Proceedings of the 24th International Conference on Parallel Processing (ICPP'95)

Data-Parallel Primitives for Spatial Operations*

Erik G. Hoel†
Computer Science Department
University of Maryland
College Park, Maryland 20742
hoel@cs.umd.edu

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

Abstract

Data-parallel primitives for performing operations on the PM_1 quadtree, bucket PMR quadtree, and R-tree spatial data structures are presented using the scan model. Algorithms are described for building these three data structures that make use of these primitives. The data-parallel algorithms are assumed to be main memory resident. The algorithms were implemented on a minimally configured Thinking Machines CM-5 with 32 processors containing 1GB of main memory.

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

1 Introduction

Spatial data consists of points, lines, regions, rectangles, surfaces, volumes, and even data of higher dimension which includes time. Spatial data arises in applications in many areas including computer graphics, computer vision, image processing, pattern recognition, robotics, computational geometry, solid modeling, computer-aided cartography, high energy physics, finite-element analysis, geographic information systems (GIS), and databases. The efficiency of solutions to problems in all of these areas is enhanced by the choice of an appropriate representation (see, e.g., [Same90a, Same90b]). The key issue is that the volume of the data is large. This has led to an interest in parallel processing of such data. There are two possible approaches termed image-space and object-space [Fole90]. In particular, the object-space approach assigns one processor per spatial object (e.g., [Best92, Hoel93, Hoel94a, Hoel94b]), while the image-space approach assigns one processor per region of space (e.g., [Fran90]).

In this paper our focus is on object-space data-parallel representations of spatial data. The representations which we discuss 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 into regions called *buckets*. Our presentation is for spatial data consisting of a collection of lines such as that found in road maps, utility maps, railway maps, etc. Of course, similar results could be obtained for other types of spatial data.

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. There are two principal methods: the R^+-tree [Falo87] and variants of the PM quadtree [Same85, Nels86]. The principal difference between them is that the latter is based on a regular decomposition of space while the former is not. Here we study the latter.

Prior research in the parallel spatial domain has been limited to quadtrees, k-D-trees, and R-trees. The quadtree research has primarily focussed on area (or 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 [Dehn91, Ibar93]. Other efforts have focussed on developing neighbor finding techniques [Nand88] as well as extracting region properties and performing set theoretic queries [Bhas88, Kasi88, Dehn91]. Some of the work has employed proprietary parallel architectures (e.g., two-dimensional shuffle exchange network [Mei86], or DRAFT [Mart86]), or different programming languages (e.g., Concurrent Prolog [Edel85]) while the majority has dealt with hypercube architectures. Bestul [Best92] extended the 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.

The k-D-tree [Bent75] research was limited to a small but important description of the algorithm for building the data structure for a collection of points using the scan model of computation [Blel89b]. The parallel R-tree research has been sparse and has concentrated on algorithms for single

cpu–multiple parallel disk systems [Kame92].

In this paper our focus is on the primitives that are needed to efficiently construct these representations. Our goal is one of showing the reader how the analogs of relatively simple sequential operations can be implemented in a data-parallel environment. Our presentation assumes that the data-parallel algorithms are main memory resident. Our algorithms were implemented in C* on a minimally configured Thinking Machines CM-5 with 32 processors containing 1GB of main memory (the algorithms have also been run on a 16K processor CM-2).

The rest of this paper is organized as follows. Section 2 briefly describes the spatial data structures on which we focus. Section 3 reviews a number of different parallel models of computation and points out how they deal with spatial data structures. Section 4 discusses the data-parallel primitives that are used to construct the data structures, while Section 5 presents the algorithms in terms of these primitives. Section 6 contains some concluding remarks.

2 Spatial Data Structures

In this section we review the three data structures that are discussed in the subsequent sections. We first present the PM quadtree family and explain the variants that we consider, as well as why some of them are not suitable for a data-parallel environment. This is followed by an explanation of the R-tree. In general, we often retain the original names of the data structures although a more proper description would use the qualifier *data parallel*. We do not make use of it unless the distinction needs to be emphasized in the case of a potential for misunderstanding a claim.

2.1 PM_1 Quadtree

The PM_1 quadtree [Same85] is a vertex–based member of the PM quadtree family. When inserting line segments into a region, the region is repeatedly subdivided until each resulting region contains at most a single vertex. Additionally, if a region contains a line segment vertex (or endpoint), it may not contain any portion of another line segment unless that other line segment shares a single vertex with the original line segment in the same region. For example, in Figure 1, line segments `c`, `d`, and `i` share a common endpoint which falls in the region labeled `A` of the quadtree. Do note that the large shaded region was subdivided as it contains line segments `d` and `i` (which share a common endpoint that falls outside the shaded regions).

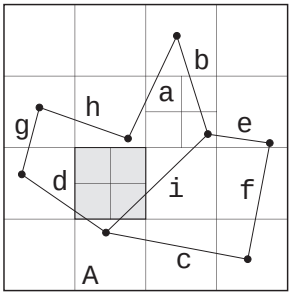


Figure 1: PM_1 quadtree for an example dataset.

The primary problem with this representation is evident when two line segments have endpoints that are very close together, resulting in a large number of subdivisions in order to separate the two endpoints (for more details of this pathological behavior, see [Nels86]). For example, consider Figure 2 where the insertion of a second line segment (line `b` in Figure 2b) results in five levels of node subdivision and the creation of fifteen new nodes in the PM_1 quadtree representation (eleven of which are empty).

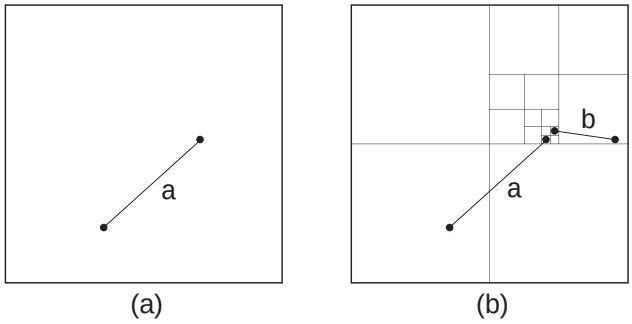


Figure 2: (a) PM_1 quadtree consisting of a single line a, and (b) the same PM_1 quadtree after the insertion of a second line segment (line b) that has a vertex that is close to one of line a's vertices.

2.2 PMR Quadtree

The PMR quadtree (for polygonal map random [Nels86, Nels87]) is an edge–based member of the PM quadtree family (see also edge-EXCELL [Tamm81]). It makes use of a probabilistic splitting rule where a block is permitted to contain a variable number of line segments. The PMR quadtree is constructed by inserting the line segments one-by-one into an initially empty structure consisting of one block. Each line segment is inserted into all of the blocks that it intersects. During this process, the occupancy of each affected block is checked to see if the insertion causes it to exceed a predetermined *splitting threshold*. If the splitting threshold is exceeded, then the block is split *once*, and only once, into four blocks of equal size. The rationale is to avoid splitting a node many times when there are a few very close lines in a block. In this manner, we avoid pathologically bad cases described in the description of the PM_1 quadtree and highlighted in Figure 2.

A line segment is deleted from a PMR quadtree by removing it from all the blocks that it intersects. During this process, the occupancy of the block and its siblings (the ones that were created when its predecessor was split) is checked to see if the deletion causes the total number of line segments in them to be less than the predetermined splitting threshold. If the splitting threshold exceeds the occupancy of the block and its siblings, they are then merged and the merging process is recursively reapplied to the resulting block and its siblings. Notice the asymmetry between the splitting and merging rules.

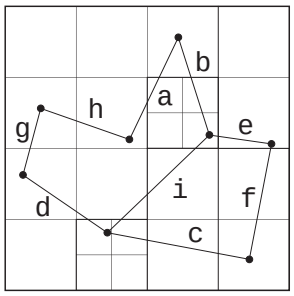


Figure 3: PMR quadtree with a splitting threshold of two for the collection of line segments of Figure 1.

Figure 3 is an example of a PMR quadtree corresponding to a set of 9 edges labeled a through i inserted in increasing order. Observe that the shape of the PMR quadtree for a given dataset is not unique; instead, it depends on the order in which the lines are inserted into it. This structure assumes that the splitting threshold value is two.

The advantage of the PMR quadtree over the PM_1 quadtree (and its vertex-based approach)

is that there is no need to subdivide in order to separate line segments that are very "close" or whose vertices are very "close" (recall Figure 2). This is important since four blocks are created at each subdivision step, and when many subdivision steps occur, many empty blocks are created, thereby leading to an increase in the storage requirements. Generally, as the splitting threshold is increased, the construction times and storage requirements of the PMR quadtree decrease while the time necessary to perform operations on it will increase.

It is interesting to point out that although a bucket can contain more line segments than the splitting threshold, this is not a problem. In fact, it can be shown that the maximum number of line segments in a bucket is bounded by the sum of the splitting threshold and the depth of the block (i.e., the number of times the original space has been decomposed to yield this block), provided that the bucket is not at the maximal depth allowed by the particular implementation of the PMR quadtree [Same90a].

2.2.1 Bucket PMR Quadtree

The shape of the PMR quadtree is dependent upon the insertion order of the input data. In the data-parallel environment, lines are inserted simultaneously during data structure construction. Thus, the ordering of the lines is unknown. Because the shape of the PMR quadtree relies upon the ordering of the data (for more details, see Section 5.2), the definition of the PMR quadtree is slightly modified to yield the bucket PMR quadtree.

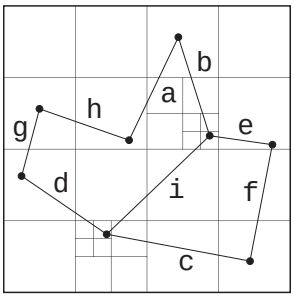


Figure 4: A bucket PMR quadtree with a bucket capacity of two and a maximal tree height of three which corresponds to the dataset shown in Figure 1.

The bucket PMR quadtree [Hoel94a] is a variant of the PMR quadtree where instead of splitting an overflowing block once, the block (or bucket) is split repeatedly until each sub-bucket contains not more than b lines (where b is the maximal bucket capacity). The motivation behind the data structure is the desire for a data structure whose shape is independent of the line segment insertion order. Note that unless the bucket capacity is greater than or equal to the maximal number of intersection lines, the recursive decomposition will continue to the maximal depth allowed by the bucket PMR quadtree. For example, consider Figure 4 where the regions corresponding to the endpoints of line `i` subdivide until the maximal depth of the quadtree (three in this case) is reached.

2.3 R-tree

The R-tree (originally used to represent collections of rectangles [Gutt84]) and its variants are designed to organize a collection of arbitrary geometric objects in d dimensions (most notably two-dimensional rectangles) by representing them as d-dimensional rectangles. Each node in the tree corresponds to the smallest d-dimensional rectangle that encloses its son nodes. The leaf nodes contain pointers to the actual geometric objects in the database, instead of sons. The objects (i.e., line segments in our case) are represented by the smallest aligned rectangle in which they are contained.

Often the nodes correspond to disk pages and, thus, the parameters defining the tree are chosen so that a small number of nodes is visited during a spatial query. Note that the bounding rectangles corresponding to different nodes may overlap. Also, a line segment may be spatially contained in several nodes, yet it is only associated with one node. This means that a spatial query may often require several nodes to be visited before ascertaining the presence or absence of a particular line segment.

The basic rules for the formation of an R-tree are very similar to those for a B-tree. All leaf nodes appear at the same level. Each entry in a leaf node is a 2-tuple of the form (R,O) such that R is the smallest rectangle that spatially contains line segment O. Each entry in a non-leaf node is a 2-tuple of the form (R,P) such that R is the smallest rectangle that spatially contains the rectangles in the child node pointed at by P. An R-tree of order (m,M) means 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 is a leaf node.

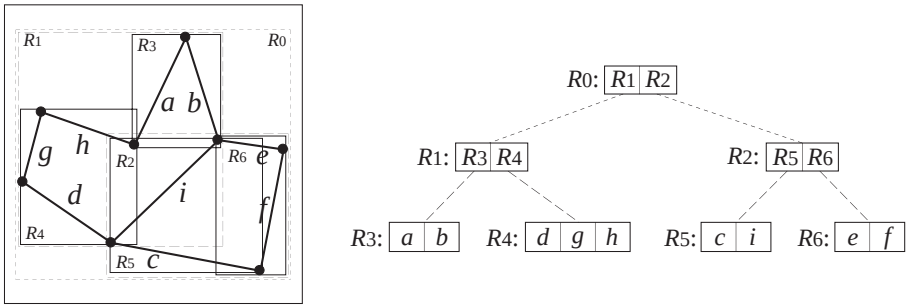


Figure 5: (a) The spatial extents of the bounding rectangles and (b) the R-tree for the example collection of line segments.

For example, consider the collection of line segments given in Figure 1. Let $M = 3$ and $m = 2$. One possible R-tree for this collection is given in Figure 5a. Figure 5b shows the spatial extent of the bounding rectangles of the nodes in Figure 5a, with broken lines denoting the rectangles corresponding to the subtrees rooted at the non-leaf nodes. Note that the R-tree is not unique. Its structure depends heavily on the order in which the individual line segments were inserted into (and possibly deleted from) the tree.

The algorithm for inserting a line segment (i.e., a record corresponding to its enclosing rectangle) in an R-tree is analogous to that used for B-trees. New line segments are added to leaf nodes. The appropriate leaf node is determined by traversing the R-tree starting at its root and at each step choosing the subtree whose corresponding bounding rectangle would have to be enlarged the least. Once the leaf node has been determined, a check is made to see if insertion of the line segment will cause the node to overflow. If yes, then the node must be split and the $M + 1$ records must be distributed in the two nodes. Splits are propagated up the tree.

There are many possible ways to split a node. One possible goal is to distribute the records among the nodes so that the likelihood that the nodes will be visited in subsequent searches will be reduced. This is accomplished by minimizing the total area of the covering rectangles for the nodes (i.e., coverage). An alternative goal is to reduce the likelihood that both nodes are examined in subsequent searches. This is accomplished by minimizing the area common to both nodes (i.e., overlap). Of course, at times these goals may be contradictory. For example, consider the four rectangles in Figure 6a. The first goal is satisfied by the split in Figure 6b, while the second goal is satisfied by the split in Figure 6c. Guttman [Gutt84] used an algorithm based on the minimization of the total area of the covering rectangles (i.e., the first of the goals described above). Beckmann [Beck90], however, employed a node splitting technique resulting in what is termed an R*-tree. This technique attempts to minimize the amount of intersection area between covering rectangles, which

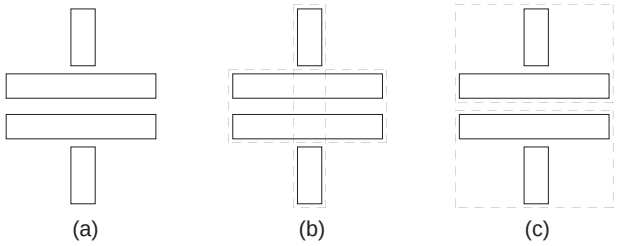


Figure 6: (a) Four rectangles and the splits that would be induced, (b) by minimizing the total area of the covering rectangles and (c) by minimizing the area of intersection between the covering rectangles of both nodes.

corresponds to the second of the previously described goals.

3 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 spatial primitive operations in Section 4.

3.1 PRAMs

An N processor Parallel Random Access Machine (or PRAM) consists of processors $P_1, P_2, \cdots, P_N$ and a global shared memory [Kuck77, Leig92]. Figure 7 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.

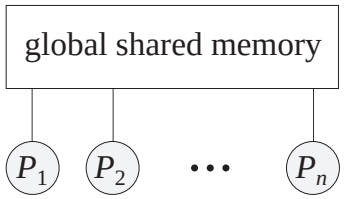


Figure 7: Simple figure representing N processors connected to a common shared memory in the PRAM model.

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 constraints 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 performance penalties [Alt87].

3.2 Scan Model

The scan model of parallel computation [Blel88, Blel89] 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 $[a_0, a_1, \cdots, a_{n-1}]$, and returns the vector $[a_0, (a_0 \bigoplus a_1), \cdots, (a_0 \bigoplus a_1 \bigoplus \cdots \bigoplus a_{n-1})]$. Blelloch [Blel90] points out that the EREW PRAM model with the scan operations included as primitives is termed the scan model. 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.

3.2.1 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 8, there are four segment groups, corresponding to segments of size 3, 4, 2, and 3.

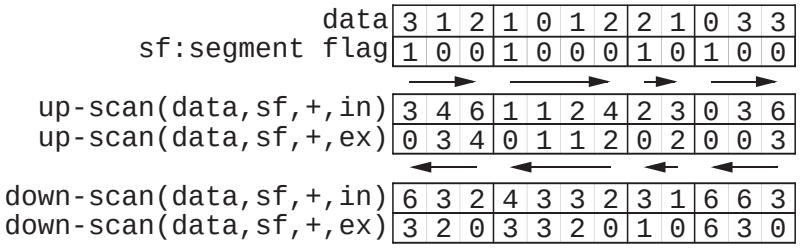


Figure 8: Example segmented scans for both the upward and downward directions (as well as inclusive and exclusive).

Finally, scan operations may be further classified as being either inclusive or exclusive. For example, an upward inclusive scan operation returns the vector $[a_0, (a_0 \bigoplus a_1), \cdots, (a_0 \bigoplus a_1 \bigoplus \cdots \bigoplus a_{n-1})]$, while an upward exclusive scan returns the vector $[0, a_0, \cdots, (a_0 \bigoplus a_1 \bigoplus \cdots \bigoplus a_{n-2})]$. Various combinations of segmented scans (where $\bigoplus$ is bound to the addition operator) are shown in Figure 8.

3.2.2 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} element of the input vectors. In Figure 9, an example elementwise addition operation is shown. `A` and `B` correspond to the two input vectors, and `ew(+,A,B)` denotes the answer vector.

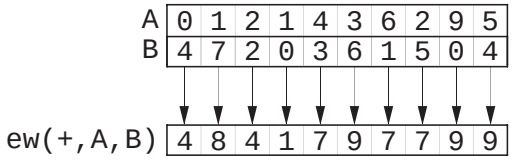


Figure 9: Example highlighting an elementwise addition operation.

3.2.3 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 10 provides an example permutation operation. `A` is the data vector, `index` is the index vector, and `permute(A,index)` denotes the answer vector.

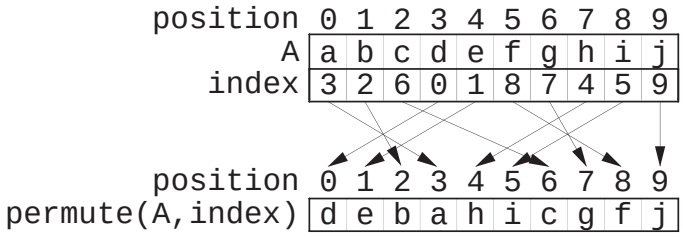


Figure 10: Example of a permutation.

3.3 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. Both within and between each linearly ordered set of processors, *monotonic mappings* may also be performed. 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 11 where the source processors are contained in processor set `A`, and the destination processors are located in processor set `B`. Figure 11a is a valid monotonic mapping, while the mapping in Figure 11b is not a monotonic mapping (as `f` comes before `c` in the linear ordering).

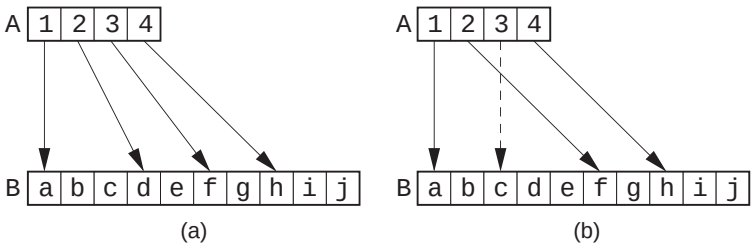


Figure 11: (a) An example monotonic mapping between two sets of processors, and (b) a similar mapping which is not monotonic.

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 5.3 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 12 where two sets of processors (set (`A,B`) and set (`C,D`) correspond to the overlapping regions in Figure 12a. 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 12b, a monotonic mapping may be maintained. The second round (depicted in Figure 12c) 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 12d.

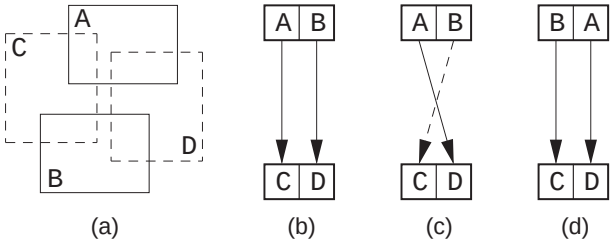


Figure 12: (a) An example collection of intersecting bounding boxes, (b) a valid monotonic mapping, (c) an invalid monotonic mapping, and (d) a valid monotonic mapping following processor reordering.

4 Data-Parallel Spatial Primitive Operations

In this section we describe the primitive operations that are needed to construct a PM_1 quadtree, bucket PMR quadtree, and an R-tree. Several of the lower-level primitives have been described elsewhere (i.e., [Nass81, Hung89]).

4.1 Cloning

Cloning (also termed *generalize* [Nass81]) is the process of replicating an arbitrary collection of elements within a linear processor ordering. Figure 13 shows an example cloning operation. Cloning may be accomplished using an exclusive upward addition scan operation, an elementwise addition, and a permutation operator.

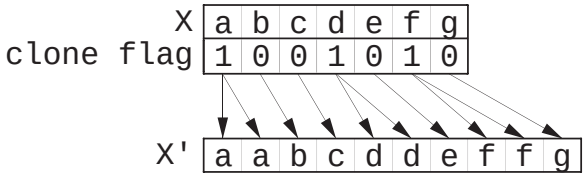


Figure 13: Example of a cloning operation.

Figure 14 details the various operations necessary to complete the cloning operation. In the figure, `clone flag` indicates which elements of `x` must be cloned; in this example, elements `a`, `d`, and `g` are to be cloned. The basic technique is to calculate the offset necessary that each existing element must be move toward the right in the linear ordering in order to make room for the new

cloned elements. This may be accomplished by employing an upward exclusive scan which sums the clone flags, as denote by `up-scan(CF,+,ex)` in the figure. After the offset has been determined, an elementwise addition on the offset value (`F1`) and the position index (`P`) determines the new position for each element in the ordering (`ew(+,P,F1)`). A simple permutation operation is then used to reposition the elements (`permute(X,F2)`). Finally, the cloning operation is completed when when each of the cloning elements copies itself into the next element in the linear ordering (denoted by the small curved arrows in the figure).

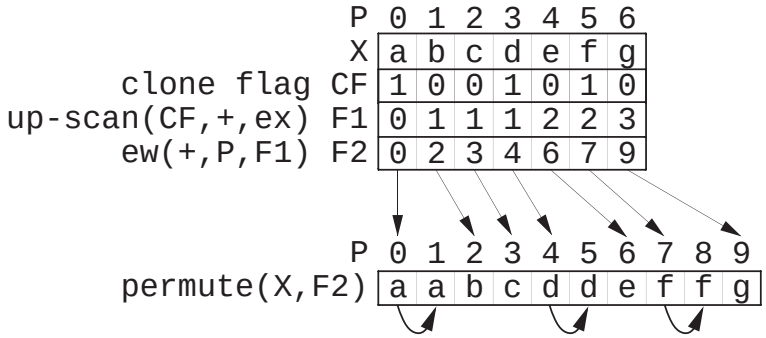


Figure 14: Example highlighting the mechanics of the cloning operation.

4.2 Unshuffling

Unshuffling is the process of physically separating two arbitrary, mutually exclusive and collectively exhaustive subsets of an original group. This operation, when applied without monotonic mappings, has also been termed *packing* [Krus85] or *splitting* [Blel89]. Unshuffling can be accomplished using two inclusive scans (one upward and one downward), two elementwise operations (an addition and a subtraction), and a permutation operator. An example unshuffling operation is shown in Figure 15.

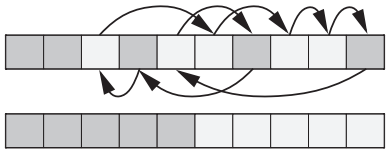


Figure 15: Example of an unshuffling operation.

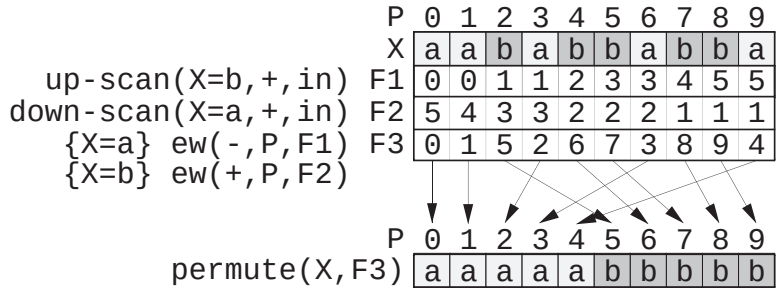


Figure 16: Example highlighting the mechanics of the unshuffle operation.

The actual mechanics of the unshuffle operation for the data of Figure 15 are illustrated in Figure 16. The two different types which must be unshuffled have type identifiers `a` and `b`. Assume

that the a's are to be repositioned toward the left, and the b's toward the right in our linear ordering. The basic technique is, for each element of the two groups, to calculate the number of elements from the other group that are positioned between itself and its desired position at either the left end or the right end. An upward inclusive scan (`up-scan(X=b,+,in)`) is used to count the number of b's between each a and the left end of the ordering. Similarly, a downward inclusive scan (`down-scan(X=a,+,in)`) is also used to count the number of a's between each individual b and the right end of the linear ordering. Once these two values are calculated, two elementwise operations are used to calculate the new position index for each element of the linear ordering. For each a element, an elementwise subtraction of the calculated number of interposed b's (F1) from the original position index P determines the new position index (`ew(-,P,F1)`). Similarly, for each b element, an elementwise addition of the calculated number of interposed a's (F2) and the original position index P determines their new position indices (`ew(+,P,F2)`). Finally, given the new position indices in F3, a simple permutation operation (`permute(x,F3)`) will reposition each element into the proper position in the linear ordering.

4.3 Duplicate Deletion

Duplicate deletion (also termed *concentrate* [Nass81]) is the process of removing duplicate entries from a sorted linear processor ordering. An example duplicate deletion (with the duplicate elements shaded) is shown in Figure 17. Duplicate deletion is accomplished using an upward exclusive scan operation, followed by a elementwise subtraction and finally a permutation operation.

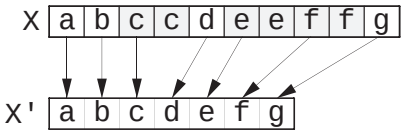


Figure 17: Example of a duplicate deletion operation.

Assuming that the elements in the linear ordering have been sorted by identifier, the basic technique employed when deleting duplicate entries is to count the number of duplicates between each element and the left side of the ordering. Each element is then moved toward the left by this number of positions. Consider Figure 18 where the elements are sorted and the duplicate items are marked (`duplicate flag`), an upward exclusive scan operation (`up-scan(DF,+,ex)`) is used to sum the number of elements in the linear ordering that are to be deleted. An elementwise operation (`ew(-,P,F1)`) is then employed to subtract the number of interposed items to be deleted (F1) from the element's position index P. This value is then used as the new position index in a simple permutation operation (`permute(X,F2)`) in completing the duplicate deletion operation.

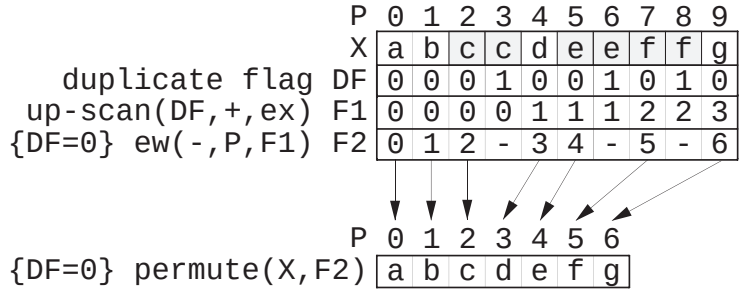


Figure 18: Example highlighting the mechanics of the duplicate deletion operation.

4.4 Node Capacity Check

For spatial decompositions such as the bucket PMR quadtree and the R-tree whose node splitting rule focuses solely on the number of items in a node, a node capacity check can be used in determining whether or not a node in the tree is overflowing and needs to be split. This can be accomplished using a downward inclusive addition scan operation, followed by an elementwise write (or read) operation. In Figure 19, the downward scan is shown for an example dataset. Following the determination of the node counts, nodes whose bucket capacity is exceeded may be marked for subdivision.

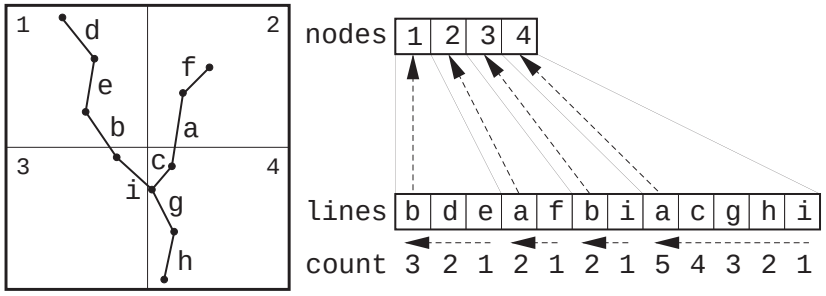


Figure 19: Example showing how a downward inclusive segmented scan operation may be employed in a node capacity check.

4.5 Determining if a PM_1 Quadtree Node Should Split

For the PM_1 quadtree, the process of determining whether or not a node should split requires more information than simply the number of lines that intersect the node. Given the maximum and minimum number of endpoints associated with all lines within a node, it is possible to determine whether or not some of the nodes must subdivide. The node must subdivide if either the maximal number of endpoints is equal to two, or if the maximal number is one and the minimal number is zero. If, however, the maximum and minimum numbers are equal to each other (i.e., 0 or 1), then additional information is necessary before the subdivision determination can be made.

The additional information that is necessary in the case of node where the maximum and minimum are both one, is whether or not a single endpoint exists within the node. If there are two or more endpoints within the node, then the node must be subdivided. This endpoint count may be determined by forming the minimal bounding box of the endpoints that lie within the node [Best92]. If the endpoint bounding box is trivially a point, then this indicates that all lines within the node share a common vertex, thus there is no need to further subdivide the node. Otherwise, the node must subdivide as there is more than one endpoint in the node.

In the case where both the minima and maxima are equal to zero, it is necessary to determine the number of lines within the node. If the number of lines within the node is greater than one, then the node must subdivide.

In parallel, each line first determines the number of its endpoints that exist within the node; either 0, 1, or 2. In Figure 20, this number is represented by the `EPs` (for endpoints) field. Using a sequence of downward inclusive segmented scan operations, the maximum and minimal number of endpoints associated with all lines within the node is determined. Figure 20 represents these values in the `min EPs` and `max EPs` fields. These two numbers are then communicated by the first line in each segment group to the corresponding node in the tree. Based upon the calculated maximum and minimum endpoint values, it can be determined that node 2 in Figure 20 must subdivide.

For the remaining nodes in the example, additional information is necessary in order to determine whether or not the node must subdivide. For nodes where the minimum and maximum number of endpoints is equal to one, the required information is whether the node contains a single endpoint that is shared among all lines in the node. This can be determined by forming the minimum

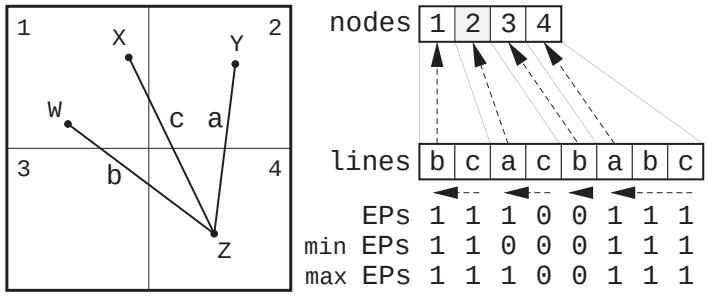


Figure 20: Initial configuration of nodes and lines. Using a sequence of downward segmented scans, the maximum and minimum number of endpoints associated with all lines in a node is determined. The grayed node is determined to require a split.

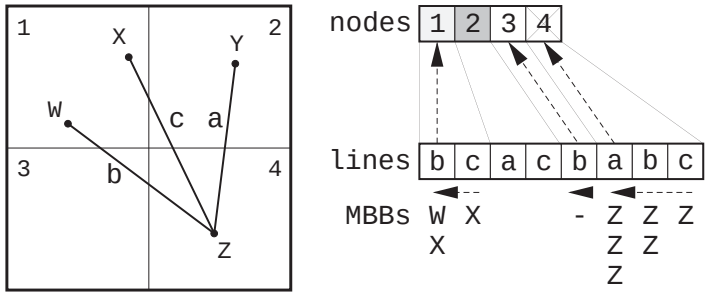


Figure 21: Calculation of the endpoint minimum bounding boxes (MBBs) for nodes where the maximum and minimum number of endpoints are equal. The dark gray node 2 was previously determined in Figure 20 to require a split, the light gray node 1 is currently determined to require a split, while the crossed node 4 does not require a subdivision.

bounding box of the endpoints that lie within the node. If the vertex bounding box is trivially a point, then this indicates that all lines within the node share a common vertex. Thus there is no need to further subdivide the node. The minimum bounding boxes can be determined using a small sequence of downward inclusive segmented scan operations. In Figure 21, the minimum bounding boxes are represented by the collection of endpoint labels (i.e., W, X, Y, and Z) beneath each line. For example, the endpoint minimum bounding box for node 1 contains endpoints X and W, while the minimum bounding box for node 4 contains only endpoint Z. Based upon the calculated bounding boxes, node 1 must subdivide, while node 4 does not need to subdivide.

In the case where both the minima and maxima are equal to zero, it is necessary to determine the number of lines within the node. If the number of lines within the node is greater than one, then it is necessary to subdivide the node. In Figure 22, the line count is calculated with a simple downward inclusive segmented scan using the addition operator. For the remaining node in question (node 3), a line count of 1 implies that the node does not need to subdivide. This final operation completes the determination of whether or not a PM_1 quadtree node must subdivide.

4.6 Splitting a Quadtree Node

The technique employed to split a quadtree node is a two stage process. After determining that a node should split, the node is first split vertically, and then horizontally. This results in the subdivision of the node into equal sized quadrants.

A node capacity check first is employed to count the number of lines associated with the node and determine whether or not the node should be split. Figure 23 depicts this process for a single

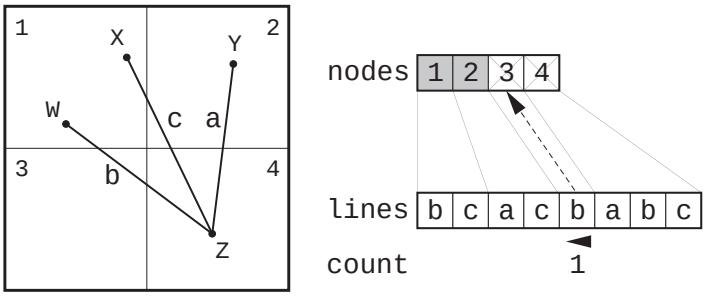


Figure 22: Calculation of the line count for the remaining node (3). Based upon the count of 1, the node is not required to subdivide. Note that previously, nodes 1 and 2 were determined to require subdivision, while node 4 did not require subdivision.

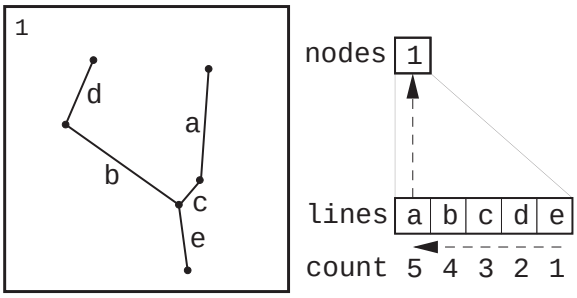


Figure 23: Example initial line to node association during a node splitting process. The node capacity check phase of the process is highlighted.

node and five associated line segments. If the number of lines associated with the node processor exceeds the predefined node capacity (4 in this example), then the node must be split into four subnodes and each of the lines must be regrouped, according to the nodes it intersects.

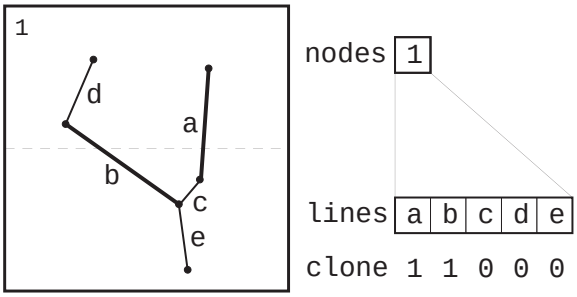


Figure 24: Determining which lines intersect the horizontal split axis and must be cloned.

Node splitting occurs in two stages, with the first stage corresponding to a vertical split of the node into two pieces. In parallel, each line in the splitting node determines whether or not it intersects the split axis. If the line intersects the split axis, it must be cloned. For the example dataset, each intersecting line (lines `a` and `b`) is shown with the `clone` value of `1`. A cloning operation, as described in Section 4.1, is then performed on the lines in the node that intersect the split axis. This is shown in Figure 24.

Once the intersecting lines have been cloned, it is necessary to regroup the lines according to whether they lie in the top or the bottom half of the splitting node. In parallel, each line may make

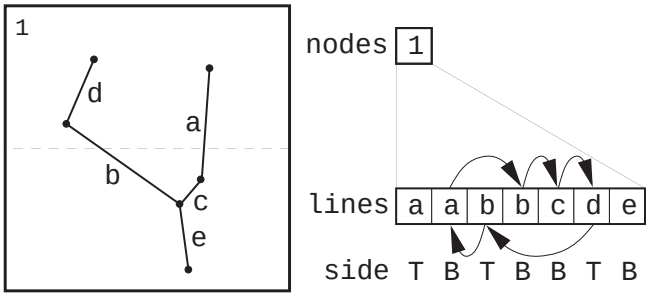


Figure 25: Following line cloning, each line in parallel determines whether it lies in the top (`T`) or bottom (`B`) half of the two resulting nodes. An unshuffle operation is then applied based upon which half the line resides in.

this determination because each line stores the size and position of the node that it resides in. In Figure 26, the `side` value represents whether the associated line is in the top (`T`) or bottom (`B`) half of the splitting node.

The regrouping of the lines is achieved with an *un-shuffle* operation as detailed in Section 4.2. The un-shuffle is used to concentrate the lines 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 splitting node. The un-shuffle operation completes the first half of the quadtree node splitting operation. The result of this un-shuffle operation is depicted in Figure 26.

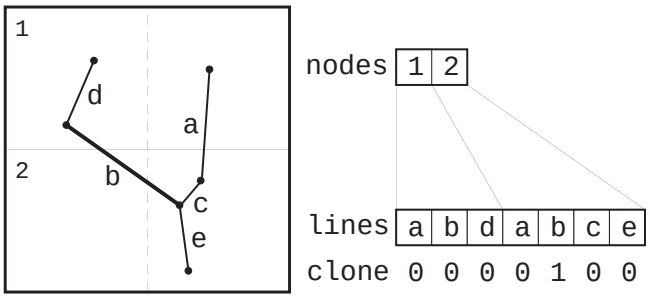


Figure 26: Result of the vertical node split. The second phase of the node split begins with each line determining whether they intersect the horizontal split axis. Intersecting lines must be cloned.

The second half of the node splitting operation uses analogous techniques in splitting the two resulting nodes again in half horizontally. This horizontal split will result in the original node depicted in Figure 23 being subdivided into four equal sized regions. The second stage begins with each line determining whether or not it intersects the horizontal split and should be cloned. In Figure 26, the intersecting lines (line `b` in node `2`) is shown with its `clone` value set to `1`.

Following the line cloning, each line in parallel determines whether it lies on the left (`L`) or right (`R`) side of the split axis. Based upon the line's position relative to the split axis, an un-shuffle operation is used on each of the two nodes in parallel to create two segment groups for each of the two splitting nodes. Each segment group will corresponds to all of the line processors which lie either in whole or in part to the left or the right of the split axis. The un-shuffle operation is shown for the example dataset in Figure 27. The result of the un-shuffle operation is depicted in Figure 28. At this point, the quadtree node splitting operation is completed.

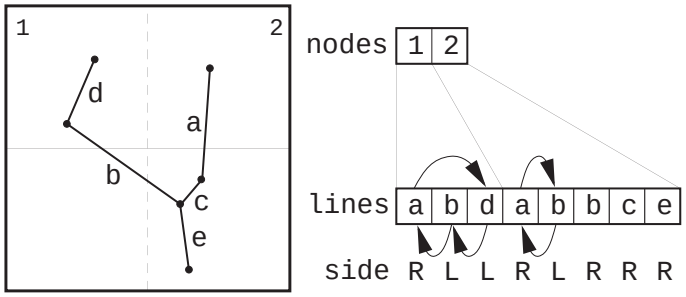


Figure 27: Following line cloning, each line in parallel determines whether it lies in the left (L) or right (R) half of the two resulting nodes. An unshuffle operation is then applied based upon which half the line resides in.

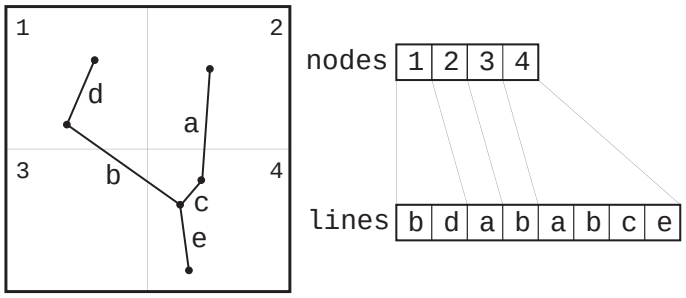


Figure 28: Final result of the example node split operation.

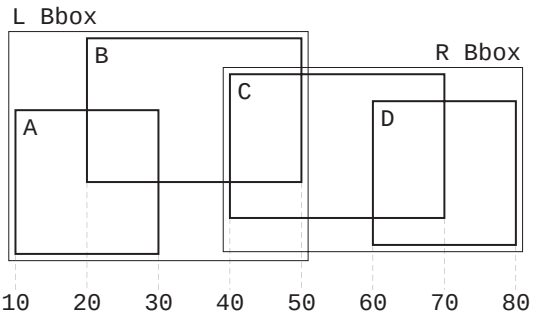
4.7 Selecting an R-tree Node Split

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. This value is then broadcast [Hung89] to all other nodes in the segment group with an upward segmented scan (using the copy operator). Each node then determines whether it lies in the left or right resulting bounding boxes. 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 bounding boxes.

The split axis and coordinate value are chosen from the two possible splits (i.e., the mean along the x-axis and the y-axis) so as to minimize the amount of area common to the two resulting nodes. This operation is of complexity $O(1)$ at each stage of the building operation as a constant number of scans dominates the computation.

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. 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. This splitting algorithm is of complexity $O(\log n)$ at each stage of the building operation as we employ two $O(\log n)$ sorts and a constant number of 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 29: Example highlighting the various scan types and their application to determining the x-coordinate values for the left and right bounding boxes.

Consider the example shown in Figure 29 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`. 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. As shown in Figure 29, 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 nodes 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 nodes left side (`L Bbox right side`). Thus, for node `B`, we see that the left and right coordinate values for the bounding box to its left (i.e., the bounding box containing nodes `A` and `B`, labeled `L Bbox` in Figure 29), have x-coordinate values `10` and `50`. These values are found in the rows of Figure 29 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 may 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`

in Figure 29) have values `40` and `80`, respectively.

After the two bounding boxes have been determined for each *legal split* (i.e., a node split 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. This splitting algorithm takes $O(\log n)$ time at each stage of the building operation as we employ two $O(\log n)$ line sorts and a constant number of scan operations.

5 Data-Parallel Build Algorithms

In this section we show how to build a PM_1 quadtree, bucket PMR quadtree, and an R-tree. The algorithms are brief and make use of the primitives described in Section 4.

5.1 PM_1 Quadtree Construction

The data-parallel PM_1-quadtree building process begins with each line assigned to a single quadtree node as depicted in Figure 30. The basic PM_1 quadtree construction is an iterative process where nodes are subdivided until their splitting criterion (refer to Sections 2.1 and 4.5 for a detailed description) is no longer satisfied.

Using the same technique as described in Section 4.5, the root node is marked for subdivision based upon the maximum number of endpoints being equal to two. The node is subdivided and the lines are split and redistributed using the quadtree node splitting method described in Section 4.6.

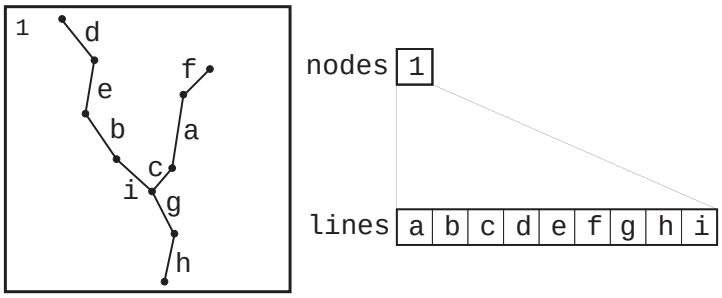


Figure 30: Initial configuration.

Following the subdivision of the root node of the PM_1 quadtree, we are left with the situation shown in Figure 31. Note that lines `a`, `b`, and `i` were cloned during this node split as they each intersected one of the split axes. This completes the first iteration of node subdivisions.

Each subsequent iteration is similar to the first: each node is first checked to see if it must subdivide, and then if necessary, the node is subdivided using the standard quadtree node splitting primitive from Section 4.6. In Figure 31, the NW, NE, and SE nodes must subdivide.

The result of the second iteration of node splitting is shown in Figure 32. At this point, one remaining subdivision must be performed on the NW child of the SE quadrant (node `10`). The final iteration results in the decomposition shown in Figure 33. Because node more nodes must be split, the PM_1 quadtree construction process is completed. For n line segments, the data-parallel PM_1 quadtree construction operation takes $O(\log n)$ time, where each of the $O(\log n)$ subdivision stages requires $O(1)$ computations (a constant number of scans, clonings, and un-shuffles).

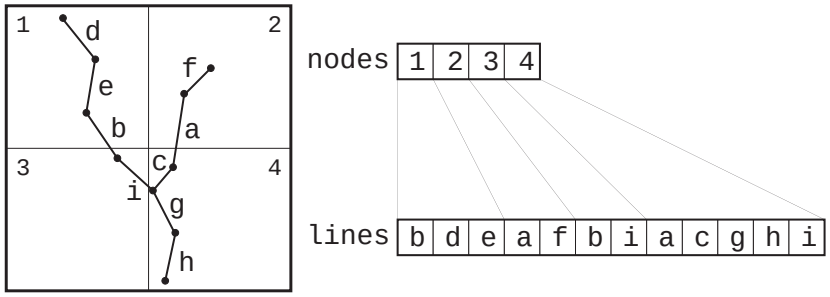


Figure 31: Result of the first round of node splitting.

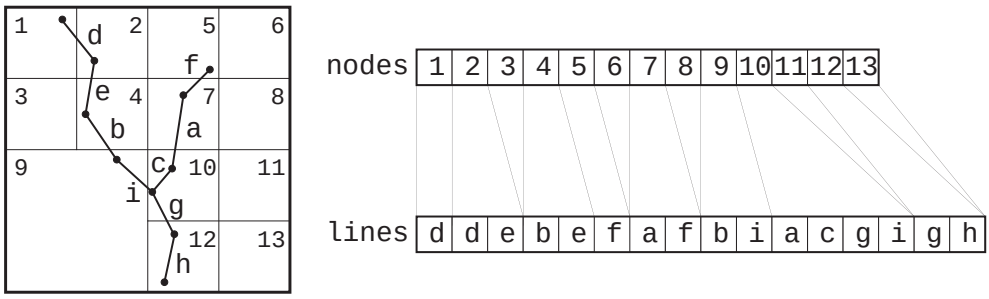


Figure 32: After second round of node splitting.

5.2 Bucket PMR Quadtree Construction

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 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 34 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 as its shape is independent of the order in which the lines are inserted and 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).

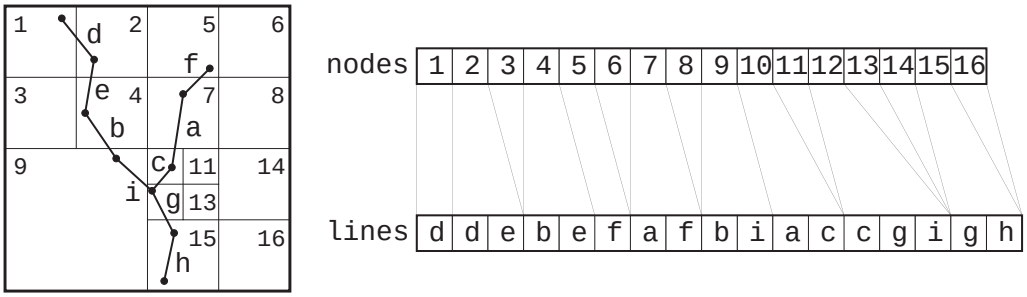


Figure 33: Final result of the PM_1 quadtree construction process for the example dataset.

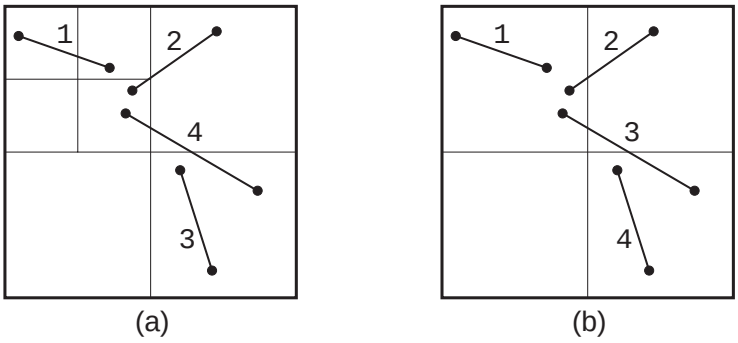


Figure 34: (a) An example PMR quadtree with a splitting threshold of two, with the lines inserted in numerical order, and (b) the resulting PMR quadtree when the insertion order is slightly modified so that line 4 is inserted before line 3.

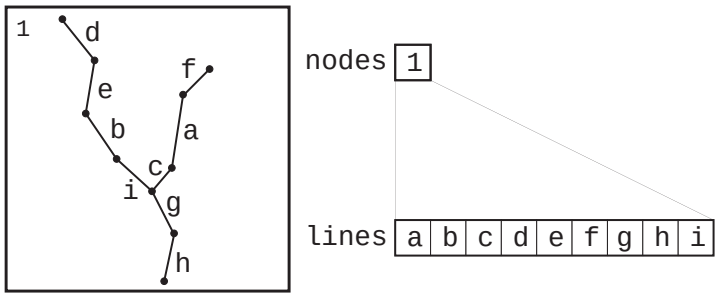


Figure 35: Initial bucket PMR quadtree processor assignments.

A bucket PMR quadtree is built in an iterative fashion, similar to that employed with the PM_1 quadtree construction algorithm. 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 35 (with the example dataset, assume we have an 8×8 quadtree of maximal height 3). The first iteration begins with the quadtree node splitting primitive as described in detail in Section 4.6. Basically, each node determines the number of lines contained in its associated segment group, and if this number exceeds the bucket capacity, the node is split using a sequence of cloning and unshuffling operations. In Figure 35, the single quadtree node 1 is subdivided as the the number of lines (9) exceeds the bucket capacity of 2 in this example. The result of the first subdivision is shown in Figure 36. Continuing with this iterative process, in Figure 36, the NW and SE nodes will subdivide, resulting in the situation depicted in Figure 37.

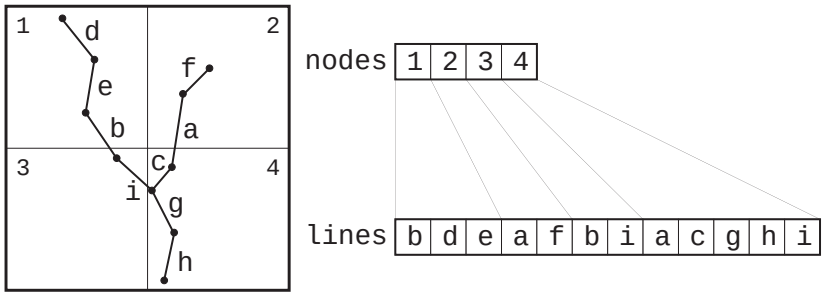


Figure 36: Result of the first node subdivision, line cloning, and un-shuffling.

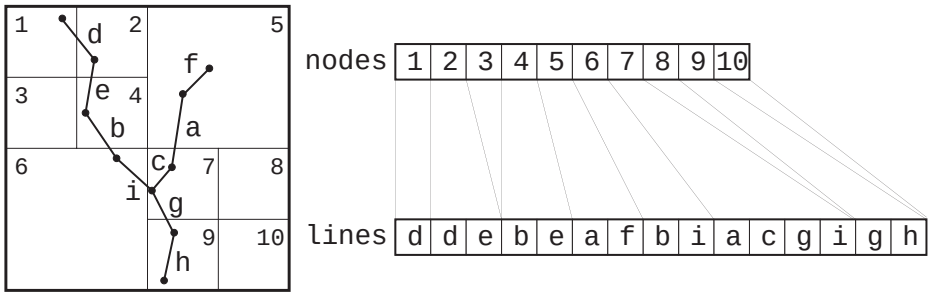


Figure 37: Result of the second round of node subdivisions.

This iterative subdivision process continues until all nodes in the bucket PMR quadtree have a line count less than or equal to the bucket capacity, or the maximal resolution of the quadtree has been reached (i.e., a node of size 1×1). This is not a problem as for practical bucket capacities (e.g., 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.

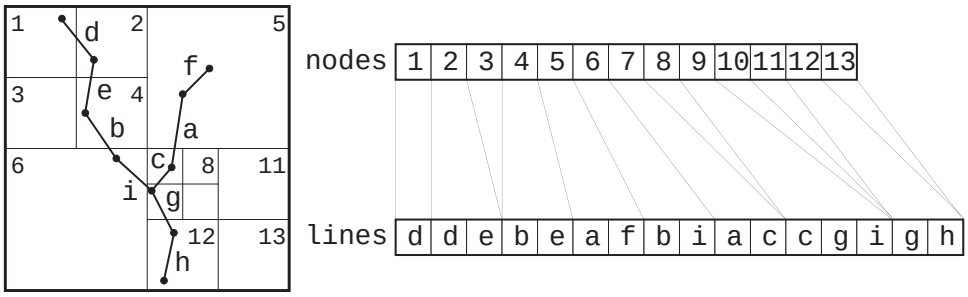


Figure 38: Result of the bucket PMR quadtree build process.

Because node 7's bucket capacity is exceeded (shown in Figure 37), 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 38. 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 further subdivided. The data-parallel bucket PMR quadtree building operation takes $O(\log n)$ time, where each of the $O(\log n)$ subdivision stages requires $O(1)$ computations (a constant number of scans and un-shuffles).

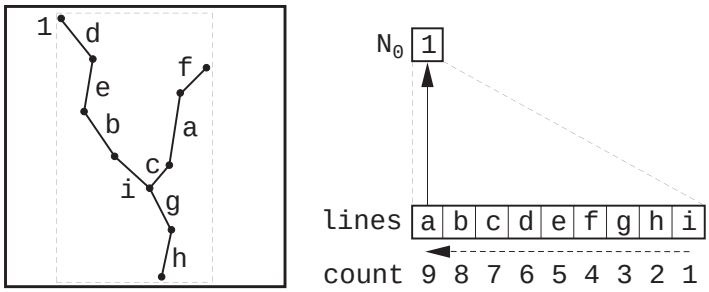


Figure 39: Initial processor assignment for the R-tree construction algorithm.

5.3 R-tree Construction

The data-parallel R-tree construction algorithm differs from the sequential R-tree algorithm as instead of 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 39. Our example assumes an order $(1,3)$ R-tree. In the figure, the label $\mathtt{N_0}$ 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 39 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 39 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.

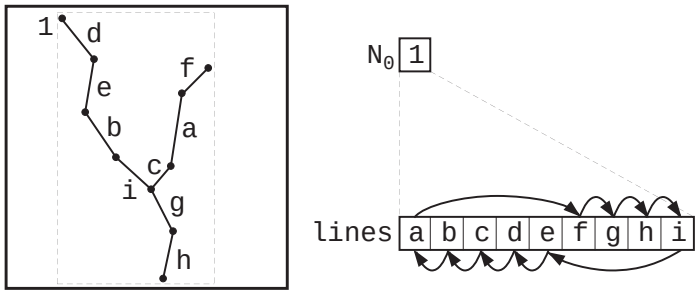


Figure 40: Un-shuffle operation.

The second of the two R-tree node splitting algorithms as detailed in Section 4.7 is used to select the splitting axis and coordinate value. Once the splitting axis and the coordinate value are chosen, an un-shuffle 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 40. 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 un-shuffle operation on the lines in Figure 40 is shown in Figure 41. 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.

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 42.

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).

The building process terminates when all nodes in the R-tree node processor set have at most

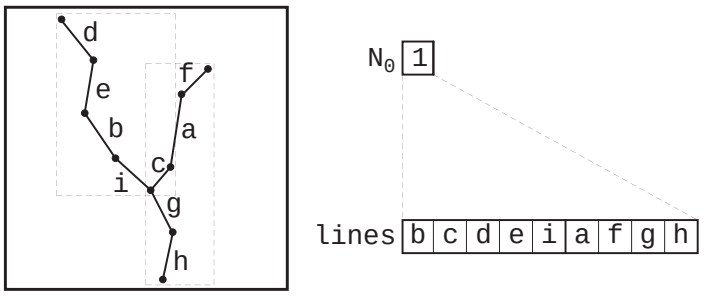


Figure 41: Result of the un-shuffle operation.

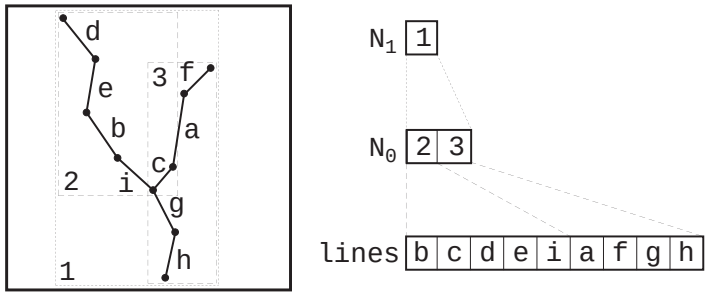


Figure 42: Completion of root node split operation.

M child processors (either internal R-tree nodes or line processors) as shown in Figure 43 for our example dataset. The data-parallel R-tree root node corresponds to the single processor in set $\mathbb{N}_2$, the leaf nodes are contained in processor set $\mathbb{N}_0$, 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).

Given n lines, the data-parallel R-tree building operation takes $O(\log^2 n)$ time, where each of the $O(\log n)$ stages requires $O(\log n)$ computations (a constant number of scans, clonings, and two sorts).

6 Conclusion

A number of data-parallel primitive operations used in building spatial data structures such as the PM_1 quadtree, bucket PMR quadtree, and the R-tree were described as well as the algorithms. These primitives have been used in the implementation of other data-parallel spatial operations such as polygonization and spatial join [Hoel93, Hoel94a, Hoel94b]. It would be interesting to see

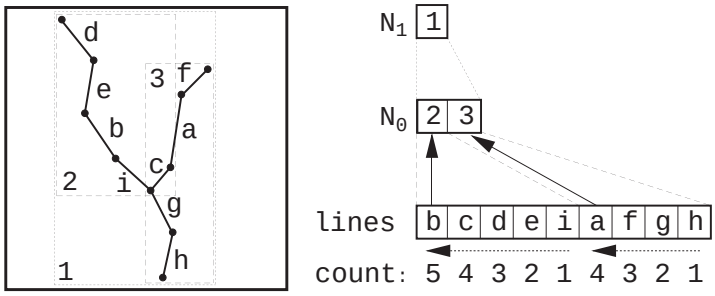


Figure 43: Broadcasting the line counts to the associated nodes.

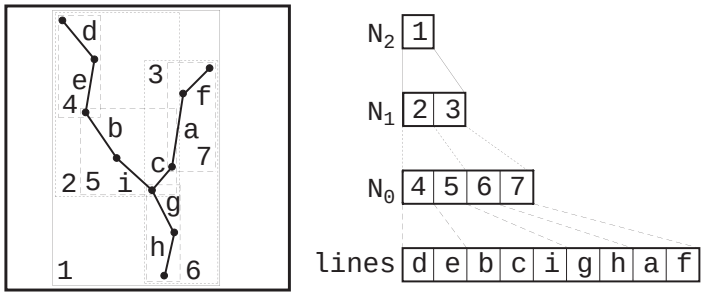


Figure 44: Completion of the data-parallel R-tree building operation.

whether these primitives are sufficient for other spatial operations and whether a minimal subset of operations can be defined. This is a subject for future research.

References

[Alt87] H. Alt, T. Hagerup, K. Mehlhorn, and F. Preparata. Deterministic simulation of idealized computers on more realistic ones. *SIAM Journal on Computing*, 16:808–835, 1987.

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

[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 (ICPP)*, volume 3, pages 218–222, St. Charles, IL, August 1988.

[Blel89] 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, 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).

[Blel90] G. E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, Cambridge, MA, 1990.

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

[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 (ICPP)*, volume 3, pages 255–262, St. Charles, IL, August 1991.

[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)*, pages 544–551, St. Charles, IL, August 1985.

[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, 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.

[Fran90] W. R. Franklin and M. Kankanhalli. Parallel object-space hidden surface removal. *Computer Graphics*, 24(4):87–94, August 1990. (also Proceedings of the SIGGRAPH'90 Conference, Atlanta, August 1990).

[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, June 1984.

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

[Hoel93] E. G. Hoel and H. Samet. Data-parallel R-tree algorithms. In *Proceedings of the 1993 International Conference on Parallel Processing (ICPP)*, 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 (ICPP)*, 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 (VLDB)*, 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*, pages 195–204, San Diego, 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).

[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 (ICPP)*, 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 (ICPP)*, 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 (ICPP)*, pages 227–229, St. Charles, IL, August 1988.

[Nass81] D. Nassimi and S. Sahni. Data broadcasting in SIMD computers. *IEEE Transactions on Computers*, C-30(2):101–107, 1981.

[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, May 1987.

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

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

[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.

[Same85] 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.

[Tamm81] M. Tamminen. The EXCELL method for efficient geometric access to data. *Acta Polytechnica Scandinavica*, 1981. (Mathematics and Computer Science Series No. 34).