Counting and Reporting Red/Blue Segment Intersections

Larry Palazzi   Jack Snoeyink*
Department of Computer Science
University of British Columbia
2366 Main Mall
Vancouver, B.C. V6T 1Z4 Canada
palazzi@cs.ubc.ca
snoeyink@cs.ubc.ca

Abstract

We simplify the red/blue segment intersection algorithm of Chazelle et al: Given sets of n disjoint red and n disjoint blue segments, we count red/blue intersections in $O(n \log n)$ time using $O(n)$ space or report them in additional time proportional to their number. Our algorithm uses a plane sweep to presort the segments; then it operates on a list of slabs that efficiently stores a single level of a segment tree. With no dynamic memory allocation, low pointer overhead, and mostly sequential memory reference, our algorithm performs well even with inadequate physical memory.

1 Introduction

Geographic information systems frequently organize map data into various layers. Users can make custom maps by overlaying roads, political boundaries, soil types, or whatever features are of interest to them. The ARC/INFO system [8] is organized around this model; even a relatively inexpensive database like the Digital Chart of the World [9] contains seventeen layers, several with sublayers. An algorithm for map overlay must be able to handle large amounts of data and compute the overlay quickly for good user response performance.

We consider a geometric abstraction of the map overlay problem. Suppose $\mathcal{R}$ is a set of red line segments in the plane and $\mathcal{B}$ is a set of blue segments such that no interiors of segments of the same color intersect. The *red/blue segment intersection problem* asks for an efficient algorithm to count or report the red/blue intersections.

Chazelle et al. [3] give output-sensitive solutions for this problem, meaning that the running time of their algorithms depends on the amount of output. They outlined relatively simple algorithms that run in $O(n \log^2 n + K)$ time and use $O(n \log n)$ space, where K is the number of intersections for the reporting problem and $K = 1$ for the intersection counting problem. We describe their method in section 2. They also state that the space can be reduced to linear by streaming [7] and the time to $O(n \log n + K)$ by a dynamic form of fractional cascading [4, 5], which they admit is

*Supported in part by an NSERC Research Grant

complicated. This paper presents an alternative way to reduce space that yields a much simpler approach to reducing the time.

The red/blue intersection problem was first considered while researchers were searching for general output-sensitive line segment intersection routines. Shamos and Hoey [13] gave a plane-sweep algorithm to detect an intersection in $\Theta(n \log n)$ time. Bentley and Ottmann [1] turned their algorithm into a general intersection reporting procedure that runs in $O((n+K)\log n)$ time and uses linear space. Mairson and Stolfi [11] applied plane-sweep to the red/blue intersection problems and obtained $O(n)$ space algorithms for reporting in $O(n \log n + K)$ time and for counting in $O(n \log n + (nK)^{1/2})$ time. Chazelle and Edelsbrunner [2] finally gave an output-sensitive algorithm for the general intersection problem with optimal running time: their algorithm runs in $O(n \log n + K)$ and uses $O(n+K)$ space. One can, of course, use these general routines to report red/blue intersections if one is willing to pay the time penalty of Bentley-Ottmann or the space penalty of Chazelle and Edelsbrunner. They do not, however, adapt to efficiently solve the intersection-counting problem.

2 Preliminaries

The *hereditary segment tree* data structure, which stores the set $S = \mathcal{R} \cup \mathcal{B}$ of red and blue segments, forms the basis of the red/blue intersection procedure of Chazelle et al. [3]. To define the hereditary segment tree, we must first define the *segment tree* [12]. Our definition is slightly non-standard—we use midpoints instead of endpoints to define vertical slabs and allow several segments of the same color to end in a slab.

Let $\{x_1, x_2, ..., x_k\}$ be the set of distinct x-coordinates of segment endpoints in increasing order. We make three "general position assumptions" that simplify the description of the algorithm and data structures: First, no red or blue endpoint lies on an oppositely colored segment. Second, no red/blue intersection point has an x-coordinate x_i, for $1 \leq i \leq k$. Third, all segments with an endpoint on the line $x = x_i$ have the same color. We will remove these assumptions in section 3.4.

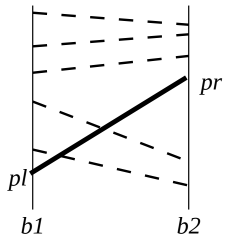

Figure 1: Intersections in a slab

Now, form the set of midpoints $M = \{m_1, m_2, ..., m_{k+1}\}$, where $m_1 = -\infty$, $m_{k+1} = \infty$ and $m_i = (x_{i-1} + x_i)/2$, for $1 < i \leq k$. Then form a balanced binary tree on k leaves such that the ith leaf node ν_i is associated with the *leaf slab* $s(\nu_i)$ of all points whose x-coordinates lie in the halfopen interval $[m_i, m_{i+1})$. Notice that the leaf slab $s(\nu_i)$ contains endpoints of at most one color. Each internal node ν' is associated with the slab $s(\nu')$ that is the union of the leaf slabs in the subtree rooted at ν'.

Now, let us look at the relation of the red and blue segments to the slab $s(\nu)$ of an internal or leaf node ν. Some segments may end in $s(\nu)$; we call them *short* in ν and store them in red or blue short lists in ν depending on their color. Others, which we call *long*, cut completely through $s(\nu)$; if a segment σ cuts through $s(\nu)$ and not through the parent's slab $s(parent(\nu))$, then store σ in the red or blue *long* list for ν.

Lemma 2.1 *On each level of the tree, a segment is stored in at most two short lists and two long lists.*

Proof: The slabs stored at any given level of the tree are disjoint. A segment, σ, is stored as *short*, therefore, in the at most two slabs that contain its endpoints. σ is stored as *long* in at most one child of each node where σ is stored as *short*. ■

We can efficiently compute the intersections between long red segments and blue segments in the slab $s(\nu)$. To begin, sort the long red segments vertically within the slab $s(\nu)$. Then clip each blue segment to the slab and locate the endpoints of each clipped blue segment in the red long list by binary search. The red segments that a clipped blue segment intersects are exactly those between the blue endpoints—one can report them in time proportional to their number or count them in constant time by subtracting the ranks of the segments above the blue endpoints (see figure 1). Similarly, one can report the intersections of short red segments with long blue segments that appear in the slab.

If we perform this procedure for every tree node ν—reporting the intersections between the long red and long and short blue segments and the long blue and short red segments—then we can show that every intersection is reported exactly once.

Lemma 2.2 *Every intersection point is the intersection of a long segment and another segment in exactly one slab.*

Proof: Consider an intersection point of a red segment r and a blue segment b, namely $p = r \cap b$. In the leaf slab that contains p, there are short segments of at most one color, so either r or b must be long. Assume that b is long, and if r is also long assume that r is not stored at a higher level than b.

Let ν be the node that stores b as *long*. By the assumptions, r is stored either as long or short at ν, so the intersection point p will be reported at ν. Since the portion of b containing p is stored as long only at ν and the portion of r is not stored as long above ν, the point p is reported only at ν. ■

How much space and time is taken by this procedure, excluding the amount used to report output? If we construct the entire segment tree, each segment is stored in at most four slabs per level by lemma 2.1, so the total space is $O(n \log n)$. In each slab we sort long segments and locate long and short segments; both can be done in $O(\log n)$ time per segment. This gives a total of $O(n \log^2 n)$ time.

This algorithm and analysis is contained in Chazelle et al. [3]. They also state that one can remove the logarithmic factor from the space by the technique of *streaming* [7]: rather than building the entire segment tree, one builds the succession of root to leaf paths, starting with the path to the leftmost leaf and ending with the path to the rightmost leaf. In moving from one path to the next only the nodes that change need to be recomputed. They also state that a logarithmic factor can be removed from the time bound by using a dynamic form of fractional cascading [4]: because each endpoint will be located in $O(\log n)$ lists, sharing elements between the lists allow repeated searches to be performed more efficiently.

In the next section we develop an alternative way of reducing the space to linear that gives an easier way to reduce the time to $O(n \log n)$. Our approach actually eliminates the segment tree data structure and replaces it by a linear list of slabs. This is an advantage because the overhead of a segment tree may not be negligible in practice. Furthermore, our approach can benefit from preprocessing the data into a special sorted order. Finally, the sequential nature of processing results in localized memory references, which reduces memory swapping and allows running of large numbers of segments. Our approach has been parallelized recently by Devillers and Fabri [6].

3 The Improved Algorithm

We have already made a key observation that allows us to reduce the space to linear. Since by lemma 2.1 each segment appears in at most four lists in each level, we can store a complete level of the segment tree in linear space. Thus, rather than streaming, we will compute the tree level by level.

An extra logarithmic factor enters the time complexity of the algorithm of Chazelle et al. [3] in two ways: sorting the segments in the long lists within each slab and locating in these long lists the endpoints of segments that are clipped to the slab. We can do most of the sorting and point location in advance: Define the *aboveness* relation on sets in the plane: $A \succ B$ if there are points $(x, y_A) \in A$ and $(x, y_B) \in B$ with $y_A > y_B$ (see figure 2). When applied to disjoint convex sets, the aboveness relation is a partial order [12].

Lemma 3.1 *The aboveness relation for disjoint convex sets in the plane is acyclic.*

Proof: Suppose that $A_1 \succ A_2 \succ \cdots \succ A_k \succ A_1$ is a cycle of minimum length. We will derive a contradiction. (Note that $k > 2$.)

Let $x_0 = \min_{\forall j} \max \{x \mid (x, y) \in A_j\}$ and let A_i be a set that attains x_0 as in figure 2. The line $x = x_0$ must intersect A_{i-1} above A_i above A_{i+1} since A_{i-1} and A_{i+1} are comparable to A_i. Omitting A_i, therefore, gives a smaller cycle, which contradicts minimality. ∎

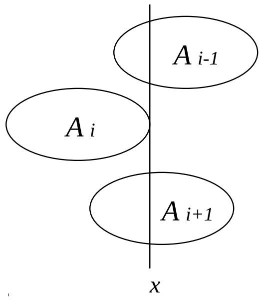

Figure 2: $A_{i-1} \succ A_i \succ A_{i+1}$ and $A_{i-1} \succ A_{i+1}$.

On the set of red segments and blue endpoints, we can extend the partial order defined by $\succ$ to a total order. In section 3.2 we describe how to do this efficiently using a simple sweep algorithm. If we add the red segments and blue endpoints to a level of the tree according to this order, two things happen automatically: 1) in each slab, the long red segments are inserted in sorted order, and 2) when a blue endpoint appears in a slab then the segment immediately above it was the last to be added to the slab. Thus, the sorting of long red segments can be omitted and (original) endpoint location is a simple matter of looking at the last long red segment added to the slab containing the blue endpoint.

The only task that remains is locating the clipped ends of blue segments. If we also use a total order of the set of blue segments and red endpoints to insert blue segments into slabs, then we obtain the ends of clipped blue segments in sorted order along the slab boundaries. Merging these blue endpoints with the long red list gives us the ranks of all clipped blue endpoints, in time proportional to the number of long segments and endpoints.

In section 3.1, we describe the data structure requirements for our algorithm. Section 3.2 outlines the sweep algorithm for pre-sorting the segments and points. Section 3.3 outlines the intersection algorithm. Section 3.4 discusses how to handle degenerate cases (in sections 3.2 and 3.3 we will assume that no degeneracies exist in our data).

3.1 Data Structure Requirements

We define a global structure for storing information on each color. We need one structure for *red* information and one for *blue* information. In each structure, we store the following: the number of points (twice the number of segments), the list of segments stored as point pairs, the list of endpoints sorted by x-coordinate (used by the sorting phase) and the sorted list of segments and endpoints (created by the sorting phase and passed to the intersection phase). To store point information, we define a structure that holds the (x, y) coordinate of each point p, the index to the current slab containing p, a count of the number of long segments above p and a pointer to be used in a linked list of point structures.

During the sorting phase of the algorithm, we use two tree structures. The first tree, which we call the *search tree*, maintains the segments that currently intersect the sweep line in sorted order by aboveness. We use this tree for finding the predecessor of the current point being swept—that is, the segment directly above the point. The second tree, which we call the *sweep tree*, is built during the sweep by making each segment a child of the segment immediately above its right endpoint. When the sweep is complete, the sweep tree holds the set of segments and endpoints so that an inorder traversal gives a total ordering consistent with the aboveness relation.

Figure 3 illustrates a sweep tree. We modify the standard trick of using child and sibling pointers to represent a higher degree tree as a binary tree [10, p. 333], and use pointers to the rightmost child and left sibling. The segment d is the child of c (because its right endpoint is below c), segments b and c are siblings, and c (along with b) is the child of a. The data structures for both trees are the standard structures for binary trees. Each node has a left (sibling) pointer and a right (child) pointer. The search tree and sweep tree structures are needed during the sorting phase only. A segment in the search tree also has a pointer to its location in the sweep tree.

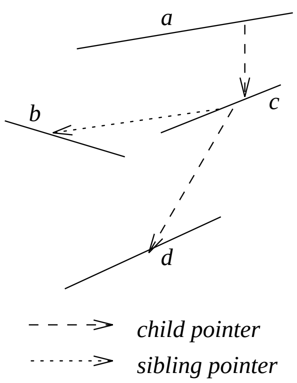

Figure 3: Sweep Tree

For the intersection phase, we define a structure to store the slab information. For each slab, we define two head pointers to linked lists storing the long segments on the left and right boundaries of the slab. We also store two counts for the number of long segments above points on the left and on the right boundaries of the slab. A list of these structures represents the list of slabs at the current level of the segment tree. Our convention is to number the slabs starting from zero so that each *even/odd* pair of slabs represents the two nodes in the segment tree that will be merged together in the next level of the segment tree. To form the actual slab boundaries we need the list of midpoints—the points between x-coordinates of endpoints as described in section 2.

3.2 The Sweep Algorithm For Pre-Sorting

We initialize the sweep tree to a node H containing a horizontal line from $(-\infty, \infty)$ to (∞, ∞) so that all segments and endpoints will be below this line. Before the sweep begins, a node for each segment and endpoint is pre-allocated. As the sweep proceeds, these nodes are linked together forming a forest of trees. Eventually, all of these subtrees will be linked to H, forming the final sweep tree. The head node, H, will then have the entire list of segments and endpoints as its child subtree and will not have any siblings. The total sorted order of the segments and points can then be recovered by traversing this tree in inorder.

Next, the sets $\mathcal{R}$ and $\mathcal{B}$ are each sorted individually by the smaller x-coordinate of each segment. The endpoints in these lists will be swept from left to right by increasing x-coordinate. We then call this sorting routine once for the red segments and blue endpoints, and once for the blue segments and red endpoints, creating two sorted lists. We describe the procedure only for red segments and blue endpoints.

The sweep begins with the line $x = -\infty$ that intersects only the dummy red segment from $(-\infty, \infty)$ to (∞, ∞). When the first endpoint, p, of a red segment is encountered, we insert the segment into the search tree. When the second red endpoint, q, is encountered, we delete the red segment $\overline{pq}$ from the search tree and, in the sweep tree, link $\overline{pq}$ as the child of the segment above q and make the former child the sibling of $\overline{pq}$. When a blue point r is encountered, we find the red segment, s, above the point r in the search tree and, in the sweep tree, link r as the child of s and link the former child as the sibling of r. This process takes logarithmic time for each point if the search tree is kept balanced—all other operations are constant time.

The sorted order of the segments and points can now be recovered from the sweep tree. We can number the nodes of the tree from 1 to the highest number, n, in inorder: starting from the root, we recursively number the siblings of a node, number the node, increment the counter, and then recursively number the children of the node. By lemma 3.2, this gives us a list of segments and points sorted according to the aboveness relation. The first element (segment or endpoint) in the list will be the highest element and the last element in the list will be the lowest element.

Lemma 3.2 *Ordering the segments and endpoints stored in the nodes in increasing order of node numbers gives a total order that is consistent with the aboveness relation.*

Proof: Define the *rightward path* for a segment (or point) s to be the path beginning at the left endpoint of s and then repeatedly continuing to the right endpoint of the segment that it lies on and extends vertically to the segment above the right endpoint. The rightward paths for two segments (or points) s and t cannot cross: when they meet, they must meet along a segment where they will join.

Look at the segment u where the rightward paths for s and t join. (Recall that a dummy infinite segment is stored at the root of the sweep tree.) if s is above t, then either $u = s$ or s and t are in subtrees of the sweep tree that are rooted at children of u. In the former case, s is the parent of t and an inorder traversal of the sweep tree numbers children of s after s. In the latter case, the root of the subtree containing s is a sibling to the left of the root of the subtree containing t; again, an inorder traversal numbers s before t. ∎

3.3 The Red/Blue Intersection Algorithm

This algorithm takes two topologically ordered lists of segments and points, assigns each segment and point to its slabs and computes the number of intersections in each slab. Figure 4 outlines our intersection algorithm. We pass the head pointers to the sorted lists of red segments and blue endpoints, and blue segments and red endpoints to this routine. When all of the red/blue intersections are found, we return the total number of intersections.

```
long int intersection_count(red, blue)
     color_data *red, *blue;
{                /* count intersections given sorted orders      */
  register int i,j;

  nslabs = prepare_first_slabs(red, blue);  /* initialize slabs, */
                      /* storing the slab index for each point */
  do            /* for each level in the segment tree            */
    { clear_slabs(nslabs, red); /* initialize slabs              */
      clear_slabs(nslabs, blue);
      make_longs(red);          /* put long segments in place    */
      make_longs(blue);

      total_long_long(nslabs, red, blue); /* find intersections  */
      total_long_short(nslabs, red, blue);
      total_long_short(nslabs, blue, red);

      fix_inslab(red);  /* fix indices and halve number of slabs */
      fix_inslab(blue);
      for (i = 1, j = 2; j < nslabs; i++, j +=2)
        mid[i] = mid[j];
      nslabs =  (nslabs + 1) >>1;
      mid[nslabs] = COORDMAX; /* set last midpoint on the right  */
    }
  while (nslabs > 1);   /* we are done when one slab remains     */
  return(total);        /* return number of intersections found  */
}
```

Figure 4: Computing red/blue segment intersections

We create (by the routine `prepare_first_slabs()`) a list of slab boundaries so that each slab contains one segment endpoint as described in section 2. This routine also stores with each point p the index to the slab containing p. Now, we count the number of intersections found at each level in the segment tree and return the sum of these totals.

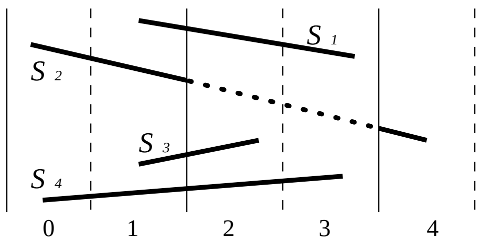

Figure 5: Inserting long segments into slabs

For each level in the segment tree, we must assign each long segment to the proper slabs. To do this the routine `make_longs()` traverses the list of segments and endpoints and inserts the long segments into the slabs in sorted order from highest to lowest. When a segment, S, is encountered, we examine the left and right slab indices, l and r, already stored with each endpoint of S. If l and r are adjacent slabs, or the same slab, then S is not stored long anywhere (see segment S_3 in figure 5). If l is an even slab, then S is stored long in slab $l+1$ (see segments S_2 and S_4 in figure 5). If r is an odd slab, then S is stored long in slab $r-1$ (see segment S_1 and S_4 in figure 5). When an endpoint, p, is encountered, the number of long segments already in the slab is recorded with p as the count of long segments (of opposite color) above p.

Next, we find the intersections between long red segments and long blue segments (by the routine `total_long_long()`). For each long red segment, we count the number of long blue segments above each of its endpoints. To do this, we *merge* the long red list with the long blue list along both boundaries of each slab by comparing y-coordinates. For each slab, starting with the left boundary of the slab, we step through each long list from top to bottom. When a blue segment is crossed, we increment a counter by one. When a red segment is reached, we store the current counter with the left red endpoint as its *above* count. (Note that we do not actually create a final merged list since we set the red endpoint counts *during* the merging process.) Similarly, we perform the merge on the right slab boundary. Then by subtracting the left and right counts for each long red segment, we obtain the number of blue segments that cross (intersect) the long red segment. For example, back in figure 1 the number of long blue segments above the left endpoint, pl, of the long red segment is 5, and the number for the right endpoint, pr, is 3. So the number of intersections along the long red segment is 2. We add the absolute value of the difference between the left and right endpoint counts to the total intersection count.

Finally, we count the intersections between long blue segments and short red segments, remembering that we have already stored the number of long blue segments above each red endpoint with each original red endpoint (in the routine `make_longs()`). We traverse the *original* list of red segments in sorted order. For each red segment we know the slabs containing each endpoint of the segment. If both endpoints are inside the same slab, we simply add the absolute difference of the counts for each endpoint to the total number of intersections. If the slabs are different, we must count the number of long blue segments above each point on the slab boundaries. Starting with the first short red segment, we clip the segment to the slab boundary. As before, we use the y-value of this intersection point to find the number of long blue segments above it on the slab boundary. The absolute difference between this count and the count stored with the starting endpoint is added to the total number of intersections. We do the same for the second short segment.

We use the same procedure to find the intersections between long blue segments and short red segments. Once this is completed for all slabs, we proceed to the next level in the segment tree by throwing out every other slab boundary and merging pairs of adjacent slabs. Then, we update the slab indices stored with each endpoint (by the routine `fix_inslab()`). When all levels of the segment tree have been processed, we return the total number of intersections found.

3.4 Special Cases

So far, we have assumed that degenerate cases do not occur; that no red or blue endpoint lies on an oppositely colored segment, that no red/blue intersection point lies on a slab boundary, and that no red and blue segment endpoints lie on the same vertical line. In practice, such situations do arise and we must ensure that they are handled properly.

When a red and blue segment intersect at the boundary dividing two slabs, s_1 and s_2 (left and right respectively), as in figure 6. We want the intersection point to be detected in only one slab, either s_1 or s_2. By our definition of slabs in section 2, only the left slab boundaries are stored with each slab, not the right boundaries. This means that the intersection should be detected in s_2, not in s_1. When we merge the y-values on slab boundaries for finding the number of long segments above a point, if we detect equal y values then we can count the intersection point only if we are on the *left* slab boundary.

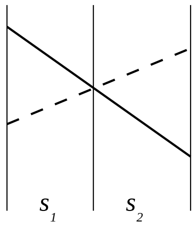

Figure 6: Intersection on boundary

We can detect the remaining degeneracies and handle most of them during the presorting phase of the algorithm.

When red and blue endpoints lie on the same vertical line, then we conceptually perturb the blue endpoints to the right of the line. If endpoints coincided or endpoints lay on vertical segments, then this perturbation can change the intersection count. The count should be repaired, depending on the policy of how to count endpoint intersections.

If a blue endpoint, b_1, lies somewhere on a red segment, $\overline{rs}$, then we must decide whether b_1 is above or below $\overline{rs}$. If the second blue endpoint, b_2, is not on the line through $\overline{rs}$, then we can make one endpoint above $\overline{rs}$ and one below $\overline{rs}$, so the intersection point will be detected during the intersection algorithm. That is, if b_2 is above $\overline{rs}$, then we choose b_1 to be below $\overline{rs}$. If b_2 is below $\overline{rs}$, then we choose b_1 to be above $\overline{rs}$.

If the point b_2 is also on $\overline{rs}$, then the two segments are colinear. In this case, we must test to see if the two segments intersect only at an endpoint, in which case we arbitrarily make one endpoint above $\overline{rs}$ and one below $\overline{rs}$. If there are an infinite number of intersection points, then we report this to the user (in addition to the final intersection count). These cases are problematic—we either have to trust the floating point computations to consistently find that these segments overlap whenever they are both stored long in a slab, or we need to mark such segments to force this consistency. Our implementation does the former (and therefore occasionally counts overlapping segments as multiple intersections); in the GIS overlay problem it is much more appropriate to do the latter. In other ways, our algorithm has been more robust than sweep algorithms such as Bentley-Ottmann [1] because all other computation can be performed on original data points instead of derived points.

4 Results of Implementation

We have implemented this algorithm in about 750 lines of C, excluding the I/O and debugging code. Total execution times and time after the initial topological sorting are reported in table 1. Synthetic data sets and GIS data from Littleton, Colorado, and the UBC research forest were used on a Sun 4/75 and a Silicon Graphics Crimson. By way of comparison, the direct implementation (checking all pairs of segments) on the Crimson takes 0.5 seconds for 400 segments of each color, 50.88 seconds for 4 000, and over 80 minutes for 40 000.

Data Set Name	Number of Segments	No. that Intersect	Sun 4/75 (secs) total	Sun 4/75 (secs) after	SGI Crimson total	SGI Crimson after
Complete	400 × 400	160 000	0.42	0.27	0.14	0.09
Grid	4 000 × 4 000	16 000 000	5.93	4.00	1.77	1.12
	40 000 × 40 000	1.6×10^9	79.40	52.07	29.81	19.95
	200 000 × 200 000	4×10^{10}	—	—	181.88	123.79
Horizontal	4 000 × 4 000	15297	7.19	5.30	1.89	1.34
& Slanted	4 000 × 4 000	290 876	10.70	7.18	2.88	1.79
	40 000 × 40 000	1 523 785	118.67	79.93	35.55	24.32
	40 000 × 40 000	29 249 076	181.72	99.52	61.23	36.94
roads/survey	11 074 × 239	536	8.87	5.33	2.70	1.51
roads/vegitation	11 074 × 5 562	202	14.12	8.98	4.130	2.44
forest/compart	116 359 × 8 053	4 637	128.82	80.97	40.22	24.39
biogeo/compart	235 635 × 8 053	3 548	—	—	81.93	49.73
biogeo/forest	235 635 × 116 359	50 045	—	—	136.66	92.53

Table 1: Total and "after-sorting" execution times on a 16 Meg Sun 4/75 (spark 2) and a 64 Meg SGI Crimson

5 Conclusions

The main advantage of this algorithm is that a segment tree data structure is not required. We merely store one level of the segment tree as a list of slabs. This means that fewer pointers are needed, less memory is required and the algorithm is easier to implement.

Another advantage is that the sorting of the segments and endpoints is done first, independently from the intersection calculation phase. In GIS overlay applications, this means that data can be pre-sorted just once prior to storage. Future accesses to this data need not sort again. This would save considerable time with little or no additional memory costs.

Acknowledgements

We thank Otfried Schwarzkopf for discussions on red/blue intersection problems, Jerry Maedel for data from the UBC research forest, and Scott Andrews for converting the program to read GIS data. We also thank the referees for their comments.

References

[1] J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers*, C–28(9):643–647, 1979.

[2] B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. *Journal of the Association for Computing Machinery*, 39:1–54, 1992.

[3] B. Chazelle, H. Edelsbrunner, L. Guibas, and M. Sharir. Algorithms for bichromatic line segment problems and polyhedral terrains. Technical Report UIUC DCS–R–90–1578, Dept. Comp. Sci., Univ. Ill. Urbana, 1990.

[4] B. Chazelle and L. J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1:133–162, 1986.

[5] B. Chazelle and L. J. Guibas. Fractional cascading: II. Applications. *Algorithmica*, 1:163–191, 1986.

[6] O. Devillers and A. Fabri. Scalable algorithms for bichromatic line segment intersection problems on coarse grained multicomputers. In *Algorithms and Data Structures (WADS '93)*, number 709 in Lecture Notes in Computer Science, pages 277–288. Springer-Verlag, 1993.

[7] H. Edelsbrunner and M. H. Overmars. Batched dynamic solutions to decomposable searching problems. *Journal of Algorithms*, 6:515–542, 1985.

[8] ESRI White Paper Series. Environmental Systems Research Institute, Inc. *ARC/INFO: GIS Today and Tomorrow*, Mar. 1992.

[9] Federal Geomatics Bulletin, 4(1), 1992. GIS Division, Energy, Mines and Resources. Ottawa, Canada.

[10] D. E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, second edition, 1973.

[11] H. G. Mairson and J. Stolfi. Reporting line segment intersections. In R. Earnshaw, editor, *Theoretical Foundations of Computer Graphics and CAD*, number F40 in NATO ASI Series, pages 307–326. Springer-Verlag, 1988.

[12] F. P. Preparata and M. I. Shamos. *Computational Geometry—An Introduction*. Springer-Verlag, New York, 1985.

[13] M. I. Shamos and D. Hoey. Geometric intersection problems. In *Proceedings of the 17th IEEE Symposium on Foundations of Computer Science*, pages 208–215, 1976.

Appendix A: C code for Red/Blue Segment Intersection

This section contains the code for the main data structures and procedures for our red/blue segment intersection algorithm. (Procedures for display, debugging, and threaded binary search tree manipulation have been omitted to save space.) We list the definitions of data structures and the main procedure first. Then we list code for phase 1 and phase 2 computations.

```
/* Red/Blue Intersection Counting  5 July 93
 *
 * From L. Palazzi and J. Snoeyink, "Counting and Reporting Red/Blue Segment Intersections",
 * WADS'93, Springer Verlag, LNCS , 1993.  with thanks to Scott Andrews for debugging and
 * modifications to read GIS data.
 * WARNING: this code was written for tight memory.  There are some dangerous hacks (such as using
 * the LSB of a pointer field as a tag bit, two names for one variable, unions...) that you should
 * watch out for if you modify this code.
 * It was also written for speed, so it sacrifices modularity and structure.
 * Degeneracies are handled by perturbing blue segments down and to the right.  As a result,
 * invocations with files in a different order may give different answers for degenerate inputs.
 */
#define MAXSLABS   100001
#define MAXSEGS    200002
#define MAXSEGSx2  400004
#define MAXSEGSx3  600006

#define MASK_LSB 0xfffffffe

typedef float COORD;
const float COORDMAX = 1e30;                 /* "Infinity"—bigger than any coordinate */

/* A segment is a pair of POINTs and each POINT comes with three pointers/integers: tree pointers
 * during the topological sort (phase 1) and slab information during the intersection counting
 * (phase 2).  We use unions to save storage.
 */

typedef struct treenode {                    /* Phase 1: TREENODE structure for POINT */
  struct POINT *child,                       /* Points to child subtree */
  *left,                                     /* Left & right children in "balanced" tree of segs
  *right;                                       intersecting sweep (threaded) */
} treenode;

#define sibling left                         /* ** WARNING: Dangerous hack *** "left" does
                                                double duty—it becomes "sibling" once a node leaves
                                                the sweep */
typedef struct slabentry {                   /* Phase 2: SLABENTRY structure for POINT */
  int inslab;                                /* Slab containing this point */
  int above;                                 /* Count of long segments above endpoint */
  struct POINT *lnext;                       /* Next long seg in slab's linked list */
} slabentry;

typedef struct POINT {                       /* POINT structure */
  COORD x, y;                                /* Coordinates */
  union {
    treenode t;                              /* P1: balanced tree & top sort tree node */
    slabentry e;                             /* P2: entry in slab list */
  } u;
} POINT;
```

```
/* Each color (red/blue) has an associated list of segments (point pairs), a topological order of
 * segments & opposite colored points, & list of slabs (phase 2) or points in x-order (phase 1).
 */
typedef struct {                              /* Phase 2: SLAB structure                        */
  int abovel, abover;                         /* # of longs above on left and right             */
  POINT *longl, *longr;                       /* Heads for left & right long lists              */
} SLAB;

typedef struct {                              /* color_ data: All info for a color              */
  int n;                                      /* Number of points == twice # of segments        */
  POINT p[MAXSEGSx2];                         /* Segments stored as POINT pairs                 */
  POINT *toporder[MAXSEGSx3];                 /* P1- >P2: Top order of segs & other points      */
  union {
    POINT *pptr[MAXSEGSx2];                   /* P1: Pointers to endpoints sorted by x          */
    SLAB slab[MAXSLABS];                      /* P2: Slab structures for long segments          */
  } u;
} color_data;

color_data red, blue;                         /* Data structures for red and blue segments. WARN-
                                              ING: Address tests assume that red is before blue. */
COORD mid[MAXSLABS];                          /* Current slab boundaries                        */
int nslabs;                                   /* Number of slabs in current use                 */

/* Compare points: TRUE if p > q.  Breaking ties with pointer addresses makes blue points greater
 * than red points when the x coordinates are equal.  */
#define PointGT(p, q) (((p)->x > (q)->x) || (((p)->x == (q)->x) && ((p) > (q))))
#define SegmentP(x, color)                    /* True if point is a segment of given color      */
   (((unsigned) (x - (color)->p)) < MAXSEGSx2)
#define EndpointP(pt, color)                  /* True if point is endpoint of "color" seg       */
   ((pt - (color)->p) & 1)

/* definitions for phase 1 tree handling */
#define DET2(a, b, c, d)                      /* 2x2 determinant                                */
   ((a)*(d) - (c)*(b))
#define DETPts(p, q)                          /* 2x2 determinant for points                     */
   DET2((p)->x, (p)->y, (q)->x, (q)->y)
#define CCW(p, q, r)                          /* Counterclockwise test                          */
   DET2((q)->x - (p)->x, (q)->y - (p)->y, (r)->x - (p)->x, (r)->y - (p)->y)
#define Swap(p, q, tmp)                       /* Swap function using temporary tmp              */
   { tmp = *(p); *(p) = *(q); *(q) = tmp; }
#define ZeroP(x) (x == 0)                     /* Zero test (maybe use epsilon later)            */

/* The tree of segments intersecting the sweep line is threaded for easy deletion and pred
 * computation.  Thus, we have to tag predecessor and successor pointers (that would otherwise be
 * nil).  We do so by setting the last bit, assuming that the valid addresses are even.  This works
 * on sgi, because the fields are multiples of 4 bytes long and are aligned to 4 byte boundaries,
 * but may fail on other c compilers/machines.  */
#define Tag(x)                                /* Tag a pointer                                  */
   ((POINT *) (((unsigned int) (x)) | 1))
#define NullP(x)                              /* True if the pointer is tagged                  */
   (((unsigned int) (x)) & 1)
#define cleanPtr(x)                           /* Remove any tag to get a valid pointer          */
   ((POINT *) (((unsigned int) (x)) & MASK_LSB))
```

```
/* definitions for phase 2 intersection counting */
#define AbsDiff(x,y)                                   /* Absolute value of difference |x − y|    */
   ((x) < (y) ? (y) - (x) : (x) - (y))
#define GetStartEnd(seg, segs, st, en)                 /* Get start and end pts of seg            */
   if (EndpointP(seg, segs)) st = (en = seg) - 1;   else en = (st = seg) + 1;
#define SegIntNumer(st, en, wallx)                     /* Numerator of seg/wall intersection      */
   DET2(wallx - st->x, st->y, wallx - en->x, en->y)
#define SegIntDenom(st, en)                            /* Denominator of seg/wall intersection    */
   (en->x - st->x)

main(argc,argv)                                        /* Main routine                            */
     int argc;
     char **argv;
{
  input_segs(&red, infileRed);                         /* Get segment information                 */
  input_segs(&blue, infileBlue);

  before_sweep(&red);                                  /* Here begins the real work               */
  before_sweep(&blue);
  topological_sort(&red, &blue);
  topological_sort(&blue, &red);
  printf(" %ld ", intersection_count(&red, &blue));
}
```

The code for phase 1 must sort segments according to the topological order.

```
int PointCompare(p, q)                                 /* For qsort                               */
     POINT *p, *q;
{   return (PointGT(p, q) ? 1 : -1); }
int PointPtrCompare(p, q)                              /* For qsort                               */
     POINT **p, **q;
{  return (PointGT(*p, *q) ? 1 : -1); }

void before_sweep(color)
     color_data *color;
{
  register int i;
  register POINT *ptr;
  POINT tmp;

  for (ptr = color->p; ptr < color->p + color->n; ptr += 2)
    if (PointGT(ptr, ptr+1)) Swap(ptr, ptr+1, tmp);   /* Make sure startpt preceeds endpt        */

  qsort(color->p, color->n >> 1,
2*sizeof(POINT), PointCompare);                        /* Sort segments by start x coord          */

  color->p[color->n].x = COORDMAX;                     /* Last segment is backwards at y=infty    */
  color->p[color->n].y = COORDMAX;
  color->p[color->n+1].x = 0;
  color->p[color->n+1].y = COORDMAX;

  for (i=0; i < color->n + 2; i++) color->u.pptr[i] = color->p + i;

  qsort(color->u.pptr, color->n,
sizeof(POINT *), PointPtrCompare);                     /* Sort pointers to points by x            */
}
```

```
POINT *Init_tree(color)
     color_data *color;
{                                                   /* Initialize root node of tree of "color"   */
  POINT *root;

  root = color->p + color->n;                       /* Use sentinel as root node                 */
  root->u.t.child = NULL;
  root->u.t.left = root->u.t.right = Tag(root);
  return(root);
}

void insert_left(parent, node)                      /* Insert node as left child of parent Code omitted
     POINT *parent, *node;                          to save space                                */

void insert_right(parent, node)                     /* Insert node as right child of parent Code omitted
     POINT *parent, *node;                          to save space                                */

POINT *delete(node)                                 /* Delete node from the threaded tree and return
     POINT *node;                                   predecessor. Code omitted to save space      */

/* topological_sort(segs, pnts)
 * Takes color data for a set of segments and points that have previously been sorted and computes a
 * total order consistent with aboveness. Uses a left/right sweep to compute a tree whose inorder
 * traversal is consistent with aboveness.  Assumes segments and points have been sorted and the
 * last segment has x coord of COORDMAX.
 */
void topological_sort(segs, pnts)
     color_data *segs, *pnts;
{
  POINT **sgptr, **ptptr;
  register POINT *sg, *pt;
  register POINT *root, *curr, *pr;
  register float test;

  root = Init_tree(segs);                           /* Init root: segment at infinity            */
  sg = *(sgptr = segs->u.pptr);
  pt = *(ptptr = pnts->u.pptr);

  while ((pt->x < COORDMAX) || (sg->x < COORDMAX))
    {
      if (PointGT(sg, pt))                          /* x-order, with red before blue on ties     */
        {                                           /* Sweep a point                             */
          curr = root->u.t.left;
          pr = root;                                /* Search for segment above curr point       */
          while (!NullP(curr))
            { test = CCW(curr, curr+1, pt);
              if (ZeroP(test))                      /* Point on seg—resolve using other endpoint */
                { if (EndpointP(pt, pnts)) test = -CCW(curr, curr+1, pt-1);
                  else test = -CCW(curr, curr+1, pt+1);
                  if (ZeroP(test))                  /* Segments are colinear–move blue below     */
                    test = ((pt > sg) ? -1.0 : 1.0);
                }
              if (test > 0) curr = curr->u.t.left;  /* If point above curr seg, else             */
              else { pr = curr; curr = curr->u.t.right; }
            }
```

```
          pt->u.t.sibling = pr->u.t.child;          /* Insert in forest                                  */
          pt->u.t.child = NULL;
          pr->u.t.child = pt;
          pt = *(++ptptr);
        }
      else * We have the start or endpoint of a seg. */
        { if (EndpointP(sg, segs))                  /* Sweep a segment endpoint                          */
            { curr = sg - 1;
              pr = delete(curr);                    /* Delete from sweep                                 */
              curr->u.t.sibling = pr->u.t.child;    /* Insert in forest                                  */
              pr->u.t.child = curr;
              sg = *(++sgptr);
            }
          else                                      /* Sweep a segment start point                       */
            { if (NullP(root->u.t.left)) insert_left(root, sg);
              else
                { curr = root->u.t.left;
                  while (1)                         /* Insert into sweepline                             */
                    {
                      test = CCW(curr, curr+1, sg);
                      if (ZeroP(test))              /* On the segment-make consistent                    */
                        test = CCW(curr, curr+1, (sg)+1);

                      if (test > 0)                 /* New seg above curr seg                            */
                        if (!NullP(curr->u.t.left)) curr = curr->u.t.left;
                        else                        /* Space available here                              */
                          { insert_left(curr, sg); break; }
                      else                          /* Point below curr seg                              */
                        if (!NullP(curr->u.t.right)) curr = curr->u.t.right;
                        else { insert_right(curr, sg); break; }
                    }
                }
              sg = *(++sgptr);
            }
        }
    }

  sgptr = segs->toporder;                           /* Stackless tree traversal to extract the topological
  curr = root;                                      order from the tree.                                 */
  curr->u.t.right = root;
  do
    {                                               /* Invariant: sibling of curr is NULL here.          */
      if ((pt = curr->u.t.child) == NULL)
        curr = curr->u.t.right;                     /* Leaf node: go to successor and output             */
      else
        {                                           /* Not a leaf: traverse u.t.child subtree            */
          pt->u.t.right = curr->u.t.right;          /* Save successor                                    */
          curr = pt;
          while ((pt = curr->u.t.sibling) != NULL)
            { pt->u.t.right = curr; curr = pt; }    /* Make parent (=successor) pointer                  */
        }
      *(sgptr++) = curr;
    }
  while (curr != root);
}
```

The code for phase 2 uses the ordering of the segments to count the intersections.

```
long int total = 0;                                /* GLOBAL: accumulate total intersections */

int prepare_first_slabs(red, blue)
     color_data *red, *blue;                       /* Find slab boundaries and give each point the index
{                                                  of slab containing it */
  register POINT *rp, *bp, **rptr, **bptr;
  register int i, blueslab;
  register COORD last;

  rp = *(rptr = red->u.pptr);                      /* Points in x-coord sorted order */
  bp = *(bptr = blue->u.pptr);
  mid[i = 0] = last = -COORDMAX;
  blueslab = PointGT(rp, bp);                      /* Is first slab red or blue? */
  do
    { if (blueslab != PointGT(rp, bp))             /* Is next point different color? */
        if (blueslab = !blueslab)                  /* Switch slab color */
          mid[++i] = (last + bp->x) / 2.0;         /* Blue point next */
        else mid[++i] = (last + rp->x) / 2.0;      /* Red point next */

      if (blueslab)                                /* Invariant: blueslab == PointGT(rp, bp) */
        { last = bp->x;                            /* Get next blue */
          bp->u.e.inslab = i;
          bp = *(++bptr);
        }
      else
        { last = rp->x;                            /* Get next red */
          rp->u.e.inslab = i;
          rp = *(++rptr);
        }
    }
  while ((rp->x < COORDMAX) || (bp->x < COORDMAX));
  return(i+1);                                     /* Return number of slabs used */
}

void clear_slabs(nslabs, longs)
     int nslabs;
     color_data *longs;
{                                                  /* Clear slabs initially */
  register SLAB *sptr;
  for (sptr = longs->u.slab; sptr < longs->u.slab + nslabs; sptr++)
    { sptr->abovel = sptr->abover = 0; sptr->longl = sptr->longr = NULL; }
}

void fix_slabs(nslabs, longs)
     int nslabs;
     color_data *longs;
{                                                  /* Reinit once long lists are computed */
  register SLAB *sptr;
  for (sptr = longs->u.slab; sptr < longs->u.slab + nslabs; sptr++)
    { sptr->abover = sptr->abovel = 0; sptr->longr = sptr->longl; }
}
```

```
void make_longs(segs)
     color_data *segs;                        /* Inserts points and segs into slabs according to topo-
{                                             logical order                                         */
  register int start_slab, end_slab;
  register SLAB *slab;
  register POINT *point, *epoint;
  POINT **order;

  for (order = segs->toporder;  ((point = *order)->x < COORDMAX); order++)
    if (!SegmentP(point, segs))               /*  Is this a point?                                  */
      point->u.e.above = segs->u.slab[point->u.e.inslab].abovel;
    else
      {                                       /*  Item is a segment—place it long                   */
        start_slab = point->u.e.inslab;
        end_slab = (epoint = point+1)->u.e.inslab;

        if (start_slab + 1 < end_slab)
          {                                   /*  If slabs are not the same or adjacent, add long
            if (!(start_slab & 1))            segment in slab adjacent to start                     */
              {
                slab = segs->u.slab + (start_slab | 1);
                if (slab->longl == NULL)      /*  Is head of list NULL?                             */
                  slab->longl = slab->longr = point; /*  Insert at head & tail                      */
                else
                  {                           /*  Insert tail & advance                             */
                    slab->longr->u.e.lnext = point;
                    slab->longr = point;
                  }
                point->u.e.lnext = NULL;
                slab->abovel++;               /*  Count as we insert                                */
              }

            if (end_slab & 1)                 /*  Add long seg in slab adj to end                   */
              {
                slab = segs->u.slab + (end_slab & MASK_LSB);
                if (slab->longl == NULL)      /*  Is head of list NULL?                             */
                  slab->longl = slab->longr = epoint; /*  Insert at head & tail                     */
                else
                  {                           /*  Insert tail & advance                             */
                    slab->longr->u.e.lnext = epoint;
                    slab->longr = epoint;
                  }
                epoint->u.e.lnext = NULL;
                slab->abovel++;               /*  Count as we insert                                */
              }
          }
      }
}
```

```
void total_long_long(nslabs, red, blue)
     int nslabs;
     color_data *red, *blue;
{                                                   /*  For each slab, compute long/long int.          */
  register int i;                                   /*  Slab index                                     */
  register POINT *redp,                             /*  Current red seg pointer and                    */
  *blp, *brp;                                       /*  Blue segs intersecting lf & rt above red       */
  int left, right;                                  /*  # of blues seen on left & right                */
                                                    /*  Intersections with the lf & rt walls           */
  register POINT *st, *en;                          /*  Start and endpoint of segment                  */
  register double rlnum, rrnum, blnum, brnum;       /*  Numerators                                     */
  register double rdenom, bldenom, brdenom;         /*  Denominators                                   */

  for (i = 0; i < nslabs; i++)
    {
      redp = red->u.slab[i].longl;
      blp = brp = blue->u.slab[i].longl;
      if ((redp != NULL) && (blp != NULL))          /*  If both lists non-empty                        */
        { left = right = 0;
          GetStartEnd(blp, blue, st, en);           /*  Find blue wall intersections                   */
          blnum = SegIntNumer(st, en, mid[i]);
          brnum = SegIntNumer(st, en, mid[i+1]);
          brdenom = bldenom = SegIntDenom(st, en);
          do
            { GetStartEnd(redp, red, st, en);       /*  Red wall intersections                         */
              rlnum = SegIntNumer(st, en, mid[i]);
              rrnum = SegIntNumer(st, en, mid[i+1]);
              rdenom = SegIntDenom(st, en);

              while ((blp != NULL) && (rlnum*bldenom < blnum*rdenom))
                { left++;                           /*  While red below blue on left get next blue     */
                  if ((blp = blp->u.e.lnext) == NULL)
                    break;
                  GetStartEnd(blp, blue, st, en);
                  blnum = SegIntNumer(st, en, mid[i]);
                  bldenom = SegIntDenom(st, en);
                }
              while ((brp != NULL) && (rrnum*brdenom < brnum*rdenom))
                { right++;                          /*  While red below blue on right get next blue    */
                  if ((brp = brp->u.e.lnext) == NULL)
                    break;
                  GetStartEnd(brp, blue, st, en);
                  brnum = SegIntNumer(st, en, mid[i+1]);
                  brdenom = SegIntDenom(st, en);
                }
              total += AbsDiff(left, right);        /*  Accumulate total                               */
              redp = redp->u.e.lnext;               /*  Get next red                                   */
            }
          while (redp != NULL);
        }
    }
}
```

```
void total_long_short(nslabs, longs, shorts)
     int nslabs;
     color_data *longs, *shorts;              /* Intersects shorts with longs by inserting shorts in
{                                                topological order                                    */
  register POINT *point, *st, *en;
  register SLAB *start_slab, *end_slab;
  register COORD middle;
  register double wallnum, walldenom, test;
  POINT **order;

  fix_slabs(nslabs, longs);                   /* Reinitialize slab wall counts                       */
  for (order = shorts->toporder; ((point = *order)->x < COORDMAX); order++)
    if (SegmentP(point, shorts))              /* If segment, check if ends in same slab              */
      if ((start_slab = longs->u.slab + point->u.e.inslab)
          == (end_slab = longs->u.slab + (point+1)->u.e.inslab))
        total += AbsDiff(point->u.e.above, (point+1)->u.e.above);
      else
        {                                     /* Start and end slabs are different                   */
          if (start_slab->longr != NULL)
            { GetStartEnd(point, shorts, st, en);  /* Handle slab containing start point             */
              middle = mid[point->u.e.inslab+1];
              wallnum = SegIntNumer(st, en, middle);
              walldenom = SegIntDenom(st, en);
              do
                {                             /* Loop past longs above current short                 */
                  GetStartEnd(start_slab->longr, longs, st, en);
                  test = (wallnum * SegIntDenom(st, en) - walldenom * SegIntNumer(st, en, middle));
                  if ((test > 0)              /* Next long is below me?                              */
                      || (ZeroP(test) && (point>st))) /* Or tie? Move blue down                      */
                    break;
                  start_slab->abover++;
                }
              while ((start_slab->longr = start_slab->longr->u.e.lnext) != NULL);
            }
          total += AbsDiff(start_slab->abover, point->u.e.above);

          if (end_slab->longl != NULL)
            { GetStartEnd(point, shorts, st, en);  /* Handle slab containing end point               */
              middle = mid[(point+1)->u.e.inslab];
              wallnum = SegIntNumer(st, en, middle);
              walldenom = SegIntDenom(st, en);
              do
                {                             /* Loop past longs above current short                 */
                  GetStartEnd(end_slab->longl, longs, st, en);
                  test = (wallnum * SegIntDenom(st, en) - walldenom * SegIntNumer(st, en, middle));
                  if ((test > 0)              /* Next long is below me?                              */
                      || (ZeroP(test) && (point>st))) /* Or tie? Move blue down                      */
                    break;
                  end_slab->abovel++;
                }
              while ((end_slab->longl = end_slab->longl->u.e.lnext) != NULL);
            }
          total += AbsDiff(end_slab->abovel, (point+1)->u.e.above);
        }
}
```

```
void fix_inslab(color)
     color_data *color;
{                                                 /*  Shift slab indices for next go-round          */
  register POINT *ptr;
  for (ptr = color->p; ptr < color->p + color->n; ptr++) ptr->u.e.inslab >>= 1;
}

long int intersection_count(red, blue)
     color_data *red, *blue;
{                                                 /*  Count intersections (given top. orders)       */
  register int i, j;

    nslabs = prepare_first_slabs(red, blue);
  do
    {
      mid[nslabs] = COORDMAX;
      clear_slabs(nslabs, red);
      clear_slabs(nslabs, blue);
      make_longs(red);                            /*  Put long segs in place                        */
      make_longs(blue);

      total_long_long(nslabs, red, blue);
      total_long_short(nslabs, red, blue);
      total_long_short(nslabs, blue, red);

      fix_inslab(red);                            /*  Halve the number of slabs and fix indices     */
      fix_inslab(blue);
      for (i = 1, j = 2; j < nslabs; i++, j +=2)
        mid[i] = mid[j];
      nslabs =  (nslabs + 1) >> 1;
      mid[nslabs] = COORDMAX;
    }
  while (nslabs > 1);                             /*  We are done when one slab remains             */
  return(total);
}
```