (depicted in Figure 6c) however violates the monotonic mapping. If processors `A` and `B` in the first set are reordered (an expensive operation for a large collection of processors), the monotonic mapping may once again be maintained as shown in Figure 6d.

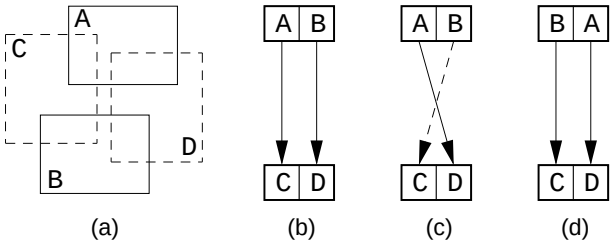


Figure 6: (a) An example collection of intersecting bounding boxes, (b) a valid monotonic mapping, (c) an invalid monotonic mapping, and (d) a valid monotonic mapping following processor reordering.

3 Data-Parallel Spatial Data Structures

We have selected three spatial data structures that represent fundamentally different approaches to storing spatial data for implementation in the data-parallel environment. The first structure, the bucket PMR quadtree [Nels86], employs a disjoint, regular decomposition. The second structure, the R-tree [Gutt84], utilizes a non-disjoint, irregular decomposition. Finally the third structure, the R^+-tree [Falo87], uses a disjoint, irregular decomposition. Below, we show how to adapt the three spatial data structures to a data-parallel environment. This results in data-parallel variants.

3.1 Data-Parallel Bucket PMR Quadtree

In the data-parallel environment, all lines are inserted simultaneously when constructing a spatial data structure. Thus there is no particular ordering of the data upon insertion. The conventional (i.e., sequential) PMR quadtree's node splitting rule is one that splits a node once and only once when a line is being inserted. This is the case even if the number of lines that result exceeds the node's capacity. Such a splitting rule is nondeterministic in the sense that the decomposition depends on the order in which the lines are inserted. For example, consider the situation depicted in Figure 7 where changing the insertion order of lines `3` and `4` results in different decompositions. This nondeterminism is unacceptable when many lines are inserted in a node simultaneously as we do not know how many times the node should be split. In order to avoid this situation, we chose

the bucket PMR quadtree for the data-parallel environment because its shape is independent of the order in which the lines are inserted and because of its well-behaved bucket splitting rule (i.e., there is no ambiguity with respect to how many subdivisions take place when several lines are inserted simultaneously).

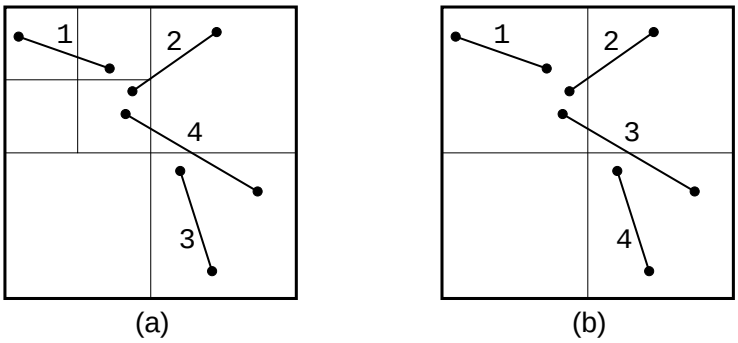


Figure 7: (a) An example sequential PMR quadtree with a splitting threshold of 2, with the lines inserted in numerical order, and (b) the resulting PMR quadtree when the insertion order is slightly modified so that line 4 is inserted before line 3.

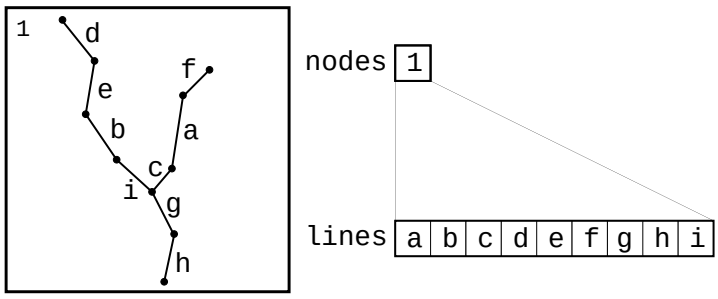


Figure 8: Initial data-parallel PMR quadtree processor assignments.

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 8 (with the example dataset, assume we have an 8×8 quadtree of maximal height 3). 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.

The splitting occurs in two stages. The regrouping is applied after each split and is achieved with an *unshuffle* operation [Best92] (where two intermixed types are rearranged into two disjoint groups termed *segments* via two monotonic mappings). The unshuffle is used to concentrate those line

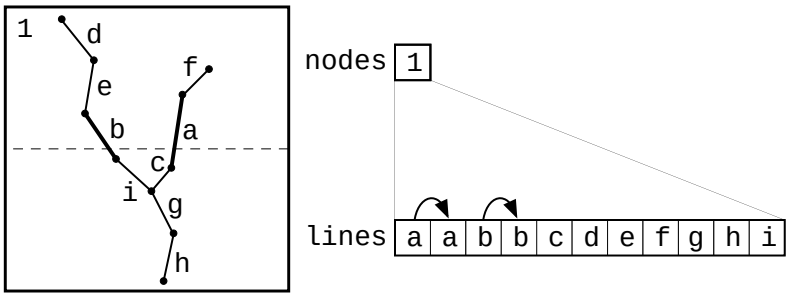


Figure 9: Cloning lines a and b due to their intersection with the vertical split.

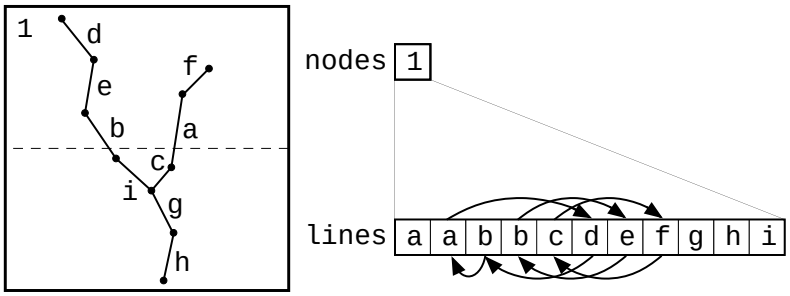


Figure 10: Application of the unshuffle operation to the y coordinate value of the center of the block associated with the node 1 processor.

processors together into two new segments, each of which corresponds to all of the line processors lying either in whole or in part above or below the y coordinate value of the center of the block associated with the node processor (regrouping without monotonic mappings has also been termed *packing* [Krus85] and *splitting* [Blel89a]). The unshuffle operation is depicted in Figure 10. 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* [Best92]). For example, consider lines a and b during the process of subdividing the first node in Figure 9. Because lines a and b intersect both the top and bottom halves of the root node, they are cloned.

The second stage first clones line i (depicted in Figure 11 with a heavier line), and then applies the unshuffle 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 to the left and right of the x coordinate value of the center of the block associated with the node processor. The application of this unshuffle operation is depicted in Figure 12.

Continuing with this iterative process, each line segment group determines the number of lines

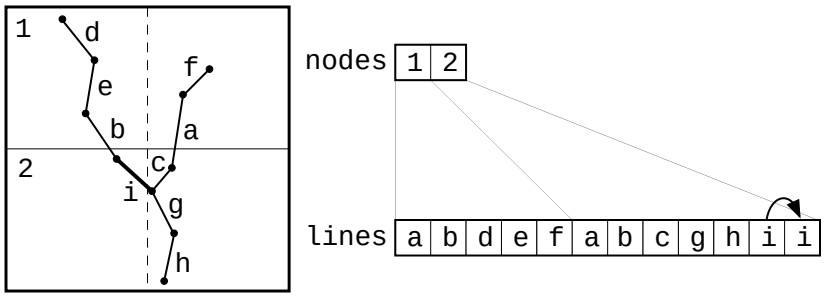


Figure 11: Cloning line i due to its intersection with the horizontal split.

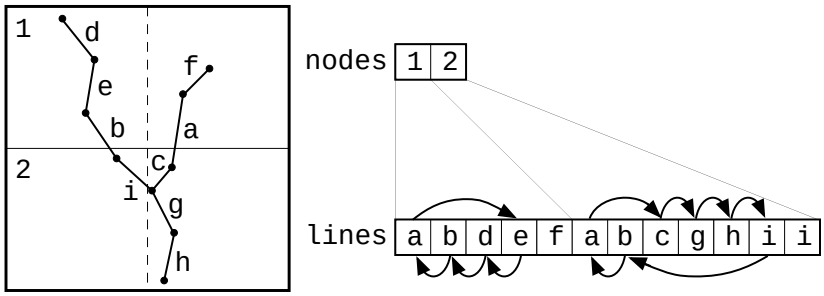


Figure 12: Application of the unshuffle operation to the x coordinate value of the center of the block associated with nodes 1 and 2.

it contains, and then communicates the count to the associated node processor. For example, in Figure 13, the first line segment group transmits a count of 3 to node 1, the second line segment group transmits a count of 2 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 13, nodes 1 and 4 will subdivide, resulting in the situation depicted in Figure 14.

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

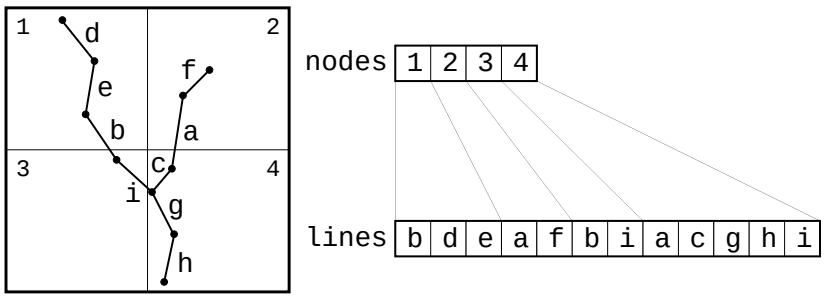


Figure 13: Result of the first node subdivision, line duplication, and un-shuffling.

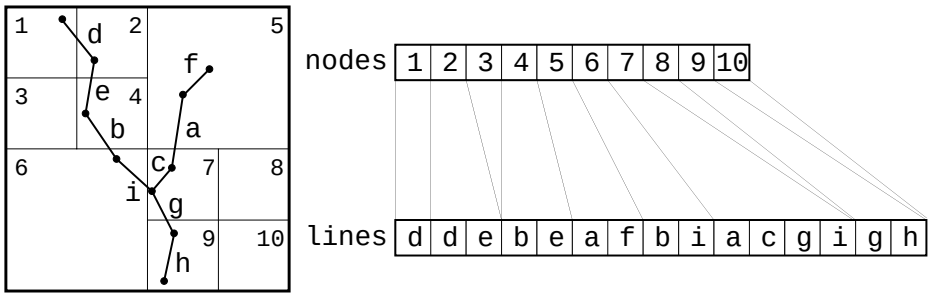


Figure 14: Result of the second round of node subdivisions.

has been reached (i.e., a node of size 1×1). This is not a problem, because for practical bucket capacities (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.

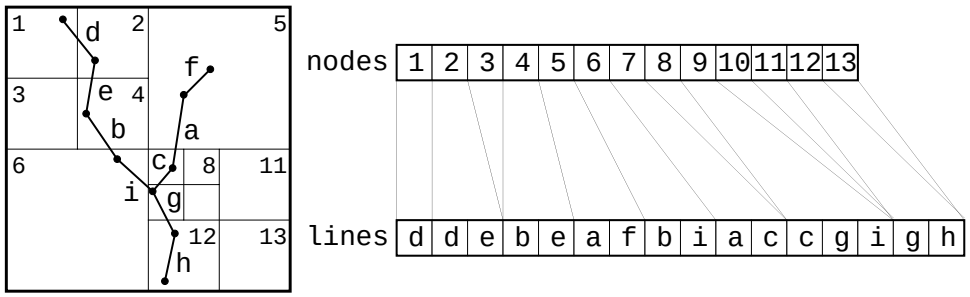


Figure 15: Result of the bucket PMR quadtree build process.

Because node 7's bucket capacity is exceeded (shown in Figure 14), and the maximal resolution has not yet been reached, another round of subdivision is necessary. The result of the third and final subdivision for our example data set is shown in Figure 15. Note that one of the quadtree nodes (node 9) still has its bucket capacity exceeded. In the example, the maximal resolution has been reached (i.e., 8×8). Therefore, node 9 will not be subdivided further. Given an $s \times s$ image, the data-parallel bucket PMR quadtree building operation takes in the worst-case $O(\log s)$ time, where each of the $O(\log s)$ subdivision stages requires $O(1)$ computations (a constant number of scans and unshuffles). In the average case where the n line segments are "roughly" uniformly distributed, the height of the tree and number of subdivision stages is $O(\log n)$ as when a node containing k lines is subdivided, it results in four children that contain approximately $k/4$ lines. The bucket size of the quadtree only affects the height of the resulting tree, with larger bucket sizes yielding shallower trees. The bucket size has no bearing on the number of computations performed during each subdivision step. Thus, in the average case, a data-parallel PMR quadtree containing

n line segments can be built in $O(\log n)$ time (i.e., $O(\log n)$ subdivision stages, each of complexity $O(1)$).

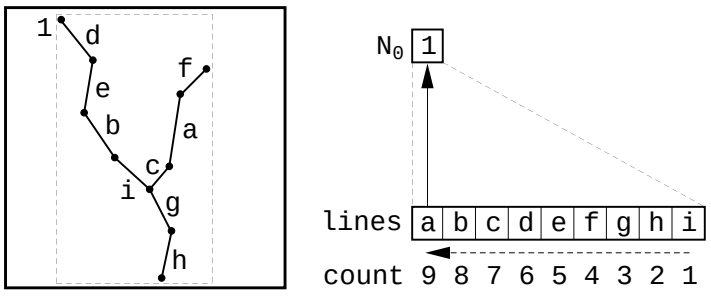


Figure 16: Initial processor assignments.

3.2 Data-Parallel R-trees

The data-parallel R-tree construction algorithm differs from the sequential algorithm in that, rather than inserting line segments sequentially into the data structure, all line segments are inserted simultaneously. The data-parallel R-tree construction algorithm proceeds as follows. Initially, one processor is assigned to each line of the data set, and one processor to the resultant data-parallel R-tree, as depicted for a sample dataset in Figure 16. Our example assumes an order $(1,3)$ R-tree. In the figure, the label `N 0` denotes the R-tree node processor set, with the associated square region containing the identifier of the R-tree node associated with the R-tree node processor. We use the term *segment* to refer to the collection of line processors associated with a particular R-tree node processor. Within the line processor set, the nine square regions contain the line identifiers. 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. This is shown in Figure 16 as the `count` field beneath the line processor set. The number of lines in the segment is then passed by the first line in the linear ordering to the single R-tree node processor (depicted in Figure 16 by the arrow from line `a` to node `1`). If the number of lines in the segment exceeds the node capacity M, then the data-parallel R-tree root node must be split into two leaf nodes and a root node (as is similarly done with the sequential R-tree). The two new leaf nodes are inserted into the R-tree node processor set, with the former root node/processor updated to reflect the two new children.

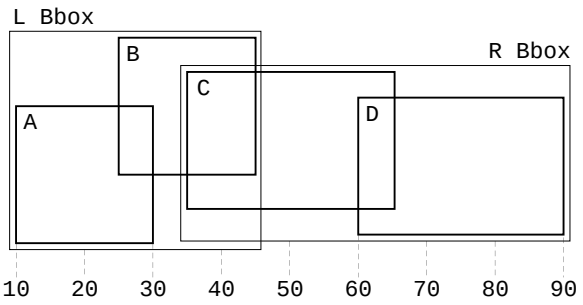
Node Splitting Algorithms

The topic of how to split an overflowing node has been the subject of much research on sequential R-trees. For example, the R*-tree [Beck90] is an R-tree variant that uses a more sophisticated node insertion and splitting algorithm than those provided with the conventional definition of the R-tree [Gutt84]. For the data-parallel R-tree, we have developed two node splitting algorithms, each of varying computational complexity.

In the first and simplest algorithm, the splitting axis (i.e., x or y-axis in the two-dimensional case) and the coordinate value are determined by finding the mean values along each axis of the midpoints of all bounding boxes in the line processor set in parallel via a sequence of scan operations. For each axis and segment group, the midpoints of the bounding boxes are first summed using a downward inclusive segmented scan operation (with the addition operator). The first node in the segment group then divides the sum by the number of bounding boxes in the segment group in order to obtain the mean value of the midpoints of the bounding boxes. This mean value is then used as the split coordinate value and is broadcast to all other nodes in the segment group with an upward segmented scan (using the copy operator). At this point, each node determines whether it lies in the left or right resulting bounding boxes by comparing the midpoint of its bounding box with the broadcasted split coordinate value. In the case of an x-axis split, if the node's bounding box midpoint is less than the broadcast split coordinate value, then the node's bounding box is assigned to the left resulting bounding box. If the node's bounding box midpoint is greater than the broadcast split coordinate value, then the bounding box is assigned to the right resulting bounding box. Finally, a small sequence of upward and downward inclusive scan operations (using either a min or max operator, depending upon the nature of the scan) is used to determine the physical extents of the two resulting bounding boxes.

The split axis and coordinate value are chosen from the two possible splits (i.e., the bounding box means along the x-axis and the y-axis) so as to minimize the amount of area common to the two resulting nodes. Next, the first node in each segment group, which contains the physical extents of the two resulting bounding boxes along each axis, selects the best split. A final upward scan operation communicates the selection among each node in the segment group. This operation is of complexity $O(1)$ at each stage of the building operation, as a constant number of scans dominates the computation.

An example illustrating this node splitting algorithm is shown in Figure 17. Consider the four



	A	B	C	D	
ls: left side	10	25	35	60	
rs: right side	30	45	65	90	
m: midpoint	20	35	50	75	
					scan type
ms: midpoint sum	180	160	125	75	down-scan(mp,,+,in)
mm:midpoint mean	45	45	45	45	up-scan(ms,,copy,in)
lr: L/R side	L	L	R	R	if (m < mm) then L else R
L Bbox left side	10	25	-	-	L:down-scan(ls,,min,in)
L Bbox right side	45	45	-	-	L:down-scan(rs,,max,in)
R Bbox left side	-	-	35	35	R:up-scan(ls,,min,in)
R Bbox right side	-	-	65	90	R:up-scan(rs,,max,in)

Figure 17: Example highlighting the various scan types and their application to determining the x-coordinate values for the left and right bounding box.

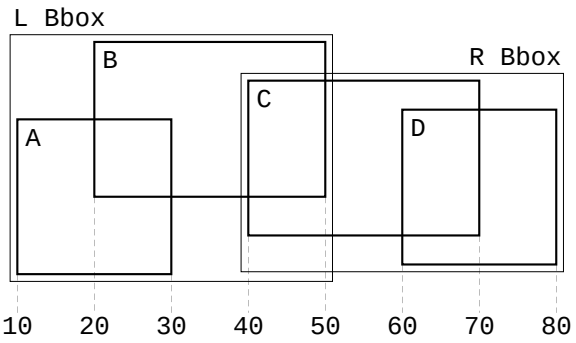
bounding boxes labeled `A`–`D`, which are assumed to comprise the same segment group. In this example, which depicts the selection of the x axis split coordinate value, we are only considering the x-coordinate values of the bounding boxes. In Figure 17, the coordinate values of the left and right sides of the four nodes are indicated on the lines labeled `ls:left side` and `rs:right side`, respectively. For example, node `C` has left and right x-coordinate values `35` and `65`, respectively. To begin the node splitting process, each node in parallel determines the x axis midpoint value of its bounding box. The values are found in the figure on the line labeled `m:midpoint`. Once all midpoints are computed, a downward inclusive scan operation using the addition operator determines the sum of the midpoint values; this is depicted in line `ms:midpoint sum`. The first node in the segment group then determines the midpoint mean by dividing the midpoint sum by the number of nodes in the segment group. This midpoint mean is then broadcast with all other nodes in the segment via an upward inclusive scan operation using the copy operator (shown in the line labeled `mm:midpoint mean`). The midpoint mean functions as the x axis candidate split coordinate value. Each node then in parallel determines whether its midpoint lies to the left or right of this midpoint mean value. If a node's midpoint value is less than the midpoint mean, it

will be grouped with all nodes to the left of the midpoint mean; otherwise, it will be grouped with nodes falling to the right of the midpoint mean.

A sequence of scan operations can be used to determine the resulting left and right bounding boxes. Initially, the bounding box left and right x coordinate values will only be found in the first and last nodes of the segment group (i.e., nodes `A` and `D`, respectively). Of course, these values may later be broadcast to all nodes in the segment group via a sequence of four scan operations using a copy operator. As shown in Figure 17, a downward minimum inclusive scan on the left x coordinate value for all nodes to be grouped in the left bounding box is used to determine the left x-coordinate value for the left bounding box (`L Bbox left side`). Similarly, a downward maximum inclusive scan on the right x-coordinate values for nodes grouped on the left will establish the right x-coordinate value for the resulting left bounding box (`L Bbox right side`). Thus, for the resulting left bounding box, the left and right x-coordinate values are `10` and `45`, respectively. These values are found in the first node (node `A`) on rows of Figure 17 labeled `L Bbox left side` and `L Bbox right side`. Analogous upward min/max exclusive scans are used to determine the left and right x-coordinate values of the resulting right bounding box. We observe that the left and right x-coordinate values for the resulting right bounding box are `35` and `90`, respectively. These values are found in the last node of the segment group (node `D`) on lines `R Bbox left side` and `R Bbox right side`, respectively.

The second node splitting algorithm first sorts all lines in the segment according to the left edge of their bounding boxes. A sequence of upward scan operations are used to determine the extents of the bounding box formed by all lines preceding a line in the sorted segment. A similar sequence of downward scans will determine the bounding box for all following lines in the segment. For all *legal splits* (i.e., where each of the two resulting nodes receives at least m/M of the lines being redistributed), the amount of bounding box overlap is calculated, with the split corresponding to the minimal amount of overlap being selected as the x-axis candidate split coordinate value. An analogous procedure is employed for the y-axis in obtaining the y-axis candidate split coordinate value. Once the two candidate split coordinate values are determined, the one corresponding to the minimal bounding box overlap is selected. In the event of a tie, some other metric such as choosing the split with the minimal bounding box perimeter lengths may be employed.

Given n lines, this node splitting algorithm takes $O(\log n)$ time in the worst-case, as we employ two $O(\log n)$ sorts (in the scan model of computation, one can perform a radix sort of n items using $O(\log n)$ scans) and a constant number of upward and downward scan operations.



	A	B	C	D	
ls: left side	10	20	40	60	
rs: right side	30	50	70	80	
					scan type
L Bbox left side	10	10	10	10	up-scan(ls,-,min,in)
L Bbox right side	30	50	70	80	up-scan(rs,-,max,in)
R Bbox left side	20	40	60	0	down-scan(ls,-,min,ex)
R Bbox right side	80	80	80	0	down-scan(rs,-,max,ex)

Figure 18: Example highlighting the various scan types and their application to determining the x-coordinate values for the left and right bounding boxes.

Below we illustrate how to calculate the legal splits. Consider the example shown in Figure 18 consisting of four bounding boxes labeled `A`–`D` where the nodes have been sorted according to their left x-coordinate values. In this example, we are only considering the x-coordinate values of the bounding boxes, though incorporation of y-coordinate values is straightforward. In the figure, the left and right coordinate values of the four nodes are indicated on the lines labeled `ls:left side` and `rs:right side`, respectively. For example, node `B` has left and right x-coordinate values `20` and `50`, respectively; while node `C` has left and right x-coordinate values `40` and `70`, respectively. Assuming that a node is grouped with all nodes on its left when forming the bounding boxes (i.e., node `C` is grouped with nodes `A` and `B` when forming node `C`'s left and right bounding boxes), the following sequence of scan operations can be used to determine the bounding boxes on the left and the right for each node. In particular, as shown in Figure 18, an upward minimum inclusive scan on the left coordinate value is used to determine the left x-coordinate value for the bounding box on a node's left side (`L Bbox left side`). Similarly, an upward maximum inclusive scan on the right x-coordinate values will establish the right x-coordinate value for the bounding box on a node's left side (`L Bbox right side`). Thus, for node `B`, we see that the left and right x-coordinate values for the bounding box to its left (i.e., the bounding box containing nodes `A` and `B`, labeled `L Bbox` in Figure 18) are `10` and `50`, respectively. These values are found in the rows of Figure 18 labeled

`L Bbox left side` and `L Bbox right side`. Analogous downward min/max exclusive scans are used to determine the left and right x-coordinate values of the bounding box to the right of each node. We observe that the left and right x-coordinate values for the bounding box to the right of node `B` (i.e., a bounding box containing nodes `C` and `D`, labeled `R Bbox left side` and `R Bbox right side`, respectively in Figure 18) are `40` and `80`, respectively.

data	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t
segment flag	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
position	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10
elements	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
legal split	N	N	N	Y	Y	Y	N	N	N	N	N	N	N	Y	Y	Y	N	N	N	N

Figure 19: Example legal splits where the minimal occupancy level is 40%.

When selecting the locally optimal node split with the R-tree, it is necessary to ensure that the chosen split results in each node receiving at least m/M of the lines being redistributed. All candidate splits that satisfy this criterion are termed *legal splits*. For example, given a minimum occupancy level of 40%, the right edges of lines `d`–`f` and `n`–`p` in Figure 19 are legal split positions). The legal splits may be determined by first enumerating the elements in the segment group with a single upward inclusive scan operation (denoted by `position` in Figure 19). A downward inclusive scan operation is then used to copy the number of elements in the segment group across the segment group (denoted by the `elements` field in the figure). Finally, each element determines whether or not it corresponds to a legal split by dividing its enumerated position (`position`) by the number of elements in the segment group (`elements`). If the result is between m/M and $(1-m)/M$, then it corresponds to a legal split. In the example, the result of this determination is shown in the `legal split` field.

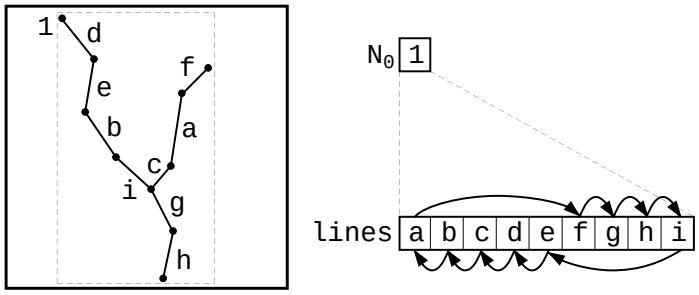


Figure 20: Unshuffle operation.

Once the splitting axis and the coordinate value are chosen, an unshuffle operation is used to concentrate those line processors together into two new segments, each of which will correspond to

one of the two R-tree leaf node processors as depicted in Figure 20. For example, all lines which have a midpoint that is less than the split coordinate value are monotonically shifted toward the left, while those whose midpoint is greater than the split coordinate value are monotonically shifted toward the right among the line processors. The result of the unshuffle operation on the lines in Figure 20 is shown in Figure 21. Note that the root node of the data-parallel R-tree is associated with two segments in the line processor set `A` (i.e., (`a,b,e,h`) and (`c,d,f,g,i`)), and must itself be subdivided in an analogous manner.

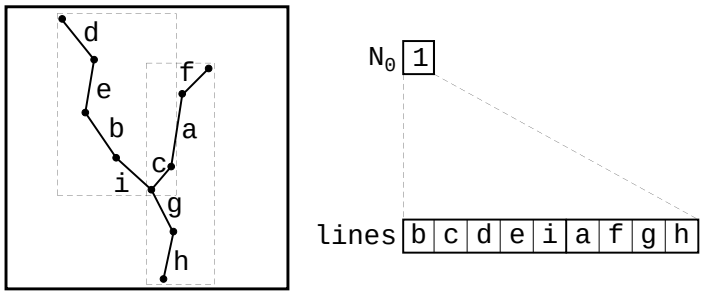


Figure 21: Result of the unshuffle operation.

Thus, at this stage after the first root node split and line redistribution, we will wind up with two segments in the line processor set, and two different R-tree processor sets $\mathtt{N}_0$ and $\mathtt{N}_1$ (each set corresponding to a node at a different height in the data-parallel R-tree), as shown in Figure 22.

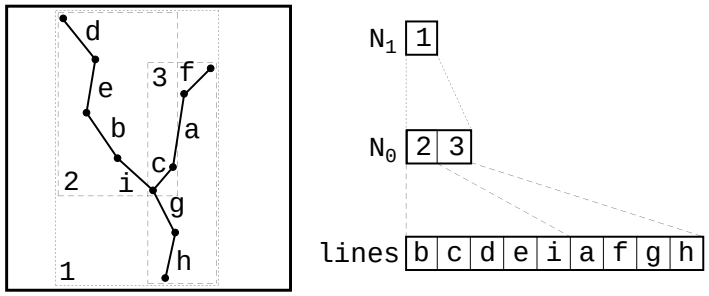


Figure 22: Completion of root node split operation.

The building algorithm will now proceed iteratively, with each segment in the collection of line processors 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 segment exceeds the node capacity M, then the segment (and corresponding R-tree node processor) will be forced to subdivide. Note that this subdivision process may result in processors that correspond to internal nodes in the data-parallel R-tree splitting themselves (with these splits possibly propagating upward through the data-parallel R-tree).

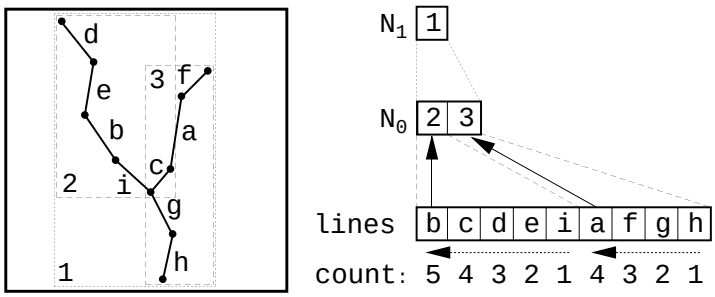


Figure 23: Broadcasting the line counts to the associated nodes.

The building process terminates when all nodes in the R-tree node processor set have at most M child processors (either internal R-tree nodes or line processors) as shown in Figure 23 for our example dataset. The data-parallel R-tree root node corresponds to the single processor in set $\mathtt{N}_2$, the leaf nodes are contained in processor set $\mathtt{N}_0$, and all lines are grouped in segments of length less than or equal to 3 in the line processor set (recall that we are dealing with an order $(1,3)$ R-tree in our example).

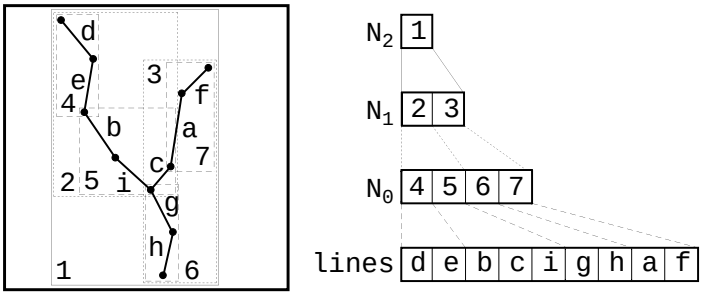


Figure 24: Completion of the data-parallel R-tree building operation.

Given n lines, the data-parallel R-tree is of worst-case maximal tree height $O(\log n)$ which is obtained by observing that each R-tree node contains at least two elements. In practice, each node of the R-tree contains considerably more than two elements; thus the tree height is much smaller. Assuming a maximal tree height $O(\log n)$, the data-parallel R-tree construction operation takes $O(\log^2 n)$ time, where each of the $O(\log n)$ subdivision stages requires $O(\log n)$ computations (a constant number of scans along with two bounding box sorts).

3.3 Data-parallel R$^+$-trees

The R$^+$-tree construction algorithm is similar to that of the R-tree with a few modifications. In order to facilitate the R$^+$-tree's disjoint decomposition, the method for handling splitting nodes must be modified. The R$^+$-tree node splitting algorithm first sorts all lines in the node according

to the left edges of their bounding boxes as is similarly done for the R-tree. Note that this splitting process is described for selecting a possible x-axis split; an analogous procedure will be followed for selecting a possible y-axis split. Next, for each node split whose result satisfies a pre-established *minimal occupancy level* of m/M lines in the two resulting nodes (termed a legal split in Section 3.2), the coordinate value of the left edge is broadcast to each of the lines in the node being split.

In an iterative process which depends on the number of legal splits in a node, each node that corresponds to a legal split in turn broadcasts the coordinate value of its right side. 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. For example, in the case of an x-axis split, a line can lie in either the node which is comprised of all lines to the left of the split coordinate value, or the node which consists of all lines to the right of the split coordinate value. 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 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 unshuffle 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 building algorithm proceeds iteratively, 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 the node capacity M, then the node (and corresponding R^+-tree node processor) are split. Similar to the R-tree building algorithm, the R^+-tree construction process terminates when all nodes in the R^+-tree node processor set have at most M children as shown in Figure 26 for our example dataset.

Note that the leaf node subdivision process may result in processors that correspond to internal

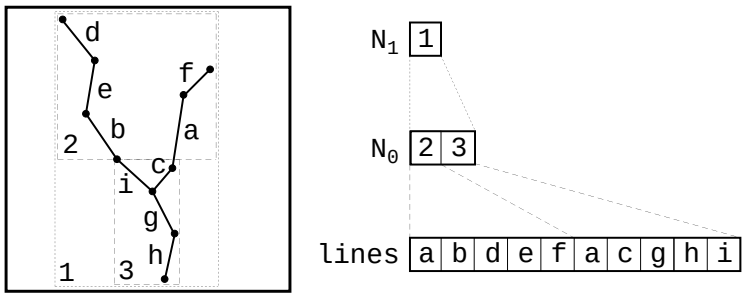


Figure 25: Result of the first round of R^+-tree node splitting for the data in Figure 8.

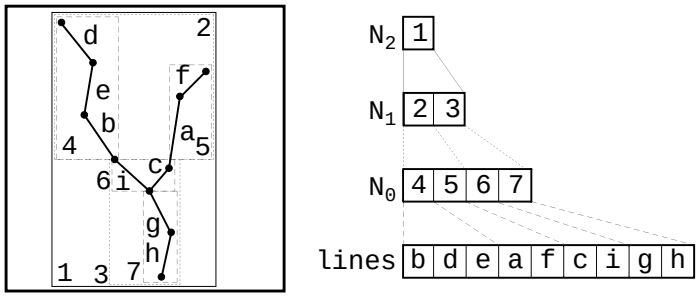


Figure 26: Completion of the data-parallel R^+-tree building operation.

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 termed *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 node being split. 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 fact that the decomposition induced by the R^+-tree must be disjoint requires that any intersecting descendents (nodes or lines) must also be split. Splitting the descendents of a node is termed a *downward split*. The R^+-tree construction process terminates when all nodes in the node processor set have at most M child processors (either internal R^+-tree nodes or line processors).

It is important to note that unlike the R-tree or R*-tree, the R^+-tree does not have a minimal node occupancy level m. This is a direct result of the downward node splits. When a downward node split is chosen, the resulting node occupancy levels of any descendent nodes that may be affected is not considered. Thus, a downward node split may result in a descendent node being split into two subnodes of very unbalanced capacity (i.e., a node with 20 elements could be split

into two nodes, the first containing two elements, and the second containing 18 elements).

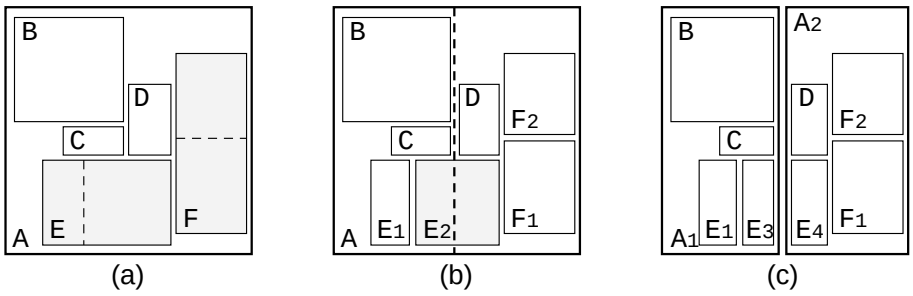


Figure 27: Example of upward and downward splits; (a) shows nodes E and F splitting, (b) shows the result of the two node splits and the resulting upward split of node A, and (c) depicts the result of the upward split of A.

For example, consider the situation depicted in Figure 27 where node A is the parent of nodes B, C, D, E, and F. Assume that the node capacity is 5, and that nodes E and F in Figure 27a are being split. The result of their subdivision into nodes E_1, E_2, F_1, and F_2 is shown in Figure 27b. Because parent node A now has seven children, the splitting of nodes E and F results in an upward split of A. Assuming that node A splits along the dashed line shown in Figure 27b, a downward split of child node E_2 will be necessary in order to maintain the disjoint decomposition. The result of parent node A's subdivision into nodes A_1 and A_2, and the downward split of node E_2 is shown in Figure 27c.

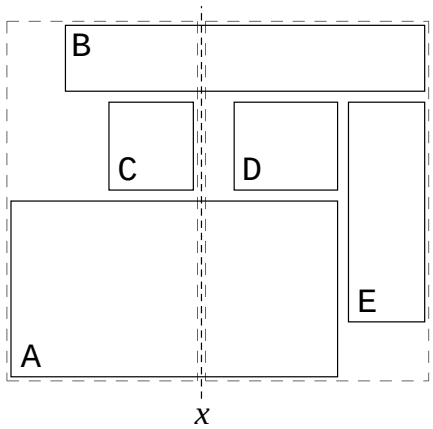


Figure 28: Example highlighting the left and right bounding boxes which correspond to the split coordinate value x associated with bounding box C during the R^+-tree node splitting process.

In comparison to the the data-parallel R-tree, the data-parallel R^+-tree construction process suffers with respect to the number of scan operations necessary to choose the locally optimal node

split. As was described in Section 3.2 (and depicted in Figure 18), a sequence of approximately ten scans is all that is necessary to select the node split axis and coordinate for a data-parallel R-tree. For the R^+-tree and its disjoint decomposition of space, the process requires significantly more scan operations. Due to the disjoint decomposition, it is sometimes necessary to split the children of internal nodes (i.e., downward splits) during the subdivision process as shown in Figure 27. For this reason, it becomes necessary to make use of additional knowledge of the contents of the children of a splitting node (e.g., the spatial extents of the grandchildren) when splitting the node.

Consider the situation shown in Figure 28 where nodes A–E represent five siblings in a node which is being split. Assume that x represents the chosen x-axis split coordinate value. When regrouping nodes A–E, node C is grouped in its entirety with the portions of nodes A and B that fall to the left of the split coordinate value x. Similarly, nodes D and E are grouped with the portions of nodes A and B that fall to the right of the split coordinate value x. If we were to apply a similar sequence of scan operations as was done with the data-parallel R-tree, then each node could independently (and in parallel) determine the two associated bounding boxes that correspond to all portions of nodes to the left of the split coordinate value x (i.e., the left halves of nodes A and B, and all of node C), and all portions of nodes to the right of the split coordinate value x (i.e., the right halves of nodes A and B, and all of nodes D and E).

For example, in Figure 28, the left bounding box corresponding to the split associated with node C consists of the left portions of nodes A and B, as well as node C in its entirety. Similarly, the right bounding box consists of the right portions of nodes A and B, and nodes D and E in their entirety. Note that this approach ignores the contents of all nodes (i.e., nodes A–E) when determining the left and right bounding boxes corresponding to a given split axis and coordinate value.

However, employing the R-tree-like approach to forming the bounding boxes for an R+-tree is flawed. Because of the disjoint decomposition of space and the standard requirement that all bounding boxes inside either the R-tree or the R+-tree be minimal bounding boxes, it sometimes becomes necessary to consider the contents of the nodes (i.e., nodes A–E in the example) when determining the left and right bounding boxes that would result from a candidate split. Consider the situation depicted in Figure 29a where node A contains *only* two children, labeled a_1 and a_2. Similarly, let node B also contain two children, labeled b_1 and b_2. For the purpose of this discussion, it is not necessary to show any children of nodes C–E.

If one were to use the R-tree-like approach (as detailed in Section 3.2), which ignores the distribution of the children nodes when determining a node split, then the left and right bounding

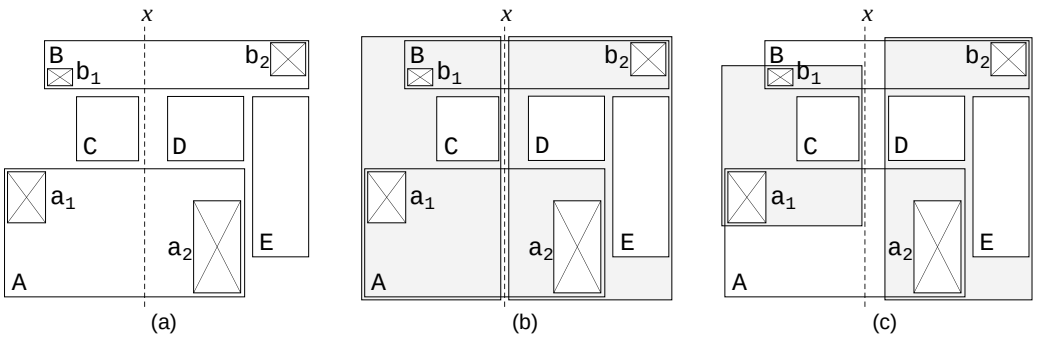


Figure 29: (a) Example from Figure 28 with the assumption that nodes A and B only have two children (depicted as crossed boxes), (b) the node split which results from using the standard R-tree-like approach, and (c) the correct R+-tree node split which considers the contents of the nodes (i.e., the crossed boxes).

boxes would correspond to the two shaded regions in Figure 29b. Given that nodes A and B only contain two children as described above, the correct bounding boxes that would result from the x-axis split using split coordinate x are quite different. For the example dataset in Figure 29a, the correct left bounding box corresponding to split coordinate value x contains the region formed by the portions of nodes A and B that lie to the left of the split coordinate (i.e., regions a 1 and b 1), as well as node C in its entirety. The correct bounding box containing a 1, b 1, and C is shown as the left shaded region in Figure 29c. Notice that this shaded region is considerably smaller than the left bounding box formed using the R-tree-like technique in Figure 29b. Similarly, the actual right bounding box that corresponds to split coordinate value x is shown as the right shaded region in Figure 29c. As we cannot accurately determine the left and right bounding boxes corresponding to each candidate split using the simple but fast R-tree-like method, it becomes necessary to employ a much more computationally expensive technique.

From this example, it is clear why the simple R-tree node splitting technique that uses a small sequence of scan operations is not applicable to the R^+-tree, as knowledge of the contents of the nodes being regrouped and possibly split must also be considered when determining the locally optimal node split. Interestingly, if one were to employ the R-tree-like node splitting technique along with the disjoint decomposition of space requirement, the ensuing construction algorithm would build a data-parallel k-d-B tree [Robi81].

4 Data-Parallel Spatial Queries

Parallel spatial queries were examined for the three implemented data-parallel spatial data structures in order to further understand their behavioral tradeoffs and differences in the data-parallel environment. We chose to implement three different spatial queries for each data structure. The first spatial query was polygonization. Polygonization is the process of determining all closed polygons formed by a collection of planar line segments. We 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 using a spatial data structure. Basically, the n lines could be sorted in parallel based upon their identifier in $O(\log n)$ time, then each line in sorted sequence would transmit 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.

The second and third queries fall into the classification of spatial joins. In particular, we look at map intersection and a spatial range query (e.g., find all lines in one map that are within distance d of any line in a second map). Each of these queries could similarly be solved by having all the lines in one map broadcast their endpoint coordinates to the lines in a second map in an $O(n)$ process. As is the case with polygonization, data-parallel spatial data structures can be used to reduce the number of scan operations (again by employing segmented scans), thus speeding the computations.

4.1 Data-Parallel Bucket PMR Quadtree Spatial Queries

As the bucket PMR quadtree has q-edges, spatial query algorithms for both sequential and data-parallel bucket PMR quadtrees are required to take into consideration the complexities introduced by the existence of q-edges (i.e., multiple feature representation). As was previously mentioned, q-edges are a by product of the disjoint decomposition imposed by the bucket PMR quadtree (this is also the case with R^+-trees, another disjoint decomposition). The existence of q-edges requires additional computation (i.e., duplicate deletion) when merging sibling nodes during the execution of some of the algorithms.

4.1.1 Polygonization

Given a bucket PMR quadtree, the polygonization process begins by constructing a partial winged-edge representation [Baum72] (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 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 $[a_i, a_i, \ldots, a_i]$). Locally, each line processor maintains the minimal and maximal angles formed at each endpoint of the corresponding line as well as the identities of the lines formed by these angles. 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).

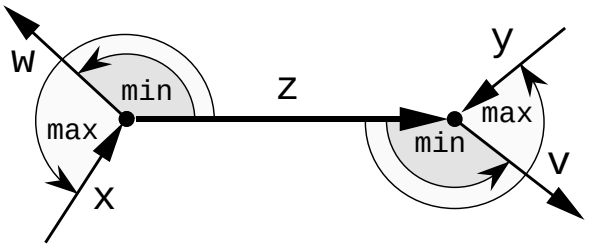


Figure 30: Selecting the initial polygon identifiers.

In Figure 30, 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 identity line identifier (z $_{R}$). In Figure 30, line z is assigned w $_{R}$ as the initial left polygon identifier, and v $_{R}$ as the right polygon identifier. Figure 31 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.

Starting at the leaf level, sibling nodes are then merged together into their parent nodes (i.e., in Figure 31, leaf nodes 4–7 are merged together, resulting in leaf node 4 in Figure 32). All the lines in the merged sibling leaf nodes are sorted, and any duplicate lines are marked. In Figure 31, the merging of sibling leaf nodes 4–7 will result in one pair of duplicate lines (line b) as line b occurs

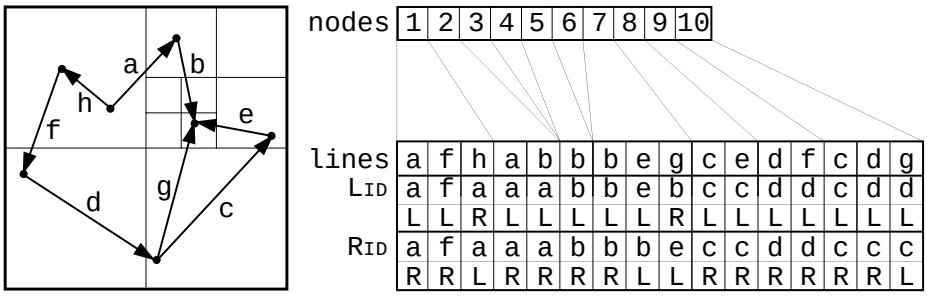


Figure 31: Initial polygon assignments.

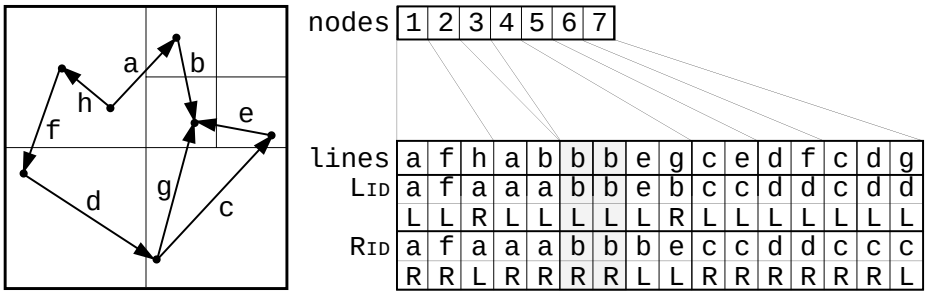


Figure 32: Result after first round of merging and prior to duplicate deletion (duplicate instances of lines are shown shaded).

in both nodes 5 and 7. The duplicate instances of line b are highlighted by the use of shading in Figure 32 which shows the results of merging nodes 4–7 prior to duplicate deletion.

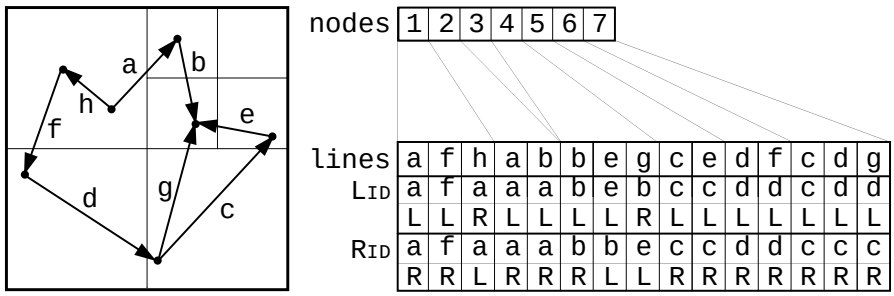


Figure 33: Polygon assignments after the first round of leaf node merging and duplicate line deletion.

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 33 the merging of sibling leaf nodes 2–5 will result in two pairs of duplicate lines (i.e.,

lines b and e) as shown in Figure 34. 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.

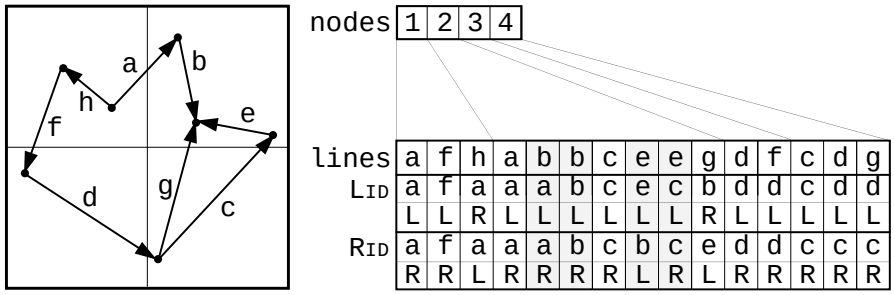


Figure 34: Result of the second round of merging prior to duplicate deletion (duplicate instances of lines are shown shaded).

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), then 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 value was previously updated to a_L during line b update broadcasts.

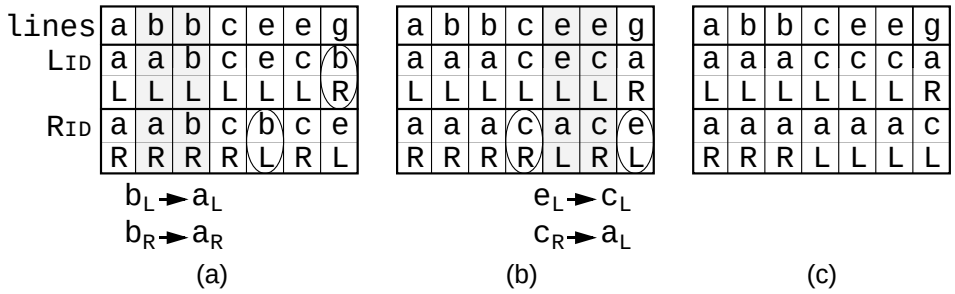


Figure 35: (a) Example of duplicate line b polygon identifier updating during the polygonization merging process for the situation also depicted in Figure 34, (b) the updating of duplicate line e's polygon identifiers, and (c) the result of updating all duplicate polygon identifiers.

For example, consider the situation depicted in Figure 35 which is taken from the merging nodes shown in Figure 34. The duplicate line b's result in two polygon updates (i.e., b_L to a_L and b_R

to a $_R$) being broadcast to the other lines in the segment group. The other lines whose left or right polygon identifiers are then updated are shown as circled items in Figure 35a (i.e., the right polygon identifier of the first line e, and the left polygon identifier of line g). Additionally, duplicate line e also results in two broadcasted polygon updates (e $_L$ to c $_L$ and c $_R$ to a $_L$ in Figure 35b). The final result of the broadcasting of polygon identifiers necessitated by the duplicate lines is shown in Figure 35c.

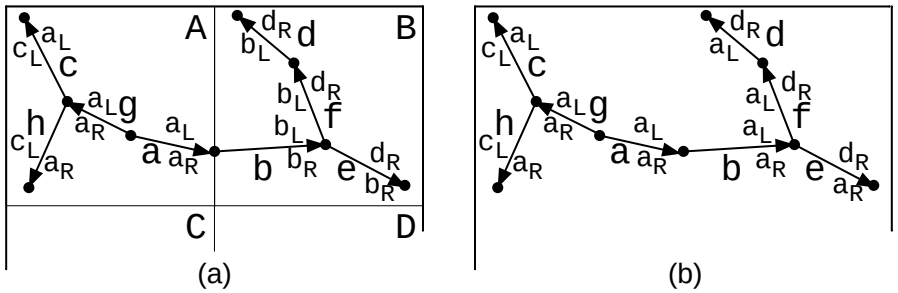


Figure 36: (a) Example of two leaf nodes A and B merging (the contents of sibling nodes C and D are not shown), and (b) the result of the merge operation.

Finally, when merging four sibling nodes together, any line whose endpoint falls on the shared node border (e.g., lines a and b in Figure 36a) must also have its endpoints and polygon identifiers broadcast among the merged nodes. Consider the example in Figure 36a 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 36b shows the resulting polygon identifiers.

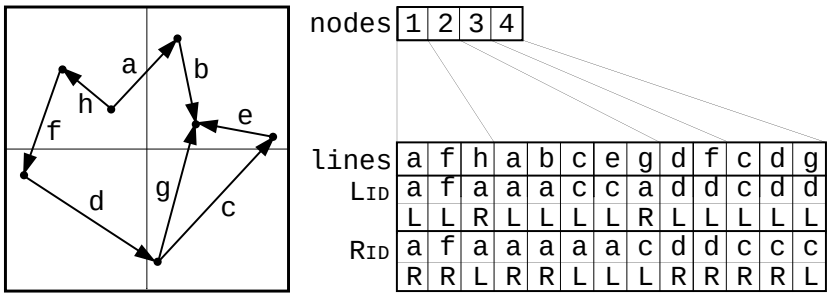


Figure 37: Polygon assignments after the second round of leaf node merging.

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 37–39, with the final assigned polygon identifiers circled).

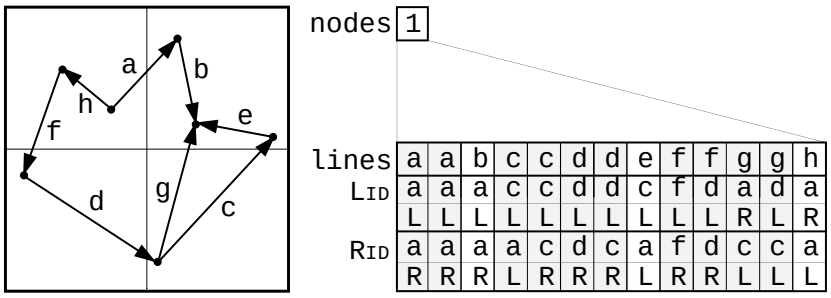


Figure 38: Result of final round of merging, prior to duplicate deletion (duplicate instances of lines are shown shaded).

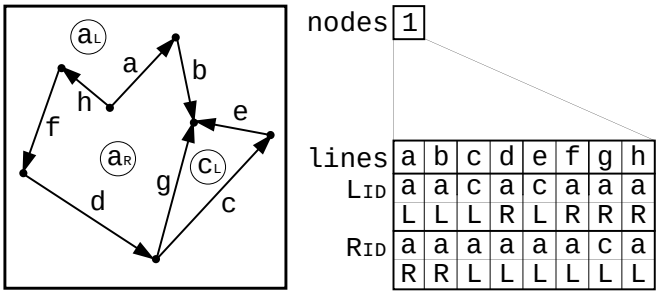


Figure 39: Result of the polygonization operation.

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.

In the worst case, where all of the n line segments in the PMR quadtree intersect the first split axis in either the x or y dimension, no lines may be removed from consideration during the node merging process. Thus, in the worst case, the polygonization process will be of complexity $O(n \log n)$.

4.1.2 Map Intersection

In the following algorithm description, assume that we are starting with two data-parallel bucket PMR quadtrees; one termed the *source* quadtree, and the second termed the *target* quadtree. The quadtrees are of equal size (i.e., they represent the same $s \times s$ area). The source quadtree will contain the reference set of lines to intersect against (e.g., the border of the city), and the target

quadtree contains the lines which will be determined to intersect the objects in the source quadtree (e.g., the roads found in the county).

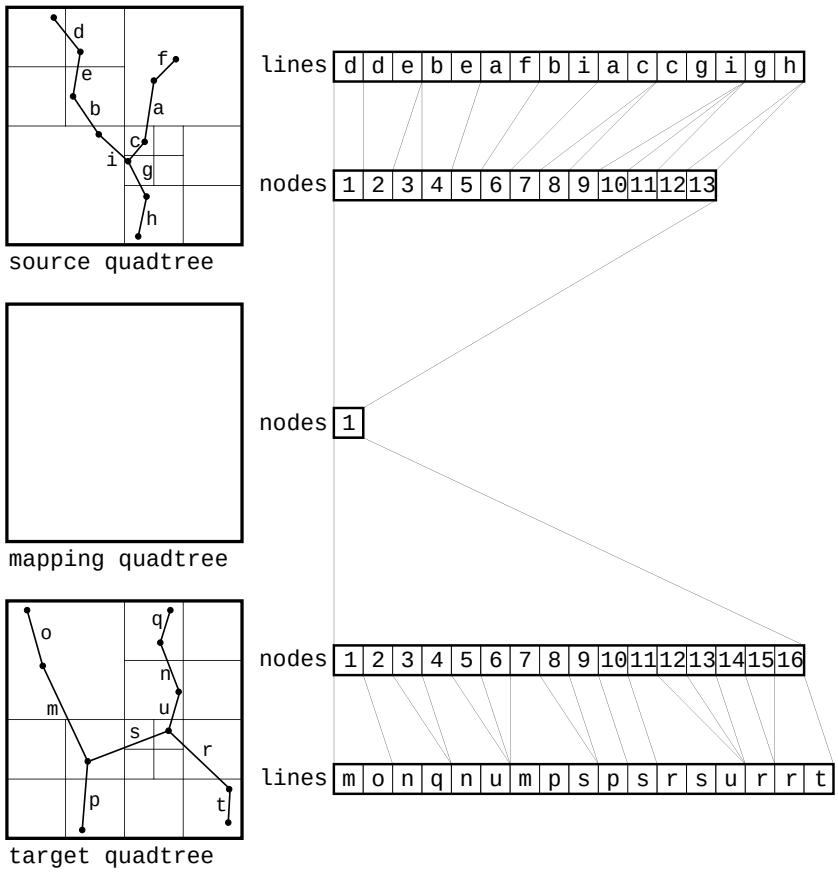


Figure 40: Example of a source and target data-parallel bucket PMR quadtree (each with a bucket capacity of 2) and a mapping quadtree prior to the first mapping node subdivision.

Given the data-parallel source and target bucket PMR quadtrees, we first establish a correspondence between the source and target quadtree nodes. This will facilitate the lessening of communication between the two quadtrees when performing the actual intersection. While establishing the source and target node correspondence, a third temporary set of quadtree nodes, termed the *mapping quadtree*, is employed. The mapping quadtree is discarded following completion of the operation.

The mapping quadtree initially consists of a single large node, equal in physical size to the exterior dimension of the source and target quadtrees (i.e., $s \times s$). The single mapping node is associated with the entire collection of both source and target quadtree leaf nodes (as an example, consider the situation depicted in Figure 40). The mapping quadtree nodes (of which there is

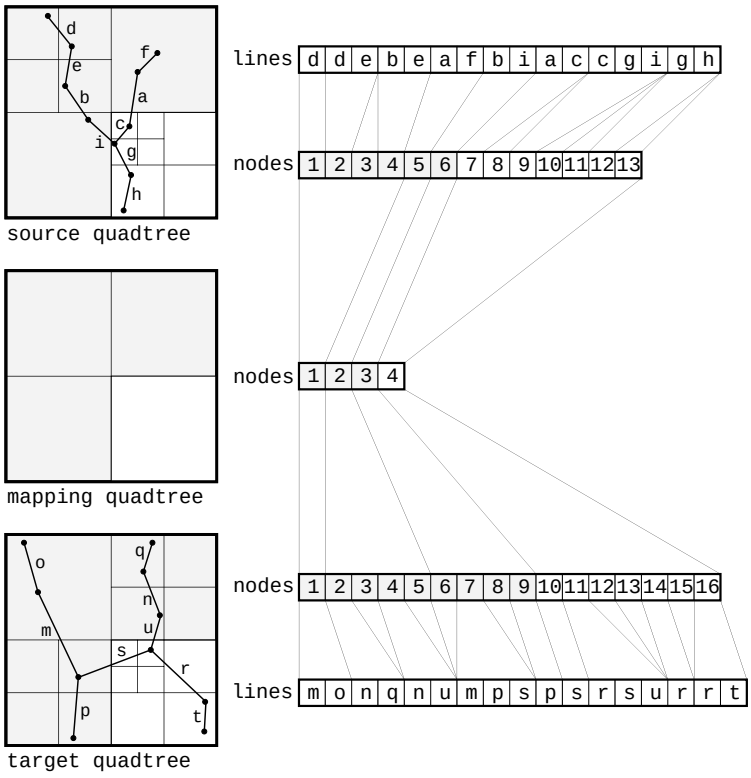


Figure 41: Mapping quadtree nodes at the completion of the first subdivision phase with completed mappings shaded.

initially only one) are then repeatedly subdivided until each mapping node is associated with either a single source node or a single target node. These subdivisions are performed using collections of scan and cloning operations in an analogous fashion to that employed in constructing the bucket PMR quadtree. Essentially, the mapping nodes are subdivided until there is a one-to-one, one-to-many, or many-to-one relationship established between the source and target nodes through the mapping nodes. For our example dataset shown in Figure 40, the single mapping node is associated with thirteen source nodes and sixteen target nodes. Thus the mapping node must be split and the source and target nodes reassigned to the appropriate mapping node. The result of the first mapping node split is shown in Figure 41. Continuing with this process, the shaded mapping nodes 1, 2, and 3 in Figure 41 have satisfied the termination condition and do not need to be split further. Mapping node 4 must be subdivided further as it is associated with seven source nodes and seven target nodes. The result of splitting mapping node 4 in Figure 41 is shown in Figure 42. The final mapping node split (of mapping node 4 in Figure 42) results in the situation depicted in Figure 43, where each mapping node corresponds to either a single source or a single target node.

Once the mapping quadtree subdivisions are completed (see Figure 43 where each mapping

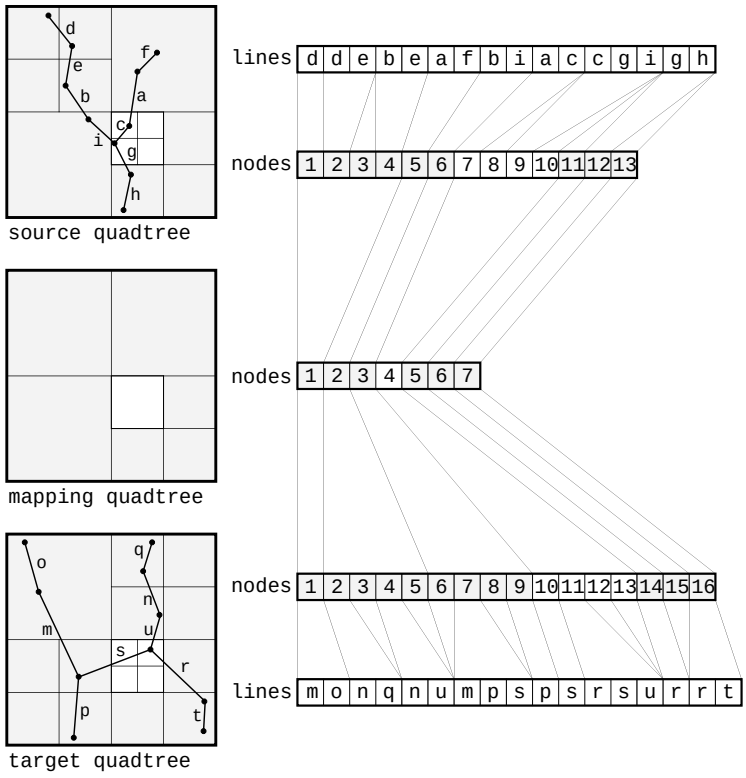


Figure 42: Mapping quadtree nodes at the completion of the second subdivision phase.

node corresponds to either a single source or target node), there exists a one-to-one, many-to-one, or one-to-many association between source and target nodes through the mapping nodes. The source and target quadtree nodes are then merged as necessary in order to establish a one-to-one relationship between the nodes in the two input maps. Merging four sibling nodes is accomplished by adjusting the Morton code[1] of the first node (to correspond to the Morton code of their parent), and deleting the three other siblings via a sequence of scans and a permutation operation. Similarly, the corresponding lines in the four segment groups are merged by resetting the segment flags to zero for the first lines in the second, third, and fourth segment groups. Once the segment flags are properly reset, the lines in the four sibling nodes are then sorted, and then any duplicate lines are deleted. For instance, if there are four source nodes associated with a single target node (refer to the first mapping node in Figure 43), then the four source nodes (which share the same parent node in the quadtree decomposition) are merged together (with duplicate line segments removed).

[1] A Morton code (or z order) is a mapping from two dimensions (or higher) to one which preserves the spatial locality of the multi-dimensional space in the one-dimensional space. The result of such a mapping is also termed a space-filling curve as the curve determined by the Morton code will pass through each point in the image (see [Oren86, Same90a] for further details).

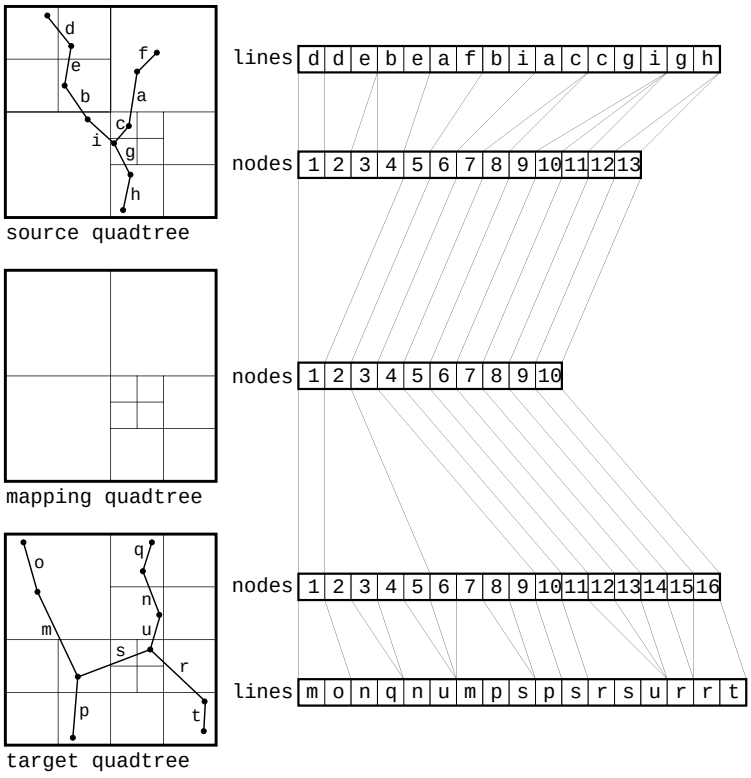


Figure 43: Mapping quadtree nodes at the completion of the subdivision phase which creates one-to-one, many-to-one, or one-to-many mappings between the source and target quadtrees.

This results in a one-to-one correspondence between these source and target nodes (see Figure 44). At the completion of the source and target node merging, the mapping quadtree may be discarded. The mapping quadtree can be conceptualized as the lowest common denominator (node-wise) of the source and target quadtrees.

It is important to note that the mapping quadtree is not a uniform grid. A naive algorithm might employ a uniform grid in order to establish a one-to-one mapping between the source and target quadtree nodes. The problem with this approach is that the size of the grid cells depends on the size of the largest possible quadtree node (i.e., the one whose corresponding block is the largest). Thus, if one of the two maps contained a significant amount of empty space, then the resulting large quadtree nodes would dictate a uniform grid composed of large cells. Thus, there might be many source quadtree nodes associated with a similarly large collection of target quadtree nodes prior to node merging. In order to avoid this behavior, we favor using a mapping quadtree which adapts to the decompositions of the source and target quadtrees.

The final mapping resulting from the use of a uniform grid rather than a quadtree is shown in

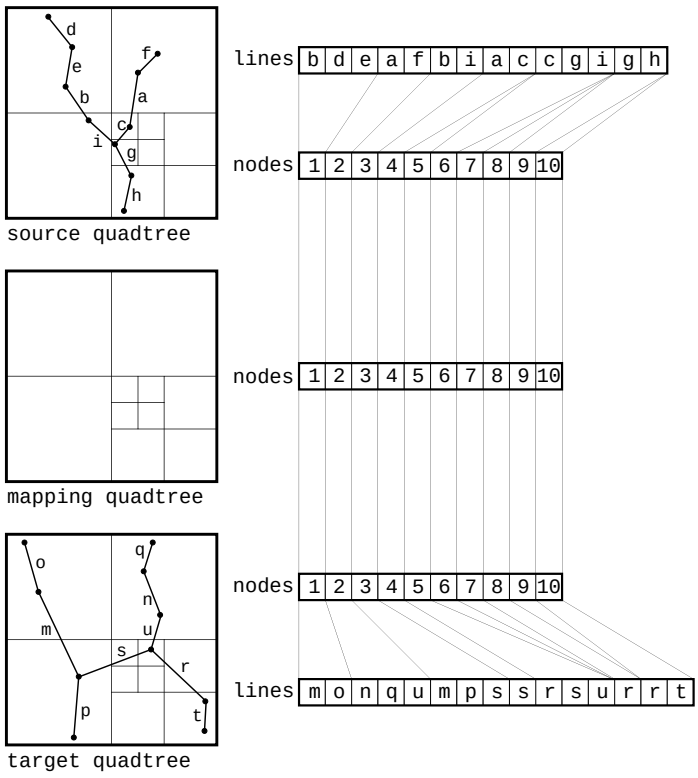


Figure 44: Quadtree nodes highlighting the one-to-one source to target node correspondence after node merging.

Figure 45. The size of the grid cells is dependent upon the largest-sized nodes in the source and target quadtrees (i.e., nodes 5 and 6 in the source quadtree, and node 1 in the target quadtree). Note that we are left with a many-to-many mapping in grid cell 4 prior to the node merging phase

The actual process of determining the line segment intersections begins with each source node broadcasting the endpoints of all associated line segments (i.e., all the line segments that are found in the quadtree node) to the set of line segments in the associated target quadtree node. Figure 46 highlights the source to target line communication for the example dataset. In the figure, the shaded nodes and lines represent inactive processors for which no action is necessary as there are either no source lines or target lines associated with the nodes. This is accomplished by the first line processor associated with each active source node (where an *active* line or node is defined as one that is participating in the current operation) passing its endpoints to the first line processor in the corresponding target node. In Figure 46, the first line processors are shown with arrows emanating from them directed at the corresponding source nodes. The source line endpoint coordinates are then shared among all line processors in the associated target node via a sequence of scan operations.

Each active target line processor then simultaneously determines whether or not the line that

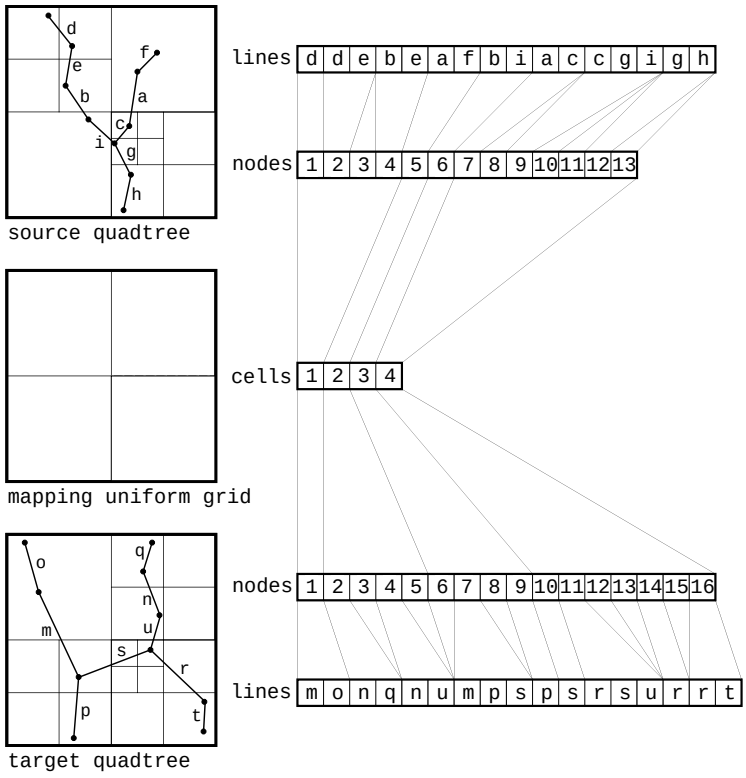


Figure 45: Alternative source to target quadtree mapping when employing a uniform grid rather than a quadtree.

it represents intersects the broadcasted source line segment. If the target line intersects the broadcasted source line, then the target line records the identifier of the intersecting line. Continuing this process, the collection of second line processors associated with each active source node passes their coordinates to the first processor in each active associated target node. Again, the line coordinates are then communicated among all line processors in the target node via a sequence of scan operations, and each target line processor determines in parallel whether or not it intersects the source line. Once all active source line processors have transmitted their coordinates to the associated target line processors, the intersection operation is complete and all target lines intersecting any of the source lines contain a list of the identifiers of the intersecting source lines. Figure 47 shows the result of the sequence of source line broadcasts. Intersecting source line identifiers are listed beneath the appropriate target line.

Given an $s \times s$ image, n line segments, and an associated quadtree with a bucket capacity b and height $\log s$, the data-parallel bucket PMR quadtree map intersection operation takes $O(n)$ time in the worst case. This degenerate case would arise if the final mapping quadtree contained a single node. In the average case where the expected tree height is $O(\log n)$, if we assume that there are

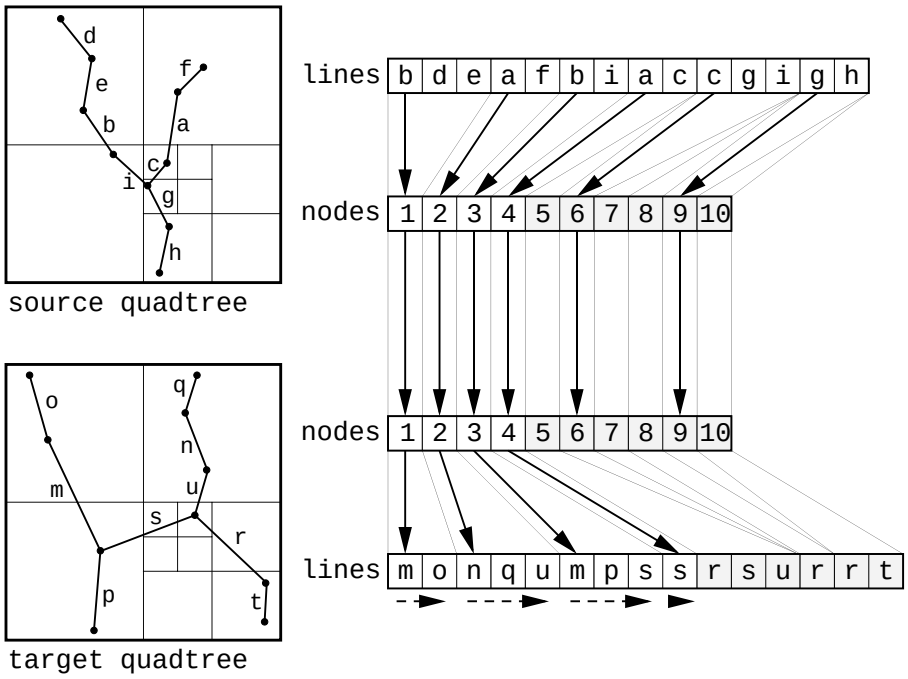


Figure 46: Source to target quadtree line endpoint communication channels for determining intersection. Shaded nodes and lines are not the recipient of any communications.

at most m source nodes that corresponded to a single target node, then the complexity of the map intersection operation is $O(\log n + mb)$. This is obtained by observing that establishing the source to target node mapping requires $O(\log n)$ operations (i.e., a fixed number of scans and reshuffles at each level of the quadtree), and that the actual intersection determination phase of the algorithm requires $O(mb)$ scans.

4.1.3 Spatial Range Query

The data-parallel bucket PMR quadtree spatial range query algorithm proceeds in a fashion similar to the intersection algorithm where the quadtree decomposition is employed to maximize the number of parallel operations. In the following description, again assume that there are two input data-parallel bucket PMR quadtrees of the same size. We will term the quadtree that contains the line segments to be expanded the source quadtree (also referred to as the *expansion set*), and the quadtree containing the line segments from which to test for intersection with the expansion set the target quadtree.

The algorithm begins by establishing a mapping between source and target nodes in an identical manner to that employed at the beginning of the intersection algorithm. Once the one-to-one

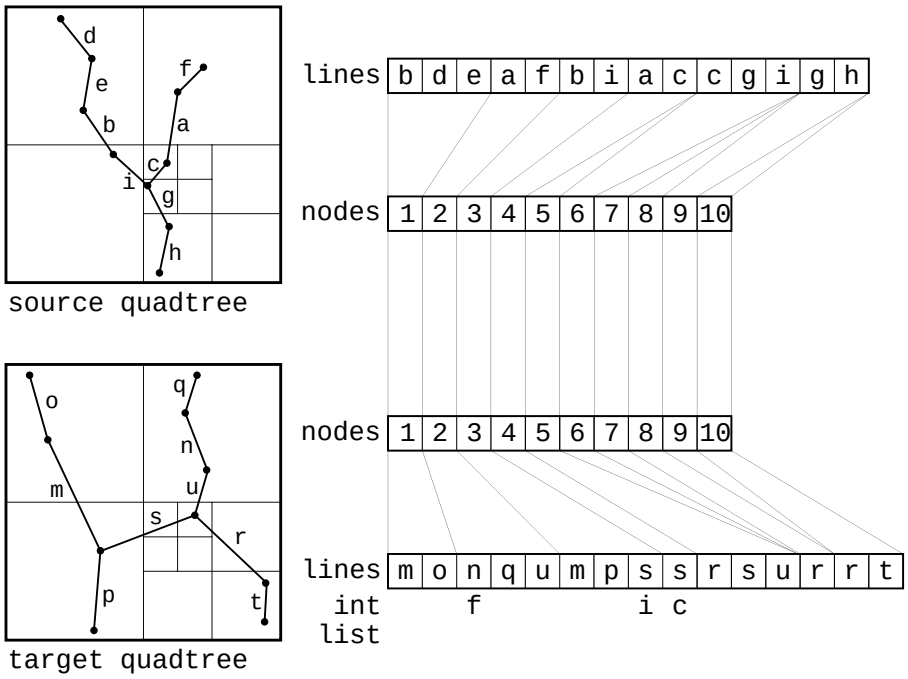


Figure 47: Result of the source line broadcast operation with intersecting source line identifiers shown beneath the appropriate target line.

source to target node mapping is established (refer to Figure 48 for an example source to target node mapping and an example expansion region which is denoted by the gray region superimposed on the source quadtree), the process of determining all target lines that intersect the region defined by the source line expansion set and the expansion radius proceeds in an iterative fashion.

The spatial range query algorithm operates on a single size set of nodes at a time (i.e., all nodes of size $r \times r$ where $r \leq s$), iterating upward from the smallest sized nodes to the root node in the quadtree representation. Each node in this set of smallest-sized nodes (shown unshaded in Figure 49) broadcasts in parallel the coordinate values of the endpoints of all associated line segments (i.e., all the line segments that are found in the quadtree node) to the set of line segments in the associated target quadtree node. This is accomplished in a similar fashion as was done with the intersection algorithm, with the first line processor associated with each active source node passing coordinate values of its endpoints to the first line processor in the corresponding target node. These coordinate values are then shared among all line processors in the associated target node via a sequence of scan operations. Each active target line processor then simultaneously calculates the Euclidean distance between itself and the communicated source line. If the separation distance is less than the radius of expansion, then the target line records the identifier of the source line as lying within the space defined by the source expansion set.

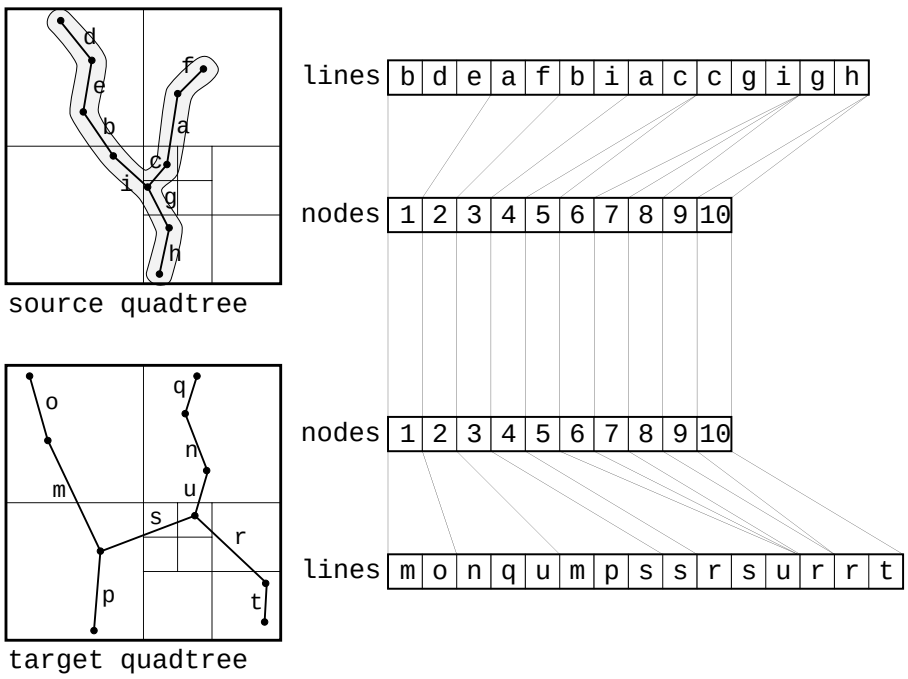


Figure 48: One-to-one source to target node mapping and region formed by the expansion set and expansion radius.

Continuing this process, the second line processor associated with each active source node passes the coordinate values of its endpoints to the first processor in each active associated target node. Again, the coordinate values of the line's endpoints are then communicated among all line processors in the target node via a sequence of scan operations, and each target line processor calculates the distance between itself and the source line. Once all active source line processors have transmitted the coordinate values of their endpoints to the associated target line processors, the communication stage for the currently active quadtree node size is complete. In our example, only line segment `s` is found to intersect the expanded region in the first iteration, intersecting line `c`.

Before the source node iteration continues and nodes of twice the current active node size are made active, each of the currently active source and target sibling nodes is merged. As an optimization to lessen the number of source line segment communications, all source line segments in the currently active source nodes whose distance from the border of their corresponding block is greater than the expansion radius are deleted. If a source line lies at distance less than the expansion radius from the border of the source node's corresponding block, then the source line must be retained for later rebroadcast. This is because a source line's region of expansion (the area within the expansion radius of the line) may intersect target lines that are not associated

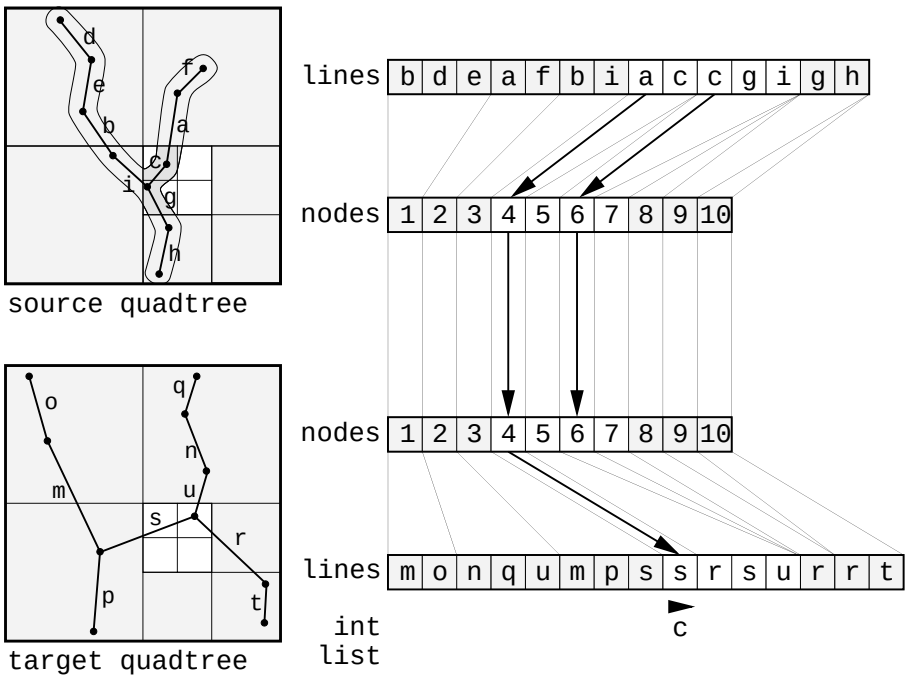


Figure 49: Active source and target nodes of minimal size (with other larger inactive nodes shaded) during the first iteration of line communications. The expansion region is superimposed in darker gray on the source quadtree.

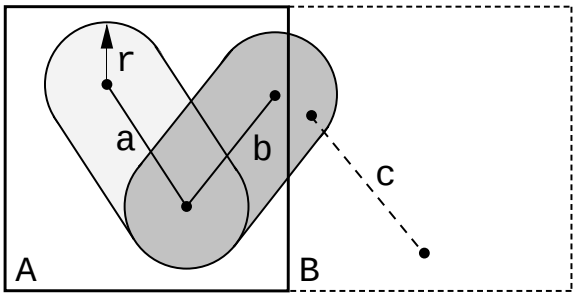


Figure 50: Example where one source line a may be deleted, and the second line b may not be deleted during the source node merge phase. For the given radius of expansion r, line b's expansion region might intersect a target line c in another currently unassociated node B.

with the block corresponding to the source line's node (i.e., a target line may lie very close to the border in an adjoining node). For example, consider the situation depicted in Figure 50 of two line segments a and b in a source node corresponding to block A. Given the example situation (with the expansion radius r), a may be safely deleted as its expansion region can not possibly intersect any other blocks outside of A, while b may not be deleted as its expansion region intersects other blocks (i.e.,block B). Note that there is no need to delete any target lines as all the target lines are checked for intersection with a source line in parallel. Thus removal of a target line does not affect

performance. Of course, if there are more target lines than processors, then this may be a useful optimization.

The result of the node merging and line deletion after processing the smallest-sized nodes is depicted for our example dataset in Figure 51. Note that no source lines were deleted as each of their expansion regions intersected the border of the blocks corresponding to their nodes in the initial quadtree.

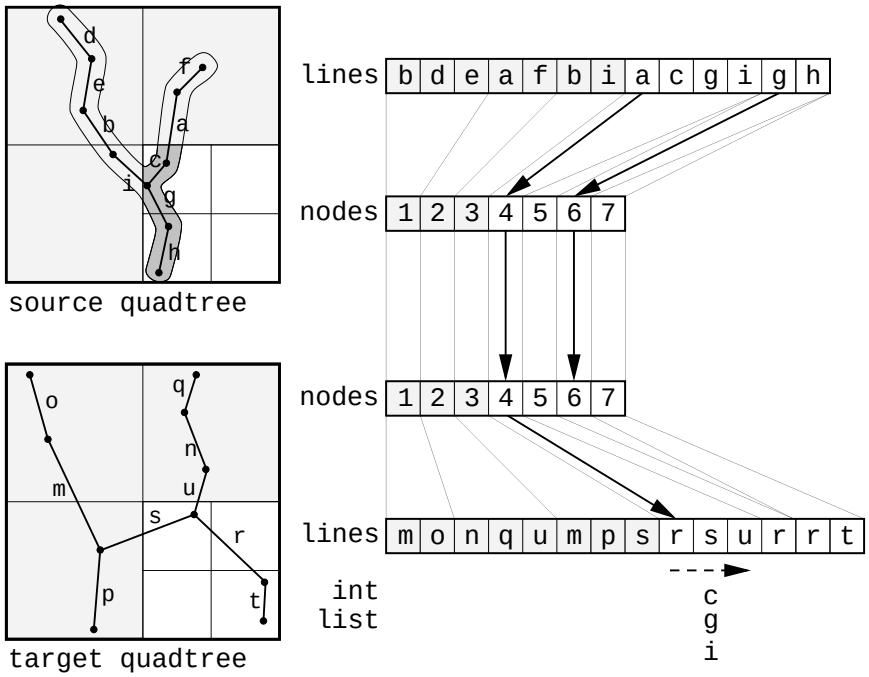


Figure 51: Active source and target nodes of minimal size (with other larger inactive nodes and lines shaded) after the first iteration of line communication and node merging. Note that the dashed target line s indicates that it was marked as intersecting the expansion region during the iteration that was just completed. The expansion region is superimposed in darker gray on the source quadtree.

After all currently active source and target nodes have been merged, we continue the above process, making all nodes of twice the size as the currently active nodes active. Basically, we are climbing one level of the quadtree as we move from the deepest node toward the root node. The result of the second iteration is shown in Figure 52 with no new intersecting target tree lines being detected. Once the level of the root of the quadtrees has been processed, the spatial range query operation is complete, with all lines in the target quadtree that intersect the source expansion set having a list of identifiers of the source lines that lay within the radius of expansion. Figure 53 depicts the situation immediately prior to the final round of source to target line communications.

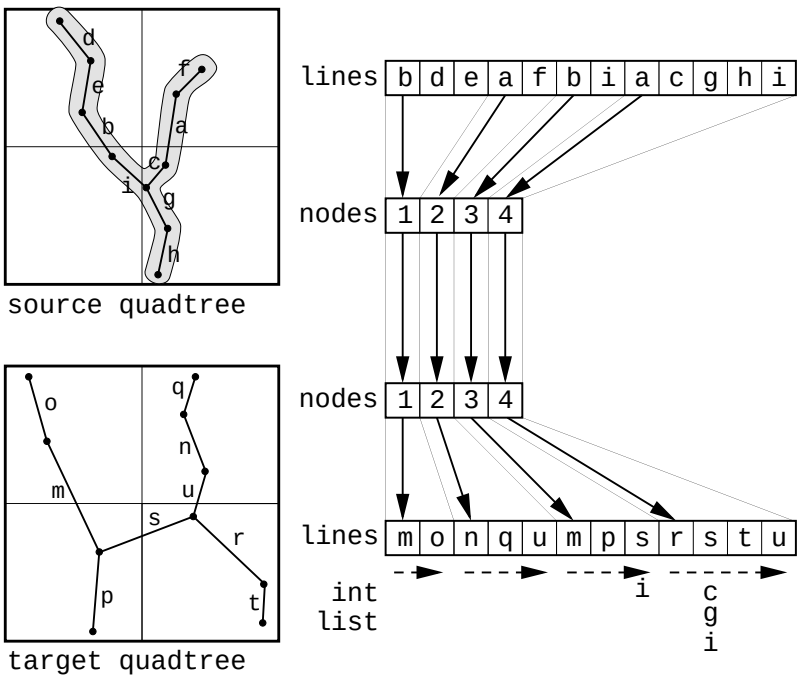


Figure 52: Active source and target nodes after the second iteration of line communication and node merging. The expansion region is superimposed in darker gray on the source quadtree.

Notice that during the algorithm, a line in the target quadtree could be marked as intersecting the query region several times. However, the reporting of the intersection only happens once at the conclusion of the algorithm. In order to avoid this duplication, once all processing of the spatial range query is completed, the set of intersecting target lines is 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 inclusive segmented maximum scan, and a downward inclusive segmented copy scan operation can be used to resolve any inconsistencies, resulting in all target lines being properly marked. Duplicate target lines may then be safely deleted (as each instance of a line is properly marked) and the result of the join may then be reported. This is important as it avoids the need to eliminate duplicate answers [Aref92].

Given an $s \times s$ image, n line segments, and an associated quadtree with a bucket capacity b and height $O(\log s)$, the data-parallel bucket PMR quadtree map spatial range query operation takes $O(n \log s)$ time in the worst case. This degenerate behavior would occur if at each of the $O(\log s)$ stages of the spatial range query, none of the source lines were removed from consideration. This is possible if the radius of expansion is equal to the length of the diagonal across the space spanned by the quadtree.

In the average case, assume that each of the n line segments is broadcast a small number of times (i.e., $O(n)$) over the entire spatial range query operation. Each of the $O(\log s)$ stages will require $O(\log n)$ time to sort and merge the lines prior to broadcast and marking. Combining the time required to broadcast the line segments with the sorting and merging costs, the data-parallel bucket PMR spatial range query will take $O(n + \log n \log s)$ time in the average case.

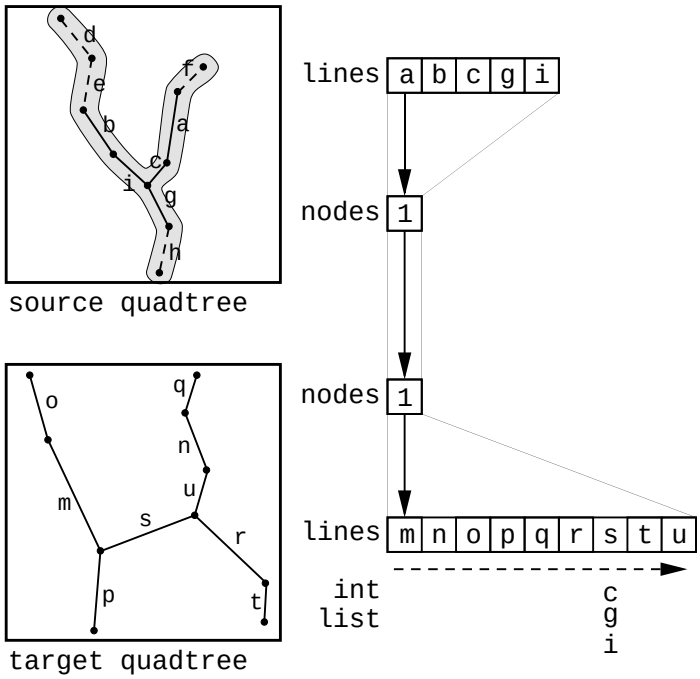


Figure 53: Active source and target nodes immediately prior to the final iteration of communication. Note that target lines `n` and `s` have been marked as intersecting the expansion region on previous iterations. Additionally, source lines `d`, `e`, `f`, and `h` were deleted during a prior source node merge phase as their expansion regions did not lie outside of the blocks corresponding to their source nodes. The expansion region is superimposed in darker gray on the source quadtree.

4.2 R-tree Spatial Queries

4.2.1 Polygonization

The polygonization process for other data-parallel spatial structures such as the R-tree is similar to that described for the bucket PMR quadtree in Section 4.1.1. Given a data-parallel R-tree, we start by constructing a partial winged-edge representation. Once the partial winged-edge representation is completed, each line processor locally assigns an initial polygon identifier for the bordering polygons on the left and right sides (see Section 4.1.1 for details).

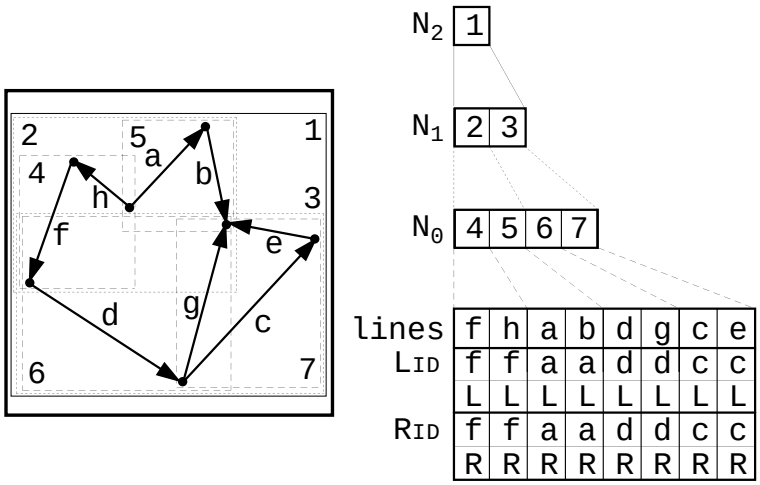


Figure 54: Initial polygon assignments.

The initial polygon assignment is shown in Figure 54 for our example dataset where the left and right polygon identifiers are contained in processor sets $\texttt{L}_{\texttt{ID}}$ and $\texttt{R}_{\texttt{ID}}$ respectively. Next, beginning with the nodes at the leaf level of the R-tree, we merge all sibling lines together into the parent nodes. All lines that intersect any of the overlapping regions formed by the bounding boxes of the nodes that have been merged are marked for rebroadcasting among the lines in the merged nodes. This is necessary in order to propagate the equivalence between the different identifiers in the merged nodes which represent the same polygon. For example, consider Figure 55a where we have two R-tree nodes `A` and `B` that are to be merged. In this example, node `A` contains lines `(a,c,g,h)`, and node `B` contains lines `(b,d,e,f)`. In the figure, lines `(a,b,d)` must be rebroadcast to the merged set of lines (i.e., lines `(a,b,c,d,e,f,g,h)`) as they intersect the overlapping region formed by the bounding boxes of nodes `A` and `B`. The purpose of this operation is to update the winged-edge representations of any necessary lines (i.e., lines `a` and `b` in Figure 55a). When the winged-edge representation is updated, we note any polygon identifiers that must also be updated. In the example in Figure 55, line `b` has both its left and right polygon identifiers updated; $\texttt{b}_{\texttt{L}}$ in Figure 55a becomes $\texttt{a}_{\texttt{L}}$ in Figure 55b, and similarly, $\texttt{b}_{\texttt{R}}$ becomes $\texttt{a}_{\texttt{R}}$. Line `a` does not have either of its polygon identifiers updated because its left and right polygon identifiers are lexicographically minimal.

For all such polygon identifier updates (e.g., $\texttt{b}_{\texttt{L}}$ to $\texttt{a}_{\texttt{L}}$ and $\texttt{b}_{\texttt{R}}$ to $\texttt{a}_{\texttt{R}}$ in Figure 55), we broadcast the updates to all other lines in the merged node via scan operations. Locally, if the transmitted polygon update matches either the left or right polygon identifiers of the local line, the local polygon identifier is updated to reflect the polygon identifiers that have been broadcast. For example, in

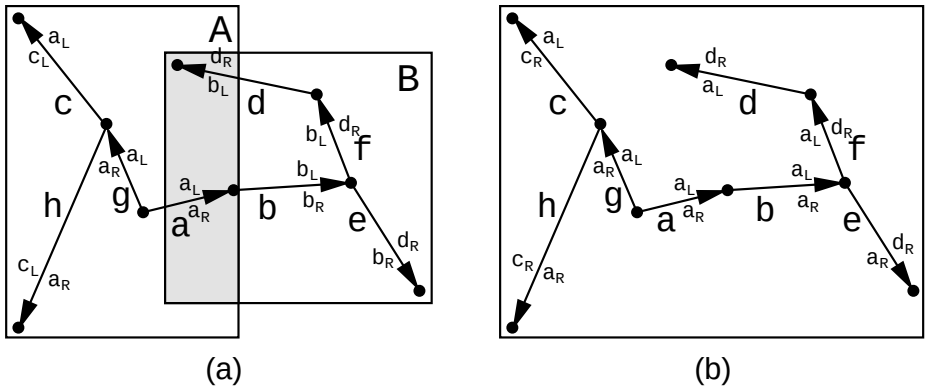


Figure 55: (a) Example of two nodes merging and which lines must be rebroadcast to all the merged lines; and (b), the result of the merge operation.

Figure 55a, the right polygon identifier of line e is updated to reflect the fact that polygon identifier b_R becomes a_R. Similarly, the left side polygon identifiers of lines d and f are updated to reflect the fact that polygon identifier b_L becomes a_L. The resulting polygon identifiers and merged nodes are depicted in Figure 55b. This process continues up the entire R-tree until all lines are contained in a single node and all necessary broadcasts have been completed. The first round results in the merging of leaf nodes 4, 5, 6, and 7 in Figure 54 into leaf nodes 2 and 3 in Figure 56, which depicts the result of the first round of leaf node merging. The final configuration of our original example dataset is depicted in Figure 57. The identifiers assigned to the three polygons are shown in the figure by enclosing the identifiers within circles.

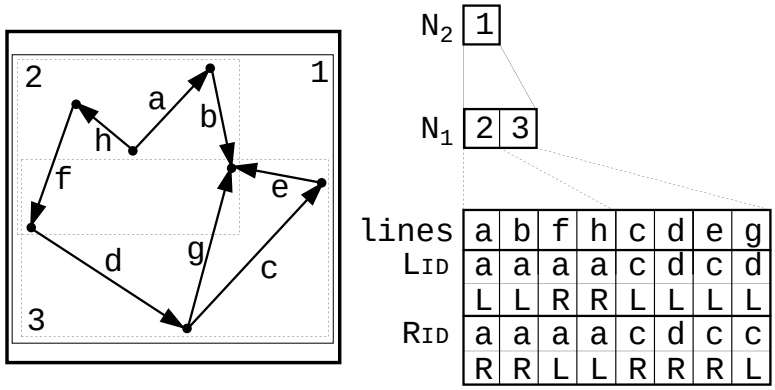


Figure 56: Polygonization after first round of leaf node merging.

The R-tree'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. However, the non-disjoint decomposition of the R-tree causes increased computational complexities in the local broadcasting phase of the sibling

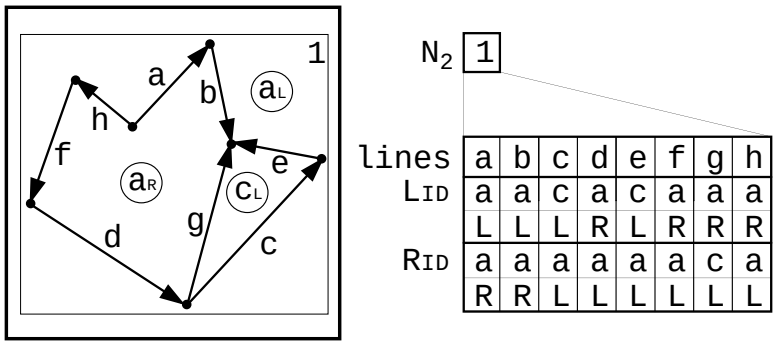


Figure 57: Completion of the polygonization operation with the final polygon labels encircled.

merge operation in comparison to an analogous disjoint decomposition spatial data structure such as the PMR quadtree [Nels86, Nels87] or the R+-tree [Falo87]. This is because it is often the case that many lines fall in the intersecting areas when the R-tree nodes are merged. With representations based on a disjoint decomposition of space, only those lines that intersect the splitting lines (i.e., cell boundaries) would need to be locally broadcast during the sibling merge operation.

Now, let us estimate the number of broadcasts necessary during the polygon identification process due to the lines intersecting overlapping regions. In the average case, assume that each R-tree node has a fanout of M. Let c (where $0 \leq c \leq 1$) be the fraction of the lines in each node that intersect one or more of the overlapping regions formed by the bounding boxes of the nodes that have been merged. Also, let h denote the height of the R-tree (without loss of generality, $h = \log_M n$, where n is the number of lines in the tree). Using the fact that $M^h = n$, it can be shown that the number of local broadcasts B that must be made during the merging phases due to the intersection of lines with the overlapping regions is

$$B = \sum_{i=2}^{h} cM^i \leq n \left(\frac{M}{M-1} \right).$$

This is in the worst case $O(n)$. However, the average-case complexity is expected to be lower. In particular, the average-case complexity of the line broadcasting step is dependent upon the ability of the node splitting algorithm to partition the buckets as much as possible (therefore lowering the fraction c of lines intersecting the overlapping regions). A more detailed analysis is a subject for further research.

4.2.2 Map Intersection

Given data-parallel source and target R-trees, a correspondence between the source and target R-tree nodes must be established which will be used to determine patterns of parallel communication. Basically, for each source R-tree leaf node s, we must determine the intersecting target leaf nodes. Any source line in s might intersect another target line contained in a target leaf node t that intersects s. For example, in Figure 58, source line segment `c` (contained in source node `B`), might intersect target line `s` (contained in target node `L`) as nodes `B` and `L` intersect. If a source and target leaf node do not intersect, then it is not possible for the associated contained lines to intersect. In Figure 58, source line `c` cannot intersect target line `r` (contained in target node `N`) as nodes `B` and `N` do not intersect.

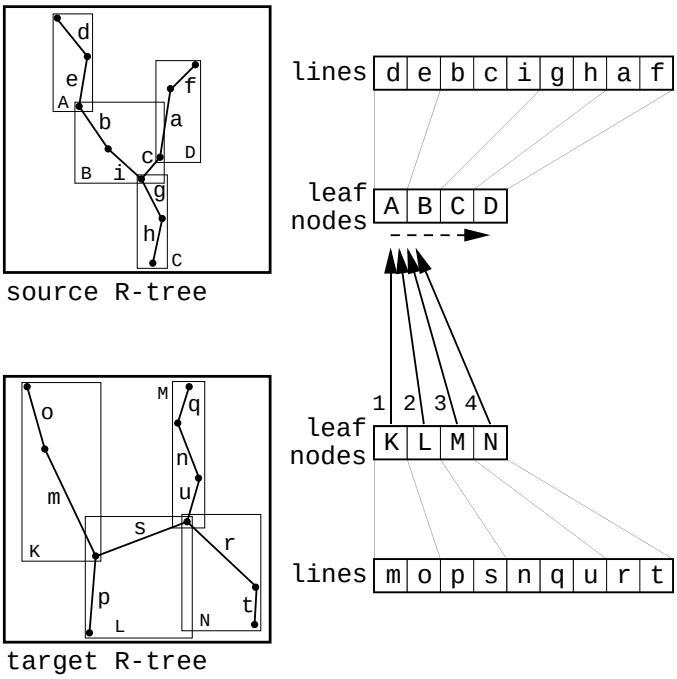


Figure 58: Example data-parallel R-trees for the same set of source and target lines as in Figure 40. Leaf node bounding rectangles are shown for both source (`A`, `B`, `C`, and `D`) and target (`K`, `L`, `M`, and `N`) data-parallel R-trees. Internal R-tree nodes and bounding boxes are omitted for clarity.

For each source leaf node, the process of determining which target leaf nodes it intersects is determined in a top-down manner (in the sequential domain, this problem has been studied for the `R`*-tree [Brin93]). We first check if the bounding rectangles of the two root nodes do not intersect. In this case, no leaf nodes can intersect. Otherwise, beginning at the root node level of the source and target R-trees, the children of the source root node are intersected with the children of the target

root node. The resulting intersections are then communicated to the child of these child nodes (i.e., the grandchildren of the root nodes) in each map. The grandchildren in the target R-tree then selectively communicate their bounding rectangles to the appropriate grandchildren in the source map (i.e., those source R-tree grandchildren whose parent was previously found to intersect the parent of the target R-tree grandchildren). This process continues on down the tree until the leaf nodes in each R-tree have determined all intersecting leaf nodes in the other data-parallel R-tree.

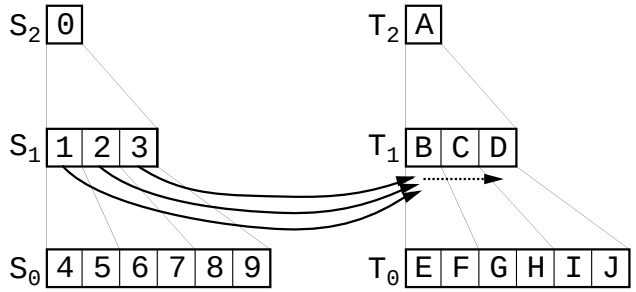


Figure 59: Example of first level node intersection determination in the top-down algorithm where three source nodes sequentially broadcast to the corresponding target nodes.

For example, consider Figure 59 where three source nodes in processor group S_1 must sequentially determine which target nodes in processor group T_1 they intersect. Because nodes 1, 2, and 3 are attempting to communicate with the same target node segment, the communications must be done sequentially (i.e., node 1 to node B, followed by node 2 to node B, etc.). In Figure 59, the dashed arrow beneath processor group T_1 signifies that a scan operation will be used to share the communicated information from the nodes in processor group S_1 (i.e., to nodes C and D). Thus, it is not necessary for nodes 1, 2, and 3 to independently communicate with target nodes C and D.

Once the example first level node intersection determination is completed in Figure 59, each source node in processor set S_1 will have established a list of intersecting target nodes in processor set T_1. This intersection information is then passed along to their children (i.e., the source nodes in processor set S_0). In this example, we assume that source node 1 intersects target nodes B and C, source node 2 also intersects target nodes B and C, and source node 3 intersects target nodes C and D. The iterative process then continues, with the source nodes in processor set S_0 determining which target nodes in processor set T_0 they intersect. In Figure 60, source nodes 4, 6, and 8 may communicate in parallel with target nodes E, G, and I, respectively, in processor set T_0 (who in turn share the communicated information with target nodes F, H, and J, respectively, via segmented scans). In the next iteration (which is not shown), source nodes 5, 7, and 9 will also communicate

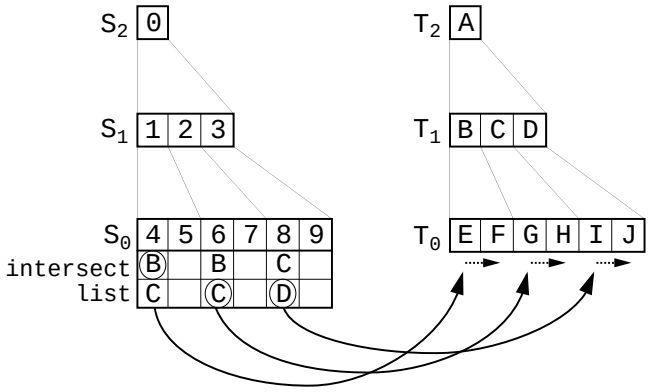


Figure 60: Example of second level node intersection determination in the top-down algorithm where three source nodes broadcast in parallel to the corresponding target nodes.

in parallel with target nodes `E`, `G`, and `I`, respectively.

Alternatively, we could employ a more straightforward though less elegant bottom-up approach to the problem of determining all source and target leaf node intersections. Each target node in turn transmits its bounding rectangle coordinates to the first source leaf node in the arbitrary linear ordering (i.e., source leaf node `A` in Figure 58). These coordinates will then be shared among all source leaf nodes via a series of scan operations. Once each source leaf node knows the coordinates of the communicated target leaf node, in parallel, each source leaf node then determines whether or not it intersects the target node. If there is an intersection, then the index of the target node is appended to the source leaf node's list of target node intersections. This situation is depicted in Figure 58, where the communication path between first target leaf node (node `K`) and the first source leaf node (node `A`) is shown. The dashed arrow beneath the source leaf nodes represents the scan operation that is employed to share the target node bounding rectangle coordinates among all of the source leaf nodes. This is more of a brute force approach where the spatial decomposition induced by the data-parallel R-tree is only used at the leaf level.

Figure 61 represents the situation found after all of the target leaf nodes have communicated their bounding rectangle coordinates to the source leaf nodes, and each source leaf node has compiled its intersection list. The intersection lists for each source leaf node are depicted as the collection of boxes beneath each source node identifier (e.g., source node `B`'s intersection list contains target node identifiers `K` and `L`).

Once all source/target node intersections have been determined, the source leaf nodes will then transmit the endpoint coordinates of all contained line segments to all intersecting target leaf node

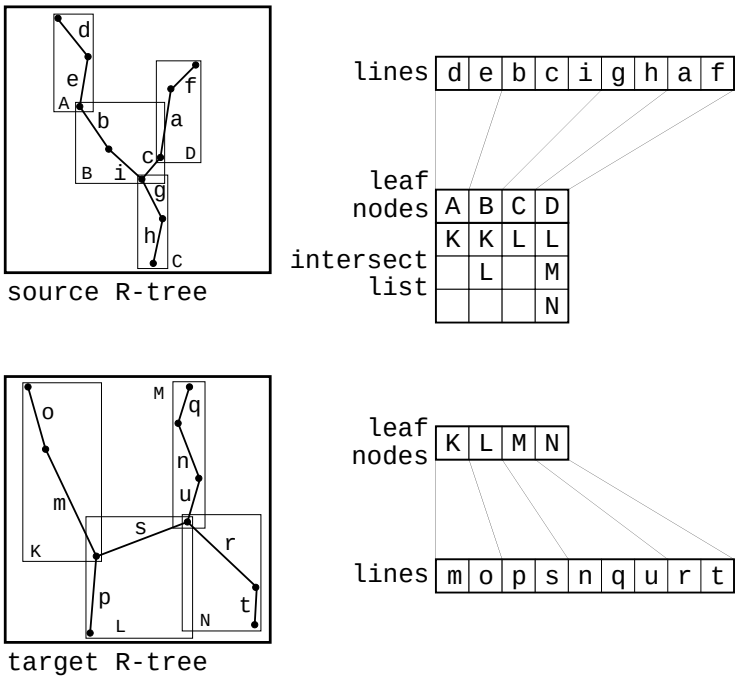


Figure 61: Source and target R-trees after all source and target leaf node intersections have been determined and recorded in the appropriate source nodes.

line segments. Each target line then determines whether or not it intersects any source line segment.

Unlike the data-parallel bucket PMR quadtree source to target node communication process, the non-disjoint irregular partitioning of space induced by the data-parallel R-tree decomposition creates additional communication difficulties. Instead of all active source nodes communicating in parallel with the associated target node in the one-to-one mapping provided by the mapping quadtree in the data-parallel bucket PMR quadtree, the data-parallel R-tree source to target leaf node communications are scheduled and made in an iterative process. This is due to the situation arising when multiple source nodes intersect a single target node (e.g., in Figure 61, source nodes `B`, `C`, and `D` each intersect target node `L`).

The scheduling problem is analogous to the Chromatic Index problem [Gare79] where the set of source and target leaf nodes may be thought of as a set of vertices, and the intersections between the nodes as edges between the vertices. These edges and vertices form a bipartite graph, and it has been shown that there exists a polynomial time algorithm for scheduling the necessary communications [Gabo76].

In solving the communication scheduling problem, a non-optimal solution was chosen using a greedy approach. At each iteration of source to target communications, the first source node

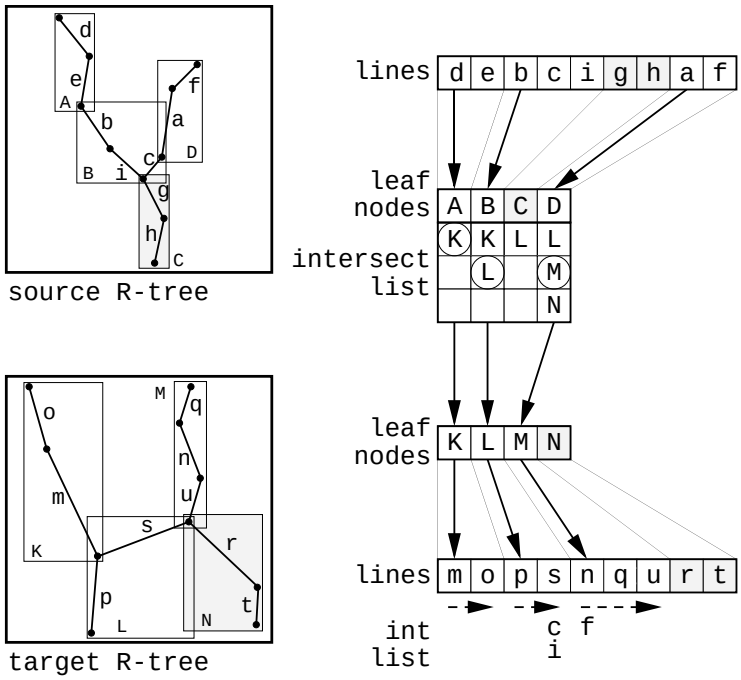


Figure 62: Source and target data-parallel R-trees with first round of communication scheduling highlighted using circles. Inactive lines and nodes are shaded. Target lines are marked with the names of the intersecting source lines.

requiring communication selects the first target node in its intersection list (e.g., in Figure 62, source node A selects target node K). The next source node requiring communication then selects the first target node that has not been previously selected by the first source node (e.g., in Figure 62, source node B selects target node L). Continuing in this fashion, all following source nodes requiring communication select the first target node in their intersection lists that has not been previously selected by another source node during the current communication iteration.

Once all possible communications have been determined for the current iteration, each source leaf node that has a scheduled communication is made active (in Figure 62, this corresponds to source nodes A, B, and D). Using a technique similar to that employed in the data-parallel bucket PMR quadtree intersection algorithm, each active source node broadcasts the endpoints of all associated line segments (i.e., all the line segments that are found in the R-tree node) to the set of line segments in the associated target R-tree node. This is accomplished by the first line processor associated with each active source node passing its endpoints to the first line processor in the corresponding target R-tree node. In Figure 62, the first line processors (i.e., source lines d, b, and a) are shown with arrows emanating from them directed at the corresponding source nodes. Once the source line endpoint coordinate values have been communicated to the first target line processor

in the associated target R-tree node, they are then shared among all target line processors in the same target node via a sequence of scan operations.

Each active target line processor then simultaneously determines whether or not it intersects the broadcasted source R-tree line segment. If the target line intersects the broadcasted source line, then the source line identifier is included in a list of intersecting lines that is associated with the target line. Continuing this process, the collection of second line processors associated with each active source R-tree node (e.g., in Figure 62, source lines `e`, `c`, and `f`) passes its endpoints to the first processor in each active associated target node. Again, the line endpoints are then communicated among all line processors in the target node via a sequence of scan operations, and each target line processor in parallel determines whether or not it intersects the source line. If a target line intersects the broadcast source line, then the source line identifier is included in the list of intersecting lines that is associated with the target line. Once all active source line processors have transmitted their coordinates to the associated target line processors, the current iteration of communications is complete, and all active source nodes delete the target node that was the recipient of their communications from their intersection lists. The target nodes to be deleted from the intersection lists at this point are the circled identifiers in Figure 62.

Continuing with this process, the second round of source node to target node communications must be scheduled. Following an identical selection process as before, the first source node requiring a communication makes its selection. Each following node (in our arbitrary linear ordering of source leaf nodes) requiring a communication first determines whether or not any of its remaining intersecting target nodes has not been selected by a preceding source node. If this is the case, the selection is made, and the following source nodes requiring communication make their selections (provided an unselected target node in their intersection lists is available). In Figure 63, the selected target nodes are shown in the intersection lists enclosed in circles.

Once all source nodes have communicated their enclosing source lines with all of the target nodes in their intersection lists, and all intersecting target lines have been marked if they intersect a broadcasted source line, the intersection operation is complete. For our example data set, the third and final round of communications is shown in Figure 64, with only source node `D` broadcasting source lines `a` and `f` to the target lines `p` and `s` contained in target node `L`.

Given n line segments and an associated R-tree with a node capacity M and height $\log n$ (recall from the discussion of the R-tree building process that an R-tree containing n lines is of worst-case height $O(\log n)$), the data-parallel R-tree map intersection operation takes $O(n + M \log n)$

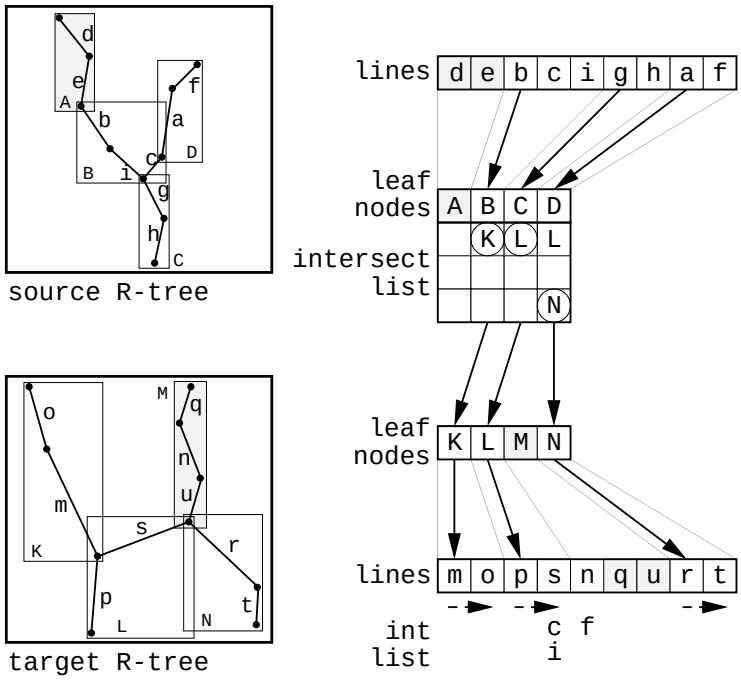


Figure 63: Source and target data-parallel R-trees with second round of communication scheduling highlighted using circles. Shading indicates nodes not participating in the current round of communication. Note that source node intersection lists have been modified following the first round communication depicted in Figure 62. Target lines are marked with the names of the intersecting source lines.

time. This is obtained by observing that establishing the source to target node mapping requires $O(M \log n)$ operations (i.e., $O(M)$ scans at each level of the data-parallel R-tree). The target line marking phase of the algorithm requires at most $O(n)$ scans. This is the case when each source leaf node intersects each target leaf node, causing each of the $O(n/M)$ source leaf nodes which contains M lines to communicate with each of the $O(n/M)$ target leaf nodes. Recall that we have a maximum of M lines in each source leaf node, and that there is a total of n lines.

4.2.3 Spatial Range Query

The algorithm for performing the spatial range query for data-parallel R-trees is very similar to that employed when computing the intersection as described in Section 4.2.2. There are two small modifications that are necessary in adapting the data-parallel R-tree intersection algorithm to a spatial range query algorithm.

The first modification involves the process of determining the target nodes with which each source node must communicate. In the intersection algorithm, this required finding all source/target

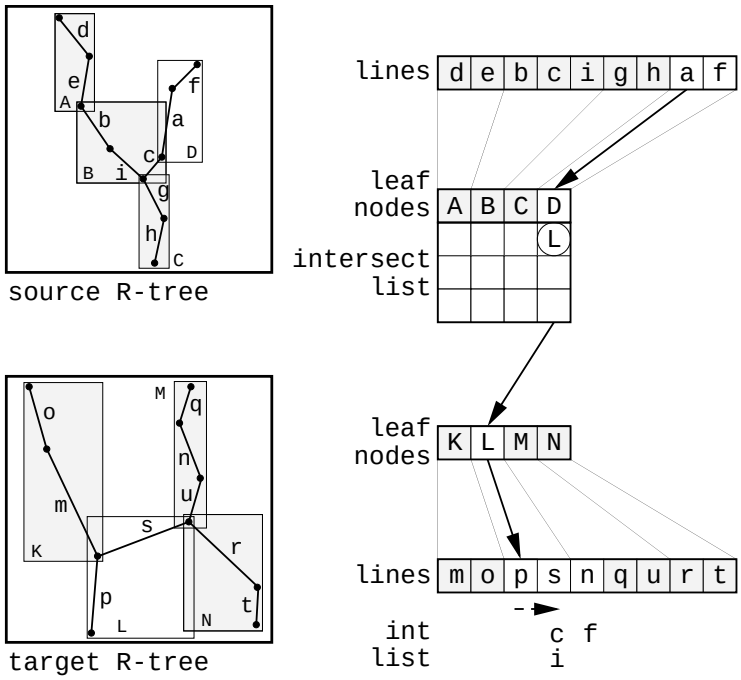


Figure 64: Source and target data-parallel R-trees with final round of communication scheduling highlighted by use of circles. Inactive lines and nodes are shaded. Note that only source node D needs to communicate with a target node. Target lines are marked with the names of the intersecting source lines.

node intersections. If the bounding rectangles for each source node are extended by the radius of expansion in each dimension prior to calculating all node intersections, we ensure that no necessary source line to target line communications are overlooked.

The second and final modification to the data-parallel R-tree intersection algorithm concerns the source line to target line intersection calculation. Rather than determining whether or not two lines intersect, we calculate the distance between the source and target lines. If the distance is less that the radius of expansion, the target line will clearly intersect the expansion region of the associated source line and should thus be marked. Once these two small modifications are made to the data-parallel R-tree intersection algorithm, it also functions as a spatial range query algorithm.

The data-parallel R-tree map spatial range query operation also takes $O(n + M \log n)$ time, with the complexity analysis being identical to that of the data-parallel R-tree map intersection operation at the end of Section 4.2.2.

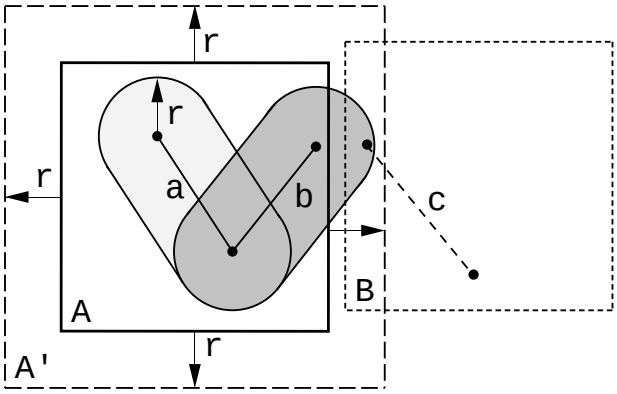


Figure 65: Example where the enlargement of the source data-parallel R-tree bounding rectangle A by the expansion radius r (creating bounding rectangle A') prior to node intersection detection is necessary. The dashed bounding rectangle B represents a target data-parallel R-tree node that contains target line c.

4.3 R$^+$-tree Spatial Queries

Give that the R$^+$-tree makes use of an irregular decomposition like the R-tree, the R$^+$-tree spatial query algorithms are quite similar to those described in Section 4.2 for the R-tree. The R$^+$-tree's disjoint decomposition does however enable line segments to exist in different leaf nodes in an analogous fashion to the bucket PMR quadtree. This small difference from the R-tree (which does permit a line segment to lie in more than one leaf node) results in some additional processing in the R$^+$-tree algorithms as compared with the corresponding R-tree algorithms due to increased numbers of lines and nodes.

4.3.1 Polygonization

The R$^+$-tree polygonization algorithm is very similar to that for the R-tree as described in Section 4.2.1. 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 4.1.1. 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 endpoints fall 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.

4.3.2 Spatial Join

The R^+-tree spatial join algorithms (intersection and spatial range query) are identical to the ones used with the R-tree in Section 4.2.3, 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, a line in the target map may be marked as within a given distance more than once. This is analogous to the situation encountered at the end of the PMR quadtree spatial join (see Section 4.1.3). In a similar fashion, target lines are first sorted by identifier, then an upward inclusive segmented maximum scan, and a downward inclusive segmented copy scan operation are used to remove any duplicates and false negatively marked lines. Lines are considered false negatively marked if another instance of the same line is marked as intersecting, and they are not marked. The removal of the duplicates and false negatives results in all target lines being properly marked.

5 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 35,000 line segments, shown in Figure 66). 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).

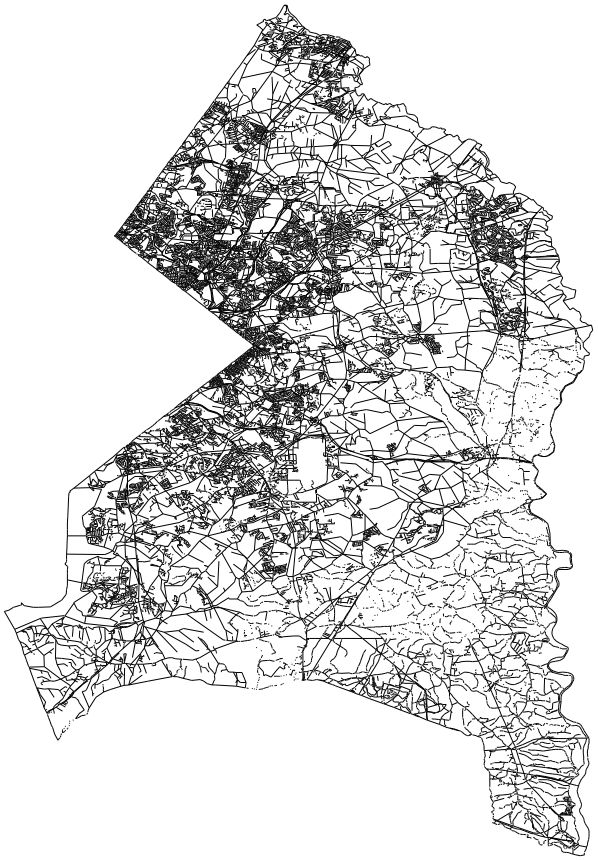


Figure 66: Map of Prince Georges County, MD.

5.1 Data Structure Build Performance

Figure 67 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

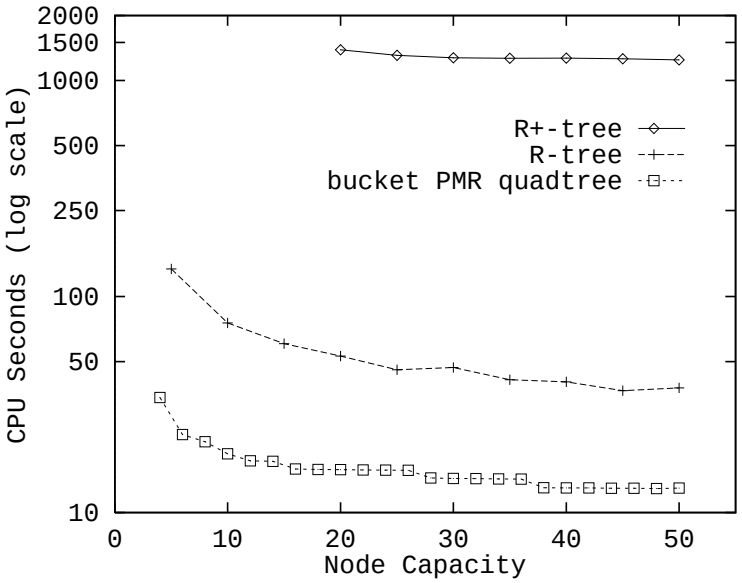


Figure 67: Build times for the three data structures for the map of Prince Georges County, MD (35,000 lines).

node sizes. Building times for the three data structures exhibit analogous behavior in the sequential environment [Hoel92]. It is also apparent that building 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 69 shows the build times for the R^+-tree of the Fredericksburg, VA map containing approximately 1700 line segments (see Figure 68 for a map of Fredericksburg). 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 [Beck90], has been observed to be 40% 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 69. As is the case for the Prince Georges map in Figure 67, increasing the node capacity also results in decreased build times.

It is important to note that although the data of Figure 69 corresponds to a map that is

Figure 68: Map of Fredericksburg, VA.

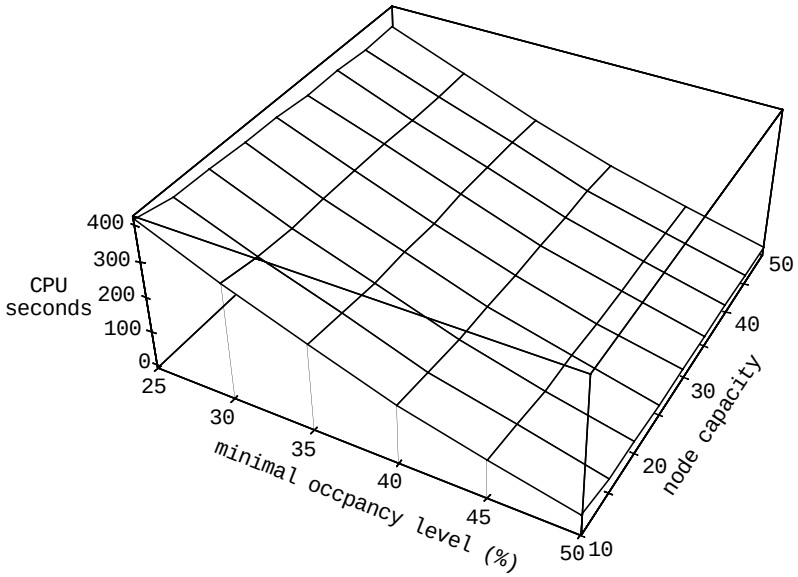


Figure 69: Execution times in seconds for the R^+-tree build algorithm for the map of Fredericksburg, VA (1700 lines).

approximately 5% of the size of that in Figure 67 (i.e., 1700 lines versus 35,000 lines), the R^+-tree takes 198.8 seconds to build while the R-tree (using a node capacity of 50 and a minimal occupancy level of 40%) for the same map requires 37.8 seconds to build and the bucket PMR quadtree requires just 13.0 seconds. We found that despite the R-tree and R^+-tree being quite similar in structure, the R^+-tree takes approximately two orders of magnitude longer to build per line segment in the dataset (e.g., see Figure 67). This difference is attributable to a combination

Table 1: Data structure build statistics for the R-tree and R+-tree both using an artificially high 49.5% minimal occupancy level for the Prince Georges map. All times are in seconds.

node	R-tree		R+-tree	
capacity	time	scans	time	scans
25	37.2	865	1309.3	28,212
30	35.6	823	1274.6	27,545
35	33.5	739	1268.0	27,305
40	30.4	654	1269.2	27,187
45	29.5	614	1261.3	27,040
50	28.5	614	1246.6	26,691

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 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 four 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 occupancy level of 49.5% (resulting in approximately 3000 line clips) and a node capacity varying between 25 and 50. This took between 1309.3 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 40% (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.

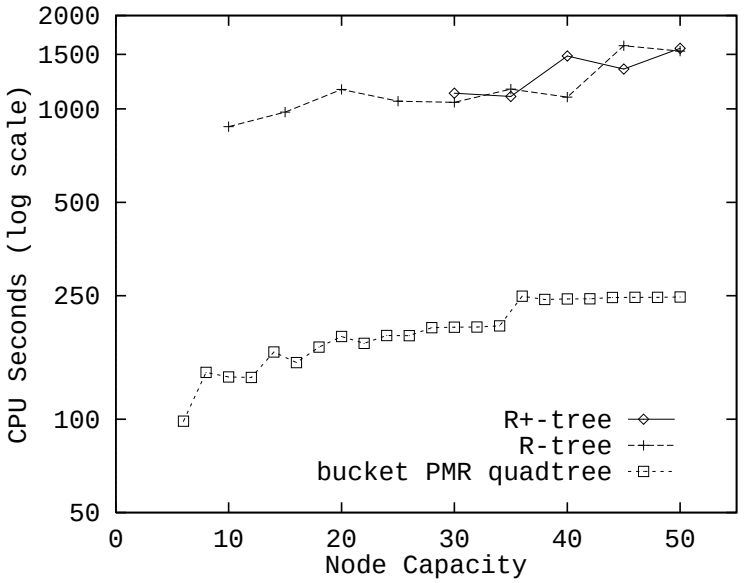


Figure 70: Polygonization execution times for the three structures.

5.2 Polygonization Performance

Figure 70 shows the execution times for map polygonization for each of the three spatial data structures using the Prince Georges maps built in Section 5.1. 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 40% level. From the figure it is clear that for polygonization, 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 M implies a fanout of M), 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) in parallel than the R-tree and the R^+-tree.

5.3 Spatial Join Performance

The key issue in the spatial join 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 size of 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 implementing 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 broadcast 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 (28,514 line segments in contrast to a total of 35,000 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 16,384 $\times$ 16,384. 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 20 to 50 for the R^+-tree.

Figure 71 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.

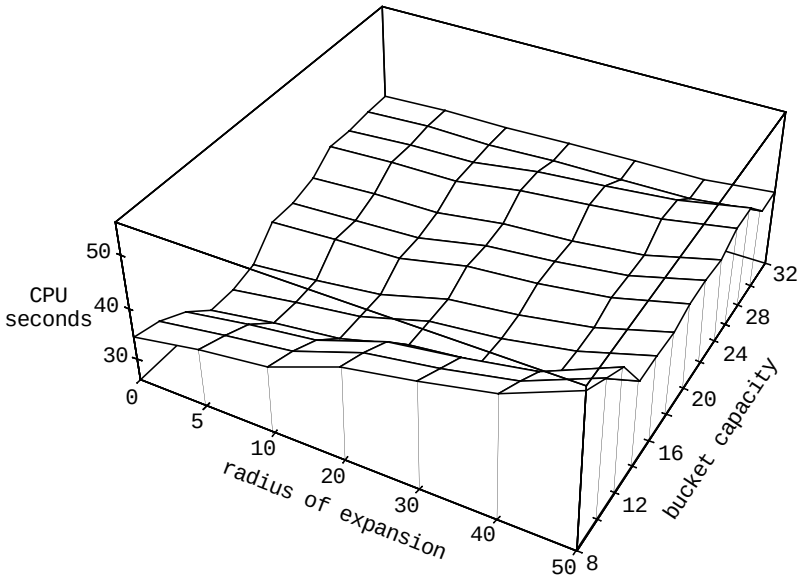


Figure 71: Execution times in seconds for the bucket PMR quadtree spatial join algorithm.

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 72 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 those 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 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 R-trees with 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

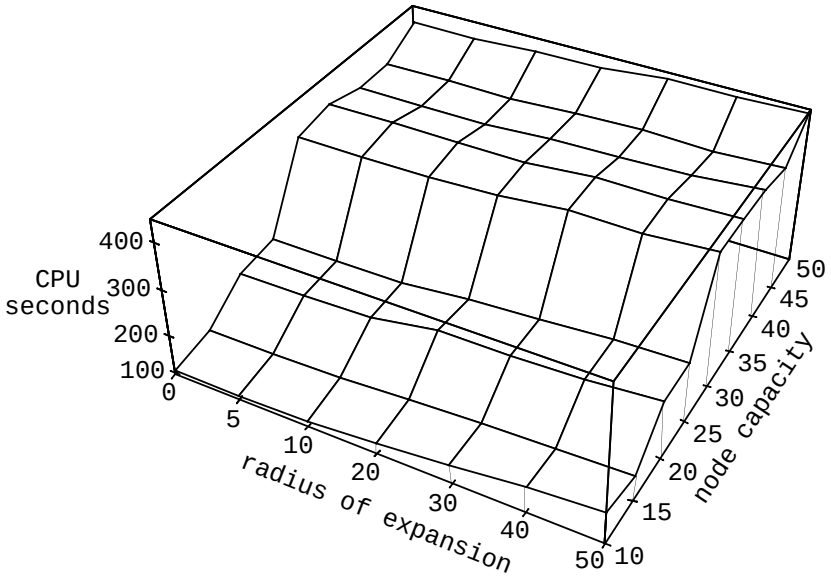


Figure 72: Execution times in seconds for the top-down R-tree spatial join algorithm.

target node is larger.

Figure 73 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 74 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 because the number of leaf nodes in the source map decreases (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 72) as the disjoint decomposition results in about twice as many leaf nodes, thus

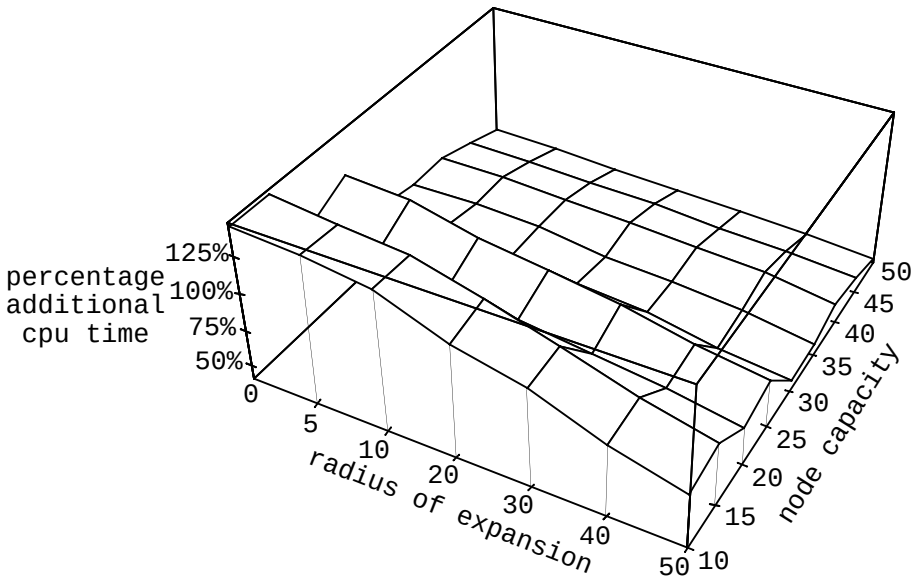


Figure 73: Percentage of additional execution time required by the leaf node intersection determination phase of the bottom-up R-tree spatial join algorithm relative to the top-down algorithm.

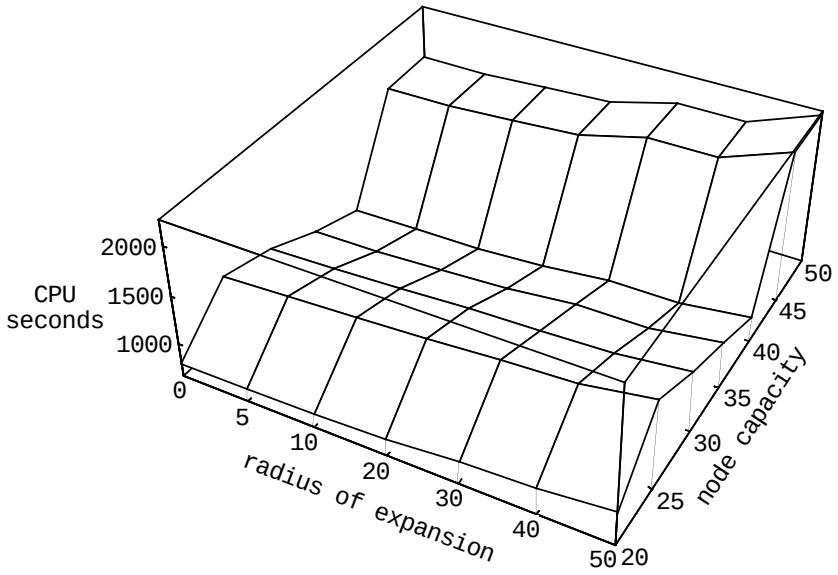


Figure 74: Execution time in seconds for the R^+-tree spatial join algorithm.

increasing the amount of source to target node communication (as well as increasing the size of the intersection lists). Finally, as is observed with the bucket 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

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

expansion radius	CPU seconds		
	PMR	R-tree	R+-tree
0	34.0	204.0	1256.0
5	34.6	205.1	1289.4
10	34.9	205.9	1324.4
20	37.0	219.6	1362.8
30	37.6	227.1	1498.4
40	39.3	235.1	1444.2
50	43.0	238.4	1575.9

performance advantages. For example, consider Table 2 which tabulates 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, 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 [Beck90].

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 54.9 cpu seconds, *regardless* of the radius of expansion. In contrast, the top-down R-tree spatial join algorithm required a minimum of 118.8 seconds for all combinations of node capacity and radius of expansion, while the bottom-up R-tree required a minimum of 151.3 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 26.4 and 55.2 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.

6 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 data-parallel domain.

References

[Ande83] D. P. Anderson. Techniques for reducing pen plotting time. *ACM Transactions on Graphics*, 2(3):197–212, July 1983.

[Aref92] W. G. Aref and H. Samet. Uniquely reporting spatial objects: Yet another operation for comparing spatial data structures. In *Proceedings of the Fifth International Symposium on Spatial Data Handling*, pages 178–189, Columbia, SC, August 1992.

[Baum72] B. Baumgart. Winged-edge polyhedron representation. Computer Science Department STAN-CS-320, Stanford University, Stanford, CA, 1972.

[Beck90] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In H. Garcia-Molina and H. V. Jagadish, editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 322–331, Atlantic City, NJ, May 1990.

[Bent75] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, September 1975.

[Best92] T. Bestul. *Parallel Paradigms and Practices for Spatial Data.* PhD thesis, University of Maryland, College Park, MD, April 1992 (also University of Maryland Computer Science Technical Report CS-TR-2897).

[Bhas88] S. K. Bhaskar, A. Rosenfeld, and A. Y. Wu. Parallel processing of regions represented by linear quadtrees. *Computer Vision, Graphics and Image Processing*, 42(3):371–380, June 1988.

[Blel88] G. E. Blelloch and J. J. Little. Parallel solutions to geometric problems on the scan model of computation. In D. H. Bailey, editor, *Proceedings of the 1988 International Conference on Parallel Processing*, volume 3, pages 218–222, St. Charles, IL, August 1988.

[Blel89a] G. E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, 38(11):1526–1538, November 1989 (also Proceedings of the 1987 International Conference on Parallel Processing, St. Charles, IL, pages 355–362, August 1987).

[Blel89b] G. E. Blelloch. *Scan Primitives and Parallel Vector Models.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, October 1989 (also Laboratory for Computer Science Technical Report MIT/LCS/TR-463).

[Bora90] 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 Transactions on Knowledge and Data Engineering*, 2(1):4–24, March 1990.

[Brin93] T. Brinkhoff, H. P. Kriegel, and B. Seeger. Efficient processing of spatial joins using R-trees. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 237–246, Washington, DC, May 1993.

[Come79] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, June 1979.

[Cope88] G. Copeland, W. Alexander, E. Boughter, and T. Keller. Data placement in Bubba. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, pages 99–108, Chicago, IL, June 1988.

[Dehn91] F. Dehne, A. G. Ferreira, and A. Rau-Chaplin. Efficient parallel construction and manipulation of quadtrees. In K. So, editor, *Proceedings of the 1991 International*

Conference on Parallel Processing, volume 3, pages 255–262, St. Charles, IL, August 1991.

[DeWi86] D. J. DeWitt, R. Gerber, G. Graefe, M. Heytens, K. Kumar, and M. Muralikrishna. GAMMA – A high performance dataflow database machine. In W. Chu, G. Gardarin, S. Onsuga, and Y. Kambayashi, editors, *Proceedings of the 12th International Conference on Very Large Data Bases*, pages 228–237, Tokyo, Japan, August 1986.

[DeWi90] D. J. DeWitt and J. Gray. Parallel database systems: The future of database processing or a passing fad? *SIGMOD Record*, 19(4):104–112, December 1990.

[DeWi92] D. J. DeWitt and J. Gray. The future of high performance database systems. *Communications of the ACM*, 35(6):85–98, June 1992.

[Edel85] S. Edelman and E. Shapiro. Quadtrees in concurrent Prolog. In D. Degroot, editor, *Proceedings of the 1985 International Conference on Parallel Processing (ICPP'85)*, pages 544–551, St. Charles, IL, August 1985.

[Elma89] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems.* Benjamin/Cummings, Redwood City, CA, 1989.

[Falo87] C. Faloutsos, T. Sellis, and N. Roussopoulos. Analysis of object oriented spatial access methods. In U. Dayal and I. Traiger, editors, *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, pages 426–439, San Francisco, CA, May 1987.

[Fole90] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics Principles and Practice.* Addison–Wesley, Reading, MA, second edition, 1990.

[Gabo76] H. N. Gabow. Using Euler partitions to edge color bipartite multigraphs. *International Journal of Computer & Information Sciences*, 5:345–355, 1976.

[Gare79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, New York, 1979.

[Ghan92] S. Ghandeharizadeh, D. DeWitt, and W. Qureshi. A performance analysis of alternative multi-attribute declustering strategies. In *Proceedings of the 1992 ACM SIGMOD*

International Conference on Management of Data, pages 29–38, San Diego, CA, June 1992.

[Günt93] O. Günther. Efficient computation of spatial joins. In *Proceedings of the Ninth IEEE International Conference in Data Engineering*, pages 50–59, Vienna, Austria, April 1993.

[Gutt84] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 47–57, Boston, MA, June 1984.

[Hill86] W. D. Hillis and G. L. Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, December 1986.

[Hoel92] E. G. Hoel and H. Samet. A qualitative comparison study of data structures for large line segment databases. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, pages 205–214, San Diego, CA, June 1992.

[Hoel93] E. G. Hoel and H. Samet. Data-parallel R-tree algorithms. In *Proceedings of the 1993 International Conference on Parallel Processing*, pages III–49–53, St. Charles, IL, August 1993.

[Hoel94a] E. G. Hoel and H. Samet. Data-parallel spatial join algorithms. In *Proceedings of the 1994 International Conference on Parallel Processing*, pages III–227–234, St. Charles, IL, August 1994.

[Hoel94b] E. G. Hoel and H. Samet. Performance of data-parallel spatial operations. In *Proceedings of the Twentieth International Conference on Very Large Data Bases*, pages 156–167, Santiago, Chile, September 1994.

[Hung89] Y. Hung and A. Rosenfeld. Parallel processing of linear quadtrees on a mesh-connected computer. *Journal of Parallel and Distributed Computing*, 7:1–27, 1989.

[Ibar93] O. H. Ibarra and M. H. Kim. Quadtree building algorithms on an SIMD hypercube. *Journal of Parallel and Distributed Computing*, 18(1):71–76, May 1993.

[Kame92] I. Kamel and C. Faloutsos. Parallel R-trees. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data (SIGMOD'92)*, pages 195–204, San Diego, CA, June 1992.

[Kasi88] S. Kasif. Optimal parallel algorithms for quadtree problems. *Computer Vision, Graphics and Image Processing*, 59(3):281–285, May 1994 (also Proceedings of the Fifth Israeli Symposium on Artificial Intelligence, Vision, and Pattern Recognition, Tel Aviv, pages 353–363, December 1988).

[Kim90] W. Kim. Research directions in object-oriented database systems. In *Proceedings of the Ninth ACM SIGACT–SIGMOD Symposium on Principles of Database Systems*, pages 1–15, Nashville, TN, April 1990.

[Krus85] C. P. Kruskal, L. Randolph, and M. Snir. The power of parallel prefix. *IEEE Transactions on Computers*, 34(10):965–968, November 1985.

[Kuck77] D. Kuck. A survey of parallel machine organization and programming. *ACM Computing Surveys*, 9(1):29–59, March 1977.

[Leig92] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures.* Morgan Kaufmann, San Mateo, CA, 1992.

[Mart86] M. Martin, D. M. Chiarulli, and S. S. Iyengar. Parallel processing of quadtrees on a horizontally reconfigurable architecture computing system. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 895–902, St. Charles, IL, August 1986.

[Mei86] G.-G. Mei and W. Liu. Parallel processing for quadtree problems. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 452–454, St. Charles, IL, August 1986.

[Nand88] S. K. Nandy, R. Moona, and S. Rajagopalan. Linear quadtree algorithms on the hypercube. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 227–229, St. Charles, IL, August 1988.

[Nels86] R. C. Nelson and H. Samet. A consistent hierarchical representation for vector data. *Computer Graphics*, 20(4):197–206, August 1986 (also *Proceedings of the SIGGRAPH'86 Conference*, Dallas, August 1986).

[Nels87] R. C. Nelson and H. Samet. A population analysis for hierarchical data structures. In U. Dayal and I. Traiger, editors, *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, pages 270–277, San Francisco, CA, May 1987.

[Oren82] J. A. Orenstein. Multidimensional tries used for associative searching. *Information Processing Letters*, 14(4):150–157, June 1982.

[Oren86] J. A. Orenstein. Spatial query processing in an object-oriented database system. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, pages 326–336, Washington, DC, May 1986.

[Pean90] G. Peano. Sur une courbe qui remplit toute une aire plaine. *Mathematische Annalen*, 36:157–160, 1890.

[Robi81] J. T. Robinson. The k–d–B–tree: A search structure for large multidimensional dynamic indexes. In *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data*, pages 10–18, Ann Arbor, MI, April 1981.

[Rose83] A. Rosenfeld, H. Samet, C. Shaffer, and R. E. Webber. Application of hierarchical data structures to geographical information systems: Phase II. Computer Science TR–1327, University of Maryland, College Park, MD, September 1983.

[Rote91] D. Rotem. Spatial join indices. In *Proceedings of the Seventh International Conference on Data Engineering*, pages 500–509, Kobe, Japan, April 1991.

[Same85b] H. Samet and R. E. Webber. Storing a collection of polygons using quadtrees. *ACM Transactions on Graphics*, 4(3):182–222, July 1985 (also Proceedings of Computer Vision and Pattern Recognition 83, Washington DC, June 1983, 127–132; and University of Maryland Computer Science Technical Report CS-TR-1372).

[Same90a] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison–Wesley, Reading, MA, 1990.

[Same90b] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS.* Addison–Wesley, Reading, MA, 1990.

[Schw80] J. T. Schwartz. Ultracomputers. *ACM Transactions on Programming Languages and Systems*, 2(4):484–521, October 1980.

[Webb84] R. E. Webber. *Analysis of Quadtree Algorithms.* PhD thesis, University of Maryland, College Park, MD, March 1984 (also University of Maryland Computer Science Technical Report CS-TR-1376).