Data-Parallel Primitives for Spatial Operations Using PM Quadtrees*

Erik G. Hoel
Computer Science Department
Center for Automation Research
Institute for Advanced Computer Sciences
University of Maryland
College Park, Maryland 20742

Hanan Samet
Computer Science Department
Center for Automation Research
Institute for Advanced Computer Sciences
University of Maryland
College Park, Maryland 20742

Abstract

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

1 Introduction

Spatial data consists of points, lines, regions, rectangles, surfaces, volumes, etc. Spatial data arises in applications in many areas including computer graphics, computer vision, image processing, and pattern recognition. The efficiency of solutions to problems in all of these areas is enhanced by the choice of an appropriate representation (see, e.g., [11, 12]).

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. The key issue is that the volume of the data is large. This has led to an interest in parallel processing of such data.

In this paper our focus is on the primitives that are needed to efficiently construct data-parallel members of the PM quadtree family using the scan model of parallel computation. 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).

*This work was supported in part by the National Science Foundation under grants IRI-92-16970 and BIR-93-18183, and by a grant from the Computer Research and Applications Group at Los Alamos National Laboratory.

The rest of this paper is organized as follows. Section 2 briefly describes the spatial data structures on which we focus. Section 3 reviews the scan model of parallel computation. 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. 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.

The PM_1 quadtree [13] 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 1a, 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).

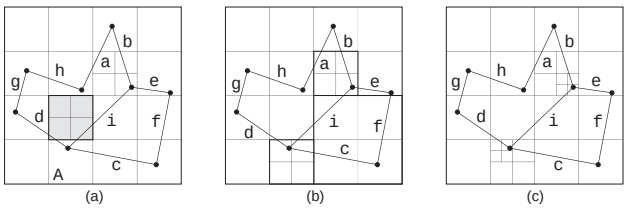

Figure 1: (a) PM_1 quadtree, (b) PMR quadtree, and (c) bucket PMR quadtree.

The PMR quadtree (for polygonal map random

[10]) is an edge–based member of the PM quadtree family. 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.

The advantage of the PMR quadtree over the PM_1 quadtree is that there is no need to subdivide in order to separate line segments that are very "close" or whose vertices are very "close". 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 needed to perform operations on it increases.

Figure 1b is an example of a PMR quadtree with a splitting threshold of two 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.

Unfortunately, in the data-parallel environment, lines are inserted simultaneously during data structure construction. Thus, the ordering of the lines is unknown. Therefore, the definition of the PMR quadtree is slightly modified to yield the bucket PMR quadtree where instead of splitting an overflowing block once, the block (or bucket) is split repeatedly until each subbucket contains no more than b lines (where b is the maximal bucket capacity). The 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 1c where the regions corresponding to the endpoints of line `i` subdivide until the maximal depth of the quadtree (three in this case) is reached.

3 Scan Model of Parallel Computation

The scan model of parallel computation [2, 3] 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 [14] takes an associative operator $\bigoplus$, a vector $[a_0, a_1, \cdots, a_{n-1}]$, and returns the vector $[a_0, (a_0 \bigoplus a_1), \cdots, (a_0 \bigoplus a_1 \bigoplus \cdots \bigoplus a_{n-1})]$. The scan model considers all primitive operations (including scans) as taking unit time on a hypercube architecture. This allows sorting operations to be performed in $O(\log n)$ time.

3.1 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.

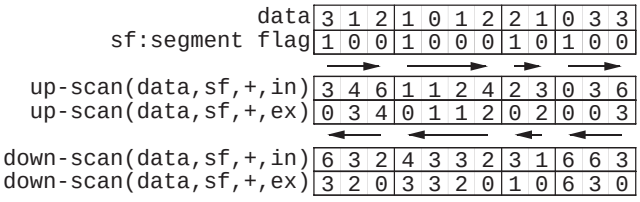

Figure 2: Segmented scans for both the upward and downward directions (as well as inclusive and exclusive).

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 2.

3.2 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 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.

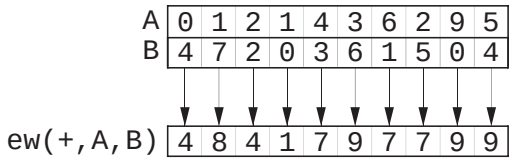

Figure 3: Example elementwise addition operation.

3.3 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.

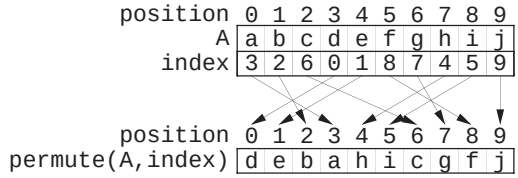

Figure 4: Example of a permutation.

4 Spatial Primitive Operations

In this section we describe the primitive operations that are needed to construct a PM_1 quadtree and a bucket PMR quadtree. Several of the lower-level primitives have been described elsewhere (i.e., [7, 9]).

4.1 Cloning

Cloning (also termed *generalize* [9]) is the process of replicating an arbitrary collection of elements within a linear processor ordering. Figure 5 shows an example cloning operation. Cloning may be accomplished using an exclusive upward addition scan operation, an elementwise addition, and a permutation operator.

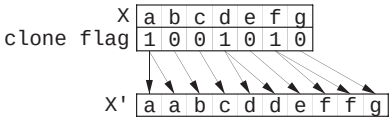

Figure 5: Example of a cloning operation.

Figure 6 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).

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* [8] or *splitting* [2]. 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 7.

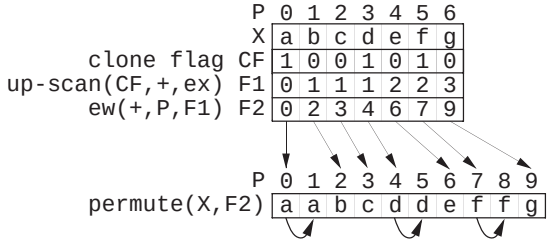

Figure 6: Mechanics of the cloning operation.

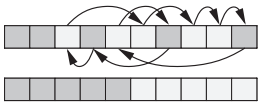

Figure 7: Example of an unshuffling operation.

The actual mechanics of the unshuffle operation for the data of Figure 7 are illustrated in Figure 8. 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.

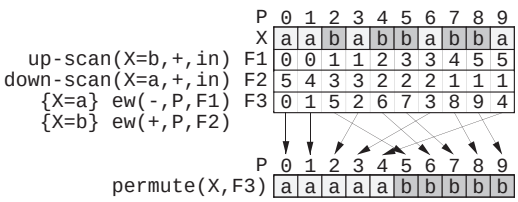

Figure 8: Mechanics of the unshuffle operation.

4.3 Duplicate Deletion

Duplicate deletion (also termed *concentrate* [9]) is the process of removing duplicate entries from a sorted linear processor ordering. Figure 9 is an example duplicate deletion (with the duplicate elements shaded). Duplicate deletion is accomplished using an upward exclusive scan operation, followed by a elementwise subtraction and finally a permutation operation.

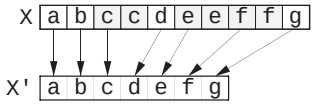

Figure 9: Example of a duplicate deletion operation.

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 10 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.

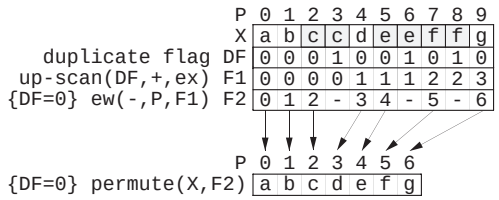

Figure 10: Mechanics of the duplicate deletion operation.

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 if 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 11, 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.

4.5 Should a PM_1 Quadtree Node 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.

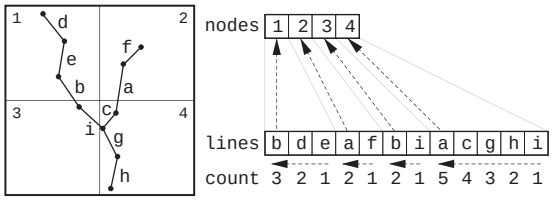

Figure 11: Example of a downward inclusive segmented scan operation being used in a node capacity check.

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 [1]. 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.

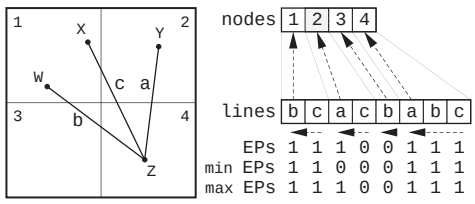

Figure 12: Initial configuration of nodes and lines. Using a sequence of downward segmented scans, the maximum and minimum number of endpoints associated with all lines in a node is determined. The grayed node is determined to require a split.

In parallel, each line first determines the number of its endpoints that exist within the node; either 0, 1, or 2. In Figure 12, 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 12 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 12 must subdivide.

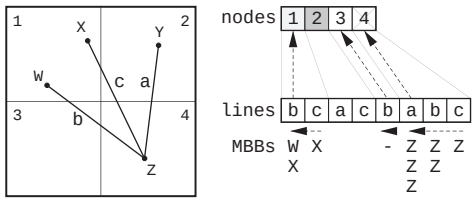

Figure 13: Calculation of the endpoint minimum bounding boxes (MBBs) for nodes where the maximum and minimum number of endpoints are equal. The dark gray node 2 was previously determined in Figure 12 to require a split, the light gray node 1 is currently determined to require a split, while the crossed node 4 does not require a subdivision.

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 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 13, 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.

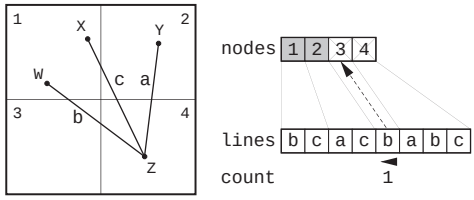

Figure 14: Calculation of the line count for the remaining node (3). Based upon the count of 1, the node is not required to subdivide. Note that previously, nodes 1 and 2 were determined to require subdivision, while node 4 did not require subdivision.

When 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 14, 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.

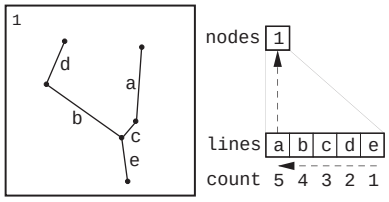

Figure 15: Example initial line to node association during a node splitting process. The node capacity check phase of the process is highlighted.

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 15 depicts this process for a single 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.

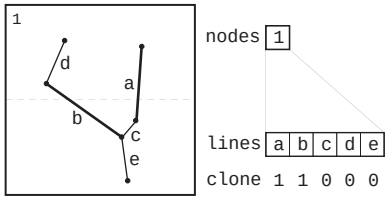

Figure 16: Determining which lines intersect the horizontal split axis and must be cloned.

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 16.

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

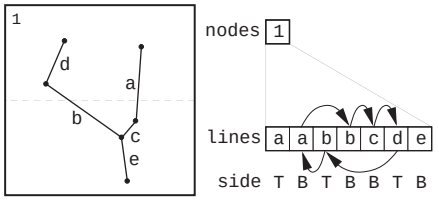

Figure 17: Following line cloning, each line in parallel determines whether it lies in the top (`T`) or bottom (`B`) half of the two resulting nodes. An unshuffle operation is then applied based upon which half the line resides in.

node. In parallel, each line may make this determination because each line stores the size and position of the node that it resides in. In Figure 18, the `side` value represents whether the associated line is in the top (`T`) or bottom (`B`) half of the splitting node. 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 18.

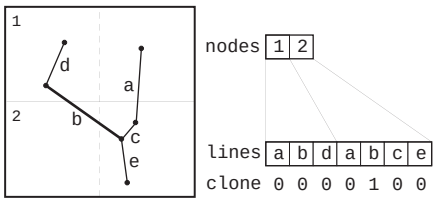

Figure 18: Result of the vertical node split. The second phase begins with each line which intersects the horizontal split axis being cloned.

The second half of the node splitting operation uses analogous techniques in splitting the two resulting nodes again in half horizontally. This horizontal split results in the original node depicted in Figure 15 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 18, the intersecting line (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 19. The result of the un-shuffle operation is depicted in Figure 20. At this point, the quadtree node splitting

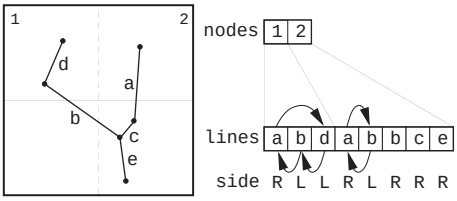

Figure 19: Following line cloning, each line in parallel determines whether it lies in the left (`L`) or right (`R`) half of the two resulting nodes. An unshuffle operation is then applied based upon which half the line resides in.

operation is completed.

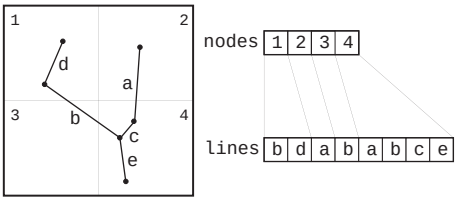

Figure 20: Final result of the node split operation.

5 Data-Parallel Build Algorithms

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

5.1 PM_1 Quadtree Construction

Building a data-parallel PM_1-quadtree begins with each line assigned to a single quadtree node as depicted in Figure 21. The basic PM_1 quadtree construction is an iterative process where nodes are subdivided until their splitting criterion (refer to Sections 2 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.

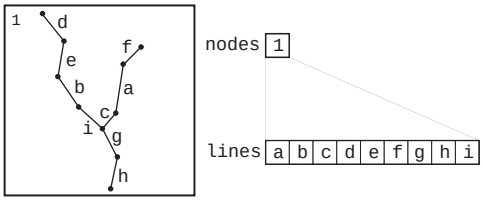

Figure 21: Initial configuration.

Following the subdivision of the root node, we are left with the situation shown in Figure 22. 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.

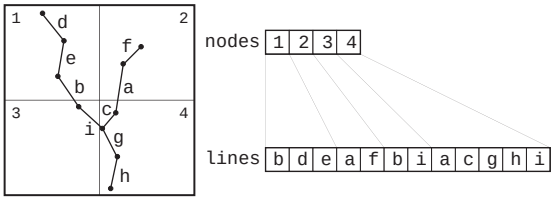

Figure 22: Result of the first round of node splitting.

Each subsequent iteration is similar to the first: each node is first checked to see if it must subdivide, and then if needed, subdivide the node using the quadtree node splitting primitive from Section 4.6. In Figure 22, the NW, NE, and SE nodes must subdivide.

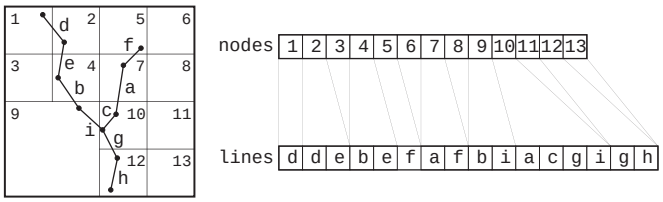

Figure 23: After second round of node splitting.

The result of the second iteration of node splitting is shown in Figure 23. 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 24. 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).

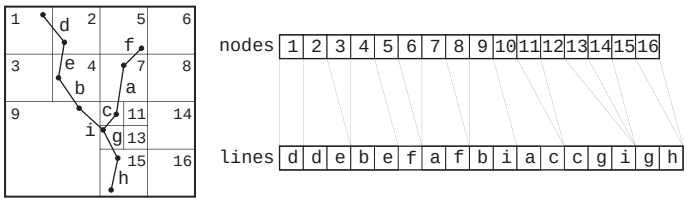

Figure 24: Result of the PM_1 quadtree build process.

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 25 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).

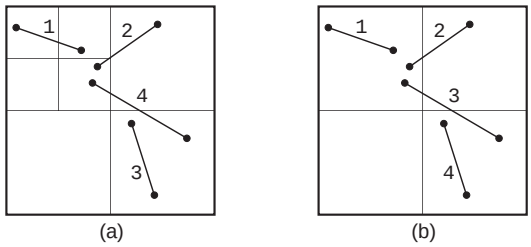

Figure 25: (a) An example PMR quadtree (splitting threshold of 2), with the lines inserted in numerical order, and (b) the resulting PMR quadtree when the insertion order is slightly modified so line 4 is inserted before line 3.

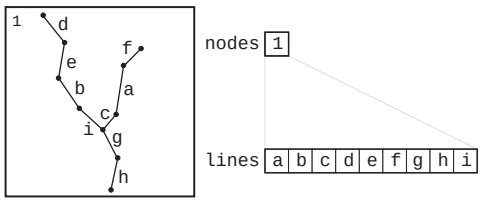

Figure 26: Initial PMR quadtree processor assignments.

A bucket PMR quadtree is built in an iterative fashion, similar to 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 26 (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 26, 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 27. Continuing with this iterative process, in Figure 27, the NW and SE nodes will subdivide, resulting in the situation depicted in Figure 28.

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.,

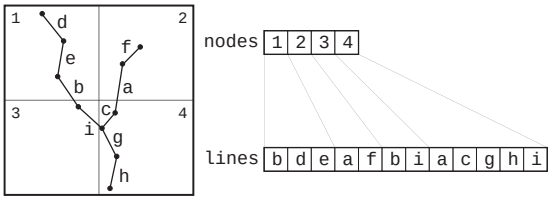

Figure 27: Result of the first node subdivision.

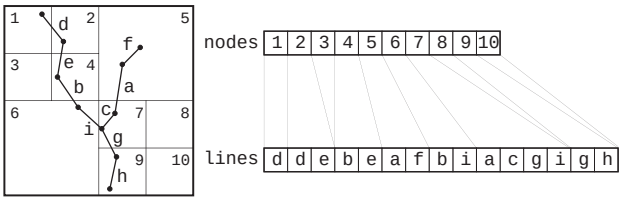

Figure 28: Result of the second node subdivisions.

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.

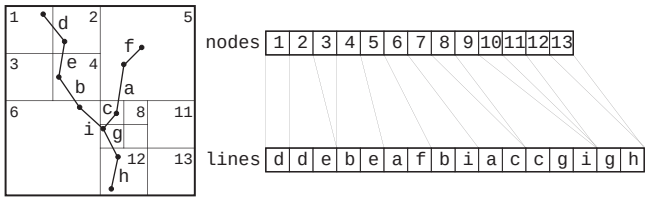

Figure 29: Result of the PMR quadtree build process.

Since node 7's bucket capacity is exceeded (see Figure 28), and the maximal resolution has not yet been reached, another round of subdivision is needed. The result of the third and final subdivision for our example data set is shown in Figure 29. 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).

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 [4, 5, 6]. It would be interesting to see 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

[1] T. Bestul. *Parallel Paradigms and Practices for Spatial Data*. PhD thesis, University of Maryland, College Park, MD, Apr. 1992.

[2] G. E. Blelloch. Scans as primitive parallel operations. *IEEE Trans. on Computers*, 38(11):1526–1538, Nov. 1989.

[3] G. E. Blelloch and J. J. Little. Parallel solutions to geometric problems on the scan model of computation. In *Proc. of the 1988 Intl. Conf. on Parallel Processing*, volume 3, pages 218–222, St. Charles, IL, Aug. 1988.

[4] E. G. Hoel and H. Samet. Data-parallel R-tree algorithms. In *Proc. of the 1993 Intl. Conf. on Parallel Processing*, volume 3, pages 49–53, St. Charles, IL, Aug. 1993.

[5] E. G. Hoel and H. Samet. Data-parallel spatial join algorithms. In *Proc. of the 1994 Intl. Conf. on Parallel Processing*, volume 3, pages 227–234, St. Charles, IL, Aug. 1994.

[6] E. G. Hoel and H. Samet. Performance of data-parallel spatial operations. In *Proc. of the 20th Intl. Conf. on Very Large Data Bases*, pages 156–167, Santiago, Chile, Sept. 1994.

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

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

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

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

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

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

[13] H. Samet and R. E. Webber. Storing a collection of polygons using quadtrees. *ACM Trans. on Graphics*, 4(3):182–222, July 1985.

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