Fast Software for Box Intersections *

Afra Zomorodian † and Herbert Edelsbrunner ‡

Abstract

We present fast implementations of a hybrid algorithm for reporting box and cube intersections. Our algorithm initially takes a divide-and-conquer approach and switches to simpler algorithms for low numbers of boxes. We use our implementations as engines to solve problems about geometric primitives. We look at two such problems in the category of quality analysis of surface triangulations.

Keywords. box intersection, algorithms, segment tree, range tree, implementation, quantification, experimentation.

1 Introduction

In this paper, we present fast implementations of a hybrid algorithm for reporting three-dimensional box and cube intersections. Compared to other methods, our algorithm is simple, small, and does not keep any data structure in memory other than the array of boxes and pointers to them. Figure 1 shows the performance of one of our implementations to inspire interest. Unless otherwise stated, all our timings are done on a Sparc Ultra-30. The running time is averaged over 10 runs for each n, the n boxes being chosen from a distribution that generates about $n/2$ intersecting pairs.

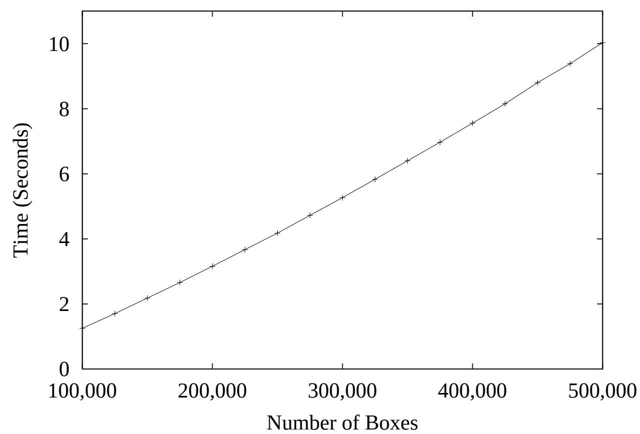


Figure 1: Performance of Hybrid Algorithm (Version One.)

Motivation. Computing information about complex geometric primitives is often expensive. While researchers in computational geometry have discovered many asymptotically efficient algorithms for such problems, the algorithms are often of theoretical interest only, as they are hard to implement or slow if implemented. One general approach to solving such tasks is to reduce the problem size enough so that a brute force method is feasible and fast. By putting axis-aligned bounding boxes around complicated primitives, we can compute, in a short amount of time, an approximate answer in the form of a small set of pairs with potential interaction. We then refine our answer by looking inside the boxes, computing the exact answer for the original data.

Many geometric problems can be restated in terms of box intersections since if the bounding boxes of two primitives intersect, we know that the primitives intersect or have close proximity. Some practical problems we may solve with this approach include detecting surface self-intersection, performing proximity queries, and measuring the effect of surface modification algorithms. We are therefore motivated to implement fast and useful tools for reporting box intersections. We will use these tools as engines for solving problems about geometric primitives. Our algorithm is industrial strength and is already part of *geomagic® Studio*, a surface modeling software product from Raindrop Geomagic.

*Research by both authors is partially supported by the Center for Simulation of Advanced Rockets funded by the U.S. Department of Energy under Subcontract B341494. Research by the second author is also partially supported by NSF under grant CCR-96-19542 and ARO under grant DAAG55-98-1-0177.

†Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois 61801.

‡Department of Computer Science, Duke University, Durham, North Carolina 27708, and Raindrop Geomagic, Research Triangle Park, North Carolina 27709.

Prior work. The idea of using simple enclosing volumes as approximations of geometric primitives is not new. Axis-aligned bounding boxes, the most popular type, are widely used in many fields including computer graphics [7], physical systems simulation (especially in collision detection [14]), and robotics [13]. Other enclosing volumes used and considered include bounding cubes and spheres, spherical shells [12], bounding boxes that are not axis-aligned (OBBs) [8], and "optimal" fitting volumes. Often, volumes are used along with a hierarchical data structures such as binary space partition (BSP) trees [16], Octrees [17], kd-trees [1], and R-trees [9].

Enclosing volume algorithms are efficient only when the volumes do not intersect significantly more times than the underlying primitives. While there are pathological cases which would cause the algorithms to perform poorly, in practice the algorithms are highly successful. Zhou and Suri derive tight worst-case bounds on the performance of axis-aligned bounding box algorithms, requiring only that the enclosed primitives be well-shaped on average [21]. The domains which interest us generally satisfy the specified requirements.

Our work. Like Knuth [11], we believe that "the best theory is inspired by practice and the best practice is inspired by theory." We use theoretically established data structures [2, 3], algorithms, and techniques [6] as ingredients for our work and guide our algorithm design by quantitative analyses of implementations on multiple platforms. Testing every design decision by actual implementations, we have refined our algorithm and uncovered surprising facts at odds with our theoretical assumptions. The result is a fast, simple, and natural algorithm.

Overview. The remainder of this paper is organized as follows. In Section 2, we define the problem and discuss simple solutions. In Section 3, we review algorithms and hierarchical data structures used for box intersection and develop building blocks. In Section 4, we forge a hybrid algorithm from our building blocks. We then experiment with various distributions in Section 5. In Section 6, we present details on our first implementation of the algorithm and discuss how our empirical studies on sorting led to faster programs. In Section 7 we discuss other implementations of our algorithm, including one for intersecting cubes. In Section 8, we show some applications of our implementations, concluding the paper in Section 9.

2 Problem and Simple Solutions

In this section, we will define our problem formally and review some classic techniques and algorithms which form the building blocks for our hybrid algorithm. Throughout this section, we consider algorithms for *reporting* box intersections for two cases of inputs:

Complete case: pairwise intersections between boxes in a single set,

Bipartite case: pairwise intersections between boxes belonging to two different sets.

We will discuss algorithms for the bipartite case and then refine them for the complete case. We will use the following notation for analysis:

Input: A set of n (or two sets of n and m) boxes,

Output: A set of k pairs of intersecting boxes.

Finally, we define a *box* to be the Cartesian product of half-open intervals with integer endpoints, closed at their low endpoints. We number the dimensions from zero. We obviate numerical inaccuracies associated with floating point computation by using integer endpoints. The input to most applications of our work, such as those shown in Section 8, consist of floating point data. In such cases, we map the floating point numbers to their ranks in the respective sorted orders.

The naive approach to the problem is the *all-pairs* algorithm. For sets of boxes A and B, we simply test each box in A against every box in B for intersection. The algorithm has $O(nm)$ complexity and is optimal if $k = \Omega(nm)$. More sophisticated algorithms are justified only when $k = o(nm)$. In most domains of interest, $k = O(n + m)$. We include all-pairs for completeness as well as noting that it makes an excellent testing tool while developing more sophisticated algorithms.

Recasting the problem. We use the following two structural properties of axis-aligned boxes to find intersecting box pairs efficiently:

Property 1. Boxes intersect if and only if they intersect in every dimension independently. As such, we may focus on algorithms for reporting intersections of one-dimensional boxes or *intervals*.

Property 2. Two intervals intersect if and only if one contains the low endpoint of the other, as shown in Figure 2. We get two disjoint cases by assuming general position, that is, the interval endpoints are pairwise different. We account for all special cases in our implementations.

Because of Property 2, we may solve the interval intersection problem as two symmetric *batched stabbing problems*: given points and intervals, we report for each point all the intervals that contain it. In our case, the points will be the low endpoints of each set of intervals

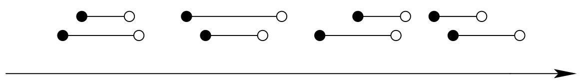


Figure 2: The four non-degenerate ways two half-open intervals can have non-empty intersection.

in turn. We will call this *viewing* a set of intervals as points.

We now focus on algorithms for batched stabbing. Our discussion will be implementation-oriented and in each case we will construct procedures for use in our hybrid algorithm.

Scanning. One technique for solving the stabbing problem is *scanning*. Given a set of n points P and a set of m intervals I, we first sort P and sort I by their low endpoints. Then, we activate the first interval $i \in I$ and make a pass through the array of points until we encounter a point $p \in P$ which is not smaller than i's low endpoint. From this point onward until we find a point which is no smaller than i's high endpoint, all the points are contained in i and we report the interval for each point. We then activate the next interval and make a new pass through the array of points starting at p. The algorithm has running time $O(n \log n + m \log m + k)$. If P and I are already sorted, the running time is $O(n + m + k)$. We will refer to this algorithm as *one-way scan*.

For the complete case, we simply use scanning with $I = P$. As each interval is represented as both an interval and a point, Property 2 implies that one-way scan will report all the intersections. We need to modify the algorithm slightly, however, not to report self-intersections. For the bipartite case, we find the intersections by scanning twice, viewing one set as points and the other set as intervals and vice versa. An alternate method is to scan both ways at the same time by having the sets compete to be viewed as intervals. We activate the interval in either set which has the lowest endpoint in each turn. We then view the set which does not contain the active interval as points and scan as before. We will refer to this algorithm as *two-way scan*.

We may also use scanning for boxes. Scanning in one dimension, we check whether the output pairs intersect in the remaining dimensions. The algorithm has running time $O(n \log n + m \log m + k')$, where k' is the number of intersections in the scanned dimension. As k' could be much larger than k in general, the algorithm is not optimal. However, for small number of boxes, the algorithm performs well.

We define the two procedures $\textsc{OneWayScan}(I, P, d)$ and $\textsc{TwoWayScan}(I, P, d)$ for later use. Both take two sets of boxes I and P as parameters and scan the sets in dimension zero. $\textsc{OneWayScan}$ treats I as intervals and P as points. The third parameter, d, specifies the highest dimension that needs to be checked. Without additional knowledge, we use $d = 2$ which means we check for every pair of boxes that intersect in dimension zero whether they also intersect in dimensions one and two.

3 Hierarchical Solutions

We now turn our attention to two hierarchical data structures which will be useful for the genesis of our algorithm: the *segment* [3] and *range* [2] trees. The segment tree solves the batched stabbing problem by storing intervals and allowing for point queries. The range tree solves the *batched range searching* problem by storing points and allowing for interval queries. The trees are very similar and a segment tree is sometimes called the range tree (e.g. [15]) and vice-versa, causing some confusion.

Segment and range trees. We demonstrate the structure and use of a segment tree with the example in Figure 3. The endpoints of n intervals are used to partition the number line into half-open intervals called *atomic segments*. In our example, the number line is divided into seven atomic segments by the three intervals. The

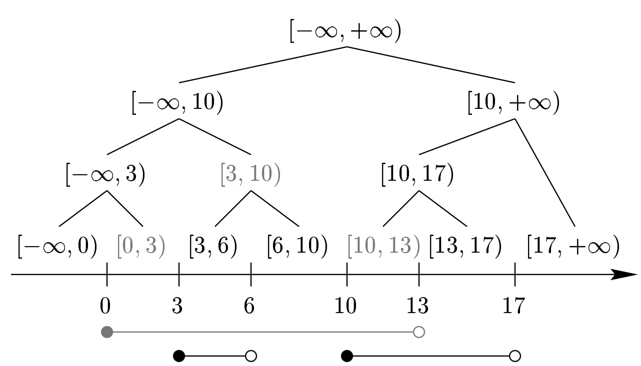


Figure 3: A segment tree for three intervals. The gray segments are the gray interval's canonical segments where the interval is stored.

segment tree is a balanced binary search tree with segments as nodes. The leaves are atomic segments and each internal segment is the union of its offspring segments. Consequently, the root segment spans the entire number line. Each node also stores the intervals that span its segment but not that of its parent. In this fashion, each interval is stored in at most $2 \log_2 n$ nodes which partition the interval into its *canonical segments*. As such, the segment tree needs $O(n \log n)$ space and may be constructed in $O(n \log n)$ time.

To report which intervals contain a particular point, we simply perform a binary search using the tree to find the atomic segment which contains the point. All the segments we encounter along the way contain the point. Therefore, we report the intervals in our path as those intervals span the segments and consequently contain the point. Observe that no interval is reported twice and that the query takes $O(\log n + k)$ time.

Much like the segment tree, the *range tree* is a binary search tree over the number line. However, the range tree uses its input set of n points to divide the number line. Each node is a half-open segment as before and stores the input points it contains. The root segment covers the entire number line so the root contains all input points. We query the range tree with an interval by two binary searches for its endpoints. During these searches, we partition the query interval into at most $2\log_2 n$ canonical segments and obtain the points the query contains by enumerating the input points stored with these segments. Again, observe that no point is reported twice and the query takes $O(\log n + k)$ time.

Intersecting boxes. We may use segment and range trees for reporting interval intersections. Given a set of n intervals A and m intervals B, we may find all intersections as follows:

(i) Build a segment tree for A viewed as intervals. Query it with B viewed as points.

(ii) Build a range tree for A viewed as points. Query it with B viewed as intervals.

Property 2 guarantees that all intersections are reported exactly once. Note that building a range tree for A and querying it with B is equivalent to building a segment tree for B and querying it with A. We will use this *duality* to simplify our algorithm later in this section.

The main insight that allows us to extend the result on intervals to boxes is that segment and range trees prepare the answer to every query in a small number of sets. Given a set of n boxes A and m boxes B, we build the two trees for the projections of A in the first dimension. We query the trees with the projection of a query box in the first dimension and get $O(\log n)$ sets of intervals or points depending on the tree. Recall that these are simply the sets stored at the nodes we visit in our search paths. When intersecting intervals, the union of these sets is our answer. In three dimensions, however, the sets are boxes which intersect our query box in the first dimension. The crucial insight here is that we may further process these sets for the second dimension by building another level of segment and range trees. That is, we build secondary trees for the projections of the boxes stored at each node of the first-level trees. We use the projection of our query box in the second dimension as query for the secondary trees. Finally, we build tertiary trees for the sets of boxes stored in the nodes of the secondary trees. This approach uses $O(n\log^3 n)$ space and finds all intersecting boxes in $O((n+m)\log^3 n + k)$ time. In the complete case, we use the above algorithm with $A = B$. As each box is represented as a point and an interval at the top level, we only need to build one tree, either the segment tree or range tree, at the top level. In the rest of the levels, we encounter bipartite problems and need both trees.

Streaming. We would like to use multi-level segment and range trees but the space requirements are prohibitive. While the *plane-sweep* technique [3, 19] may be used to shave a $\log n$ factor from both the running time and the space requirement of the above algorithm, we are still unsatisfied with the resulting $O(n\log^2 n)$ space requirement: we would like an algorithm which uses only linear space. Fortunately, we may use *streaming* to achieve this goal [6]. Suppose we perform all queries to our multi-level trees simultaneously, in other words, as a *batch*. The trees then will be traversed only once in post-order and therefore do not need to be stored in memory. Rather, we may construct each node as needed by using recursive procedures. Consequently, we have eliminated our trees and only need $O(n + m)$ space to store the boxes. Streaming the segment and range trees gives us two mutually recursive procedures. We may further simplify our algorithm by noting that calls to the recursive procedures have two sets of boxes as parameters, one set's projections viewed as intervals and the other's as points. Therefore, instead of streaming a range tree for the set viewed as intervals, we may stream a segment tree for the set viewed as points. The transformation makes essential use of the duality between intervals and points, as discussed before.

We define Stream3 as the procedure which streams a three-level segment tree for intersecting boxes. We initially call Stream3 two times, once for the segment tree and once for the dual of the range tree. The algorithm has $O((m+n)\log^3(m+n)+k)$ running time and takes $O(m + n)$ space. For the complete case, we only need to call Stream3 once to stream its segment tree in $O(n\log^3 n + k)$ time.

Computing medians. Stream3, like Quicksort [5], is a *divide-and-conquer* algorithm. In traversing a segment tree, Stream3 divides the segment of the current node $[lo, hi)$ into subsegments $[lo, mi)$ and $[mi, hi)$ for its children. In our approach, we choose mi from the input set of boxes being viewed as points. We need to choose mi judiciously in order to traverse a balanced segment tree. Initially, we used a variant of the *median-of-three* algorithm which was suggested for Quicksort by Singleton [20] and is part of almost any standard library implementation of Quicksort. More recently, however, we became aware of a fast approximate median finding algorithm by Clarkson et al [4]. While their work mostly examines approximating *centerpoints*, a higher dimensional equivalent of medians, they also apply their method in one dimension and give an algorithm for approximating medians (Algorithm 2 in the paper).

Intuitively, the algorithm constructs a ternary tree of height h with random points as leaves and evaluates it in a bottom-up fashion. Each internal node is the median of its three children (median-of-three) and the root is reported. We give a pseudo-code for a recursive implementation of this algorithm below in Figure 4.

```
Algorithm ApproxMedian(P, h):
1. if h = 0 then return a random p ∈ P;
2. return MedianOf3(
        ApproxMedian(P, h − 1),
        ApproxMedian(P, h − 1),
        ApproxMedian(P, h − 1)
   );
```

Figure 4: The ApproxMedian algorithm.

ApproxMedian returns an approximate median point from the set of points P by traversing a tree of height h. MedianOf3 computes the median of three points. We have found that using ApproxMedian improves the performance of our hybrid algorithm. This improvement stems partly from the pruning of the segment tree by one of our hybridization methods, as described in the next section.

We still need to determine the optimal height h for ApproxMedian. As far as we know, there has been no theoretical analysis of ApproxMedian within the context of a divide-and-conquer algorithm. For a fixed m, we find the best height h_m experimentally for m boxes. We then compute the height for a set of n points so that the same ratio of points is sampled. Specifically, $h_n = h_m + \log_3(n/m)$. We define the procedure Level(n) returning the computed height for n boxes.

4 The Hybrid Algorithm

In this section, we construct our hybrid algorithm from the procedures we defined in the last section. The basis of our algorithm is Stream3, which we combine with scanning for a faster algorithm. We do so using two hybridization techniques, which we will discuss next. Then, we fully flesh out our hybrid algorithm in pseudocode and show the impact of our methods using timed runs.

Using scanning. Our first hybridization technique is eliminating one level of the streamed segment tree with scanning. After streaming two levels of the segment tree, we are left with two sets of boxes called intervals and points which need to be intersected in dimension zero. We may use scanning to report all intersections in linear time. As each box is represented as both a point and an interval in each level (through the two calls to Stream3), we only need to use OneWayScan to scan. This technique, however, requires us to have the boxes in sorted order. We have two options:

1. We may sort the boxes before using the streamed segment tree and keep the sorted order through the two levels of the tree. This complicates the code and requires additional space for stable partitioning and merging of boxes. The running time drops to $O((m+n)(\log^2(m+n)) + k)$ with linear space. This option corresponds to Version One in our implementations.

2. We may sort just before scanning. This is equivalent to sorting $O(n\log^2 n + m\log^2 m)$ boxes in total and is theoretically slower. However, we do not need keep the order stable so we have simpler code and no need for additional space. The space and time complexity of the algorithm is the same as Stream3. This option corresponds to Version Two in our implementations.

Applying the first hybridization technique to Stream3, we obtain the hybrid algorithm Stream2-Scan.

Using cutoffs. We use scanning again for our second hybridization method. While the segment tree is a powerful algorithm, it has higher hidden constants than scanning. As a result, scanning is faster than the segment tree for low numbers of boxes even if the boxes are two- or three-dimensional (recall our discussion in Section 2.) This motivates us to scan whenever the number of boxes falls below a certain threshold or *cutoff*. We may do so with a single call, OneWayScan(I, P, d), where d is the dimension corresponding to the current level of the streamed segment tree.

When using option 1 for our first hybridization method, however, we may only scan in dimension zero where the boxes are sorted. Because of this, we are forced to use TwoWayScan to capture all possible box intersections in dimension zero. We need to modify TwoWayScan so that it simulates the one-way scan in dimension d by a one-way intersection test. That is, it reports a box intersection between $i \in I$ and $p \in P$ in dimension d only when i's projection in that dimension (viewed as an interval) contains p's projection (viewed as a point.) We call this new procedure ModifiedTwoWayScan.

To review, our hybrid algorithm, is a two-level streamed segment tree which uses scanning both for the dimension zero and for pruning the trees.

Procedure. Recall that the streamed segment tree is a post-order traversal of the segment tree. Therefore, we will have a call to a recursive procedure corresponding to each node of the segment tree. We will call our procedure Hybrid. The procedure will have as input a segment $[lo, hi)$ and two sets of boxes corresponding to the node's intervals I and query points P. We will distinguish between the first and second level segment trees by also passing an integer d denoting the dimension we are intersecting. We stream the segment tree in dimensions two and one and perform scanning in dimension zero.

For two sets of boxes A and B, the procedure is initially called as follows (corresponding to the roots of the two segment trees):

$$\textsc{Hybrid}(A, B, -\infty, +\infty, 2);$$
$$\textsc{Hybrid}(B, A, -\infty, +\infty, 2);$$

In the complete case with box set A, we have a single call:

$$\textsc{Hybrid}(A, A, -\infty, +\infty, 2);$$

The procedure is shown in Figure 5. We review the steps of the algorithm:

<u>Algorithm</u> $\textsc{Hybrid}(I, P, lo, hi, d)$:
1. `if` $I = \emptyset$ `or` $P = \emptyset$ `or` $hi \le lo$ `then`
   `return;`
   `endif;`
2. `if` $d = 0$ `then`
   $\textsc{OneWayScan}(I, P, 0)$;
   `endif;`
3. `if` $|I| < c$ `or` $|P| < c$ `then`
   $\textsc{ModifiedTwoWayScan}(I, P, d)$;
   `endif;`
4. $I_m = \{i \in I \mid [lo, hi) \subseteq i\}$;
   $\textsc{Hybrid}(I_m, P, -\infty, +\infty, d-1)$;
   $\textsc{Hybrid}(P, I_m, -\infty, +\infty, d-1)$;
5. $mi = \textsc{ApproxMedian}(P, \textsc{Level}(|P|))$;
6. $P_l = \{p \in P \mid p < mi\}$;
   $I_l = \{i \in I - I_m \mid i \cap [lo, mi) \neq \emptyset\}$;
   $\textsc{Hybrid}(I_l, P_l, lo, mi, d)$;
7. $P_r = \{p \in P \mid p \ge mi\}$;
   $I_r = \{i \in I - I_m \mid i \cap [mi, hi) \neq \emptyset\}$;
   $\textsc{Hybrid}(I_r, P_r, mi, hi, d)$;

Figure 5: The Hybrid algorithm.

1. This line corresponds to the base of our recursion, when we have an empty interval or point set or an empty segment.
2. This is our first hybridization method: scanning instead of the third level of the segment tree.
3. This is our second hybridization method: using cutoffs to switch to scanning.
4. In this line, we first generate I_m, the list of intervals that span the segment and hence would be stored at this node. We then make two calls for roots of the two segment trees at the next level.
5. ApproxMedian computes mi which divides the segment $[lo, hi)$ into subsegments $[lo, mi)$ and $[mi, hi)$.
6. This line corresponds to the left child of the current node. P_l is the set of points contained in the left subsegment $[lo, mi)$. I_l is the set of intervals that intersect the left subsegment $[lo, mi)$ but do not span the entire segment $[lo, hi)$.
7. This line corresponds to the right child of the current node. P_r and I_r are defined as in step 6 for the right subsegment $[mi, hi)$. Note that I_l and I_r are usually not disjoint.

First experimental observations. In Figure 6, we compare implementations of the different procedures we have presented so far for the complete case. Stream2-Scan does not perform much better than Stream3 so our first hybridization technique does not seem very effective. However, we believe it will have a more significant effect for larger number of boxes. The performance of Hybrid shows that our second technique is highly effective: by pruning the segment tree, we replace excessive partitioning by scanning, which is faster for low

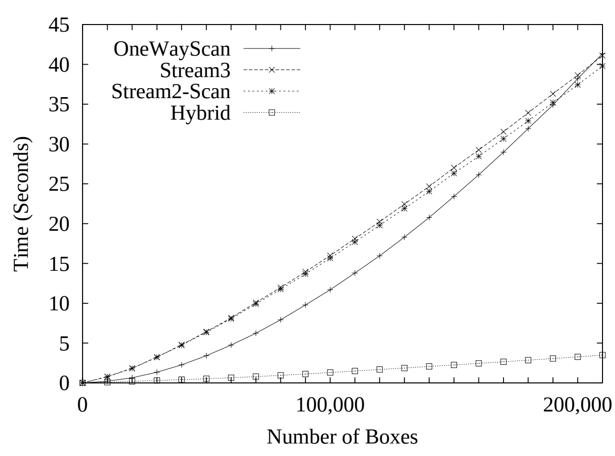


Figure 6: Comparison of Version One algorithms.

number of boxes. Finally, we note that the graphs for Stream2-Scan and OneWayScan would cross much lower (around 70,000 boxes) if the former was not employing ApproxMedian. In other words, using ApproxMedian is only effective for a pruned segment tree, as in Hybrid.

One issue we have not addressed is the determination of optimal cutoffs for Hybrid. Theoretical analysis of Hybrid for determining optimal cutoffs does not seem to be feasible. There are a large number of factors involved: the distribution of the input, the memory hierarchy and architecture of the machine, and so on. Currently, we can only determine the optimal cutoffs using experiments. We do so for the distribution we use for our timed experiments in the next section. We should point out two important facts, however. First, as scanning is being used in a divide-and-conquer algorithm, the optimal cutoff is much lower than the cross-over point of the graphs in Figure 6. Second, the optimal cutoff is highly dependent on the distribution of interval intersections for the projections of the input boxes in each dimension. Recall that higher-dimensional scanning, as discussed in Section 2, has running time which is dependent on k', the number of intersections in the scanned dimension. The optimal cutoff occurs roughly when we have a distribution of intersections in the scanned dimension for which scanning two times is more efficient than further traversal of a segment tree.

5 Distributions and Cutoffs

We begin the discussion of experiments by examining cutoffs in the context of different input distributions. After describing the distribution we use for most of our timed runs, we will explore Hybrid's performance for that distribution, find what the optimal cutoffs are, and discuss in some detail the reason why the cutoffs occur where they occur. Having gained a better understanding the way Hybrid works, we will define a family of distributions and use extremely skewed distributions to test Hybrid in worst case scenarios. Finally, we use distributions that we will more likely encounter to show the usefulness of Hybrid for applications.

The balanced distribution. In most of our timed runs, we used the following method for generating n boxes with a linear number of intersections. Recall that each box is the Cartesian product of three intervals. We choose the $3n$ intervals independently from each other by picking the low endpoint uniformly at random from 1 to $n - \lfloor n^{2/3} \rfloor$. The interval's length is chosen uniformly at random from 1 to $\lfloor n^{2/3} \rfloor$. Consequently, the expected length of the interval is $\lfloor n^{2/3} \rfloor/2$, two intervals intersect with probability about $1/n^{1/3}$, and two boxes intersect with probability about $1/n$. This gives us around $n/2$ box intersections in our set of n boxes. For bipartite problems, we generate two sets of $n/2$ boxes, giving us about $n/4$ intersecting pairs. As the probability of a pair of boxes intersecting is the same in each dimension, we call this distribution the *balanced distribution.*

Optimal cutoffs. We examine the behavior of Hybrid for the balanced distribution, allowing cutoffs to be set for each level of the segment tree independently. In Figure 7, we show the performance of Version Three, our implementation for the bipartite case, for 300,000 boxes. As the projected contour lines demonstrate, the

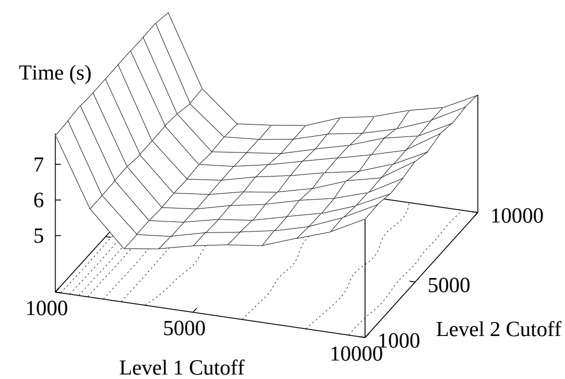


Figure 7: Performance of Version Three on 300,000 boxes for different cutoffs (average of five runs.)

algorithm is not influenced by the cutoff for the second level of the segment tree. This is because the second level is almost never used for the given cutoffs. Another immediate observation is that optimal cutoffs are very high, on the order of thousands.

Setting both cutoffs to the same value, we observe that the optimal cutoff for this distribution slides, that is, it is a function of the input size n, as seen in Figure 8. Intrigued, we decide to observe our implementation at work. By generating traces of Hybrid's traversal during execution, we examine the segment trees. In Figure 9, we show the first segment tree (of the two trees associated with two calls to Hybrid) for two runs of Version Three. Observe that with cutoffs set to 1,000, we never use the secondary segment trees as I_m, the set of intervals spanning a node's segment, is always empty. Hybrid is pruning the tree in such a way that the remaining nodes have $I_m = \emptyset$. When we lower the cutoff to 750, we reach nodes where $I_m \neq \emptyset$. However, as the set of points is often smaller than the cutoff at this point, the calls to the second level of the segment tree result in two calls to the scanning procedure (the gray nodes in Figure 9). This is highly inefficient because I_m is often very small, less than 10, compared to the

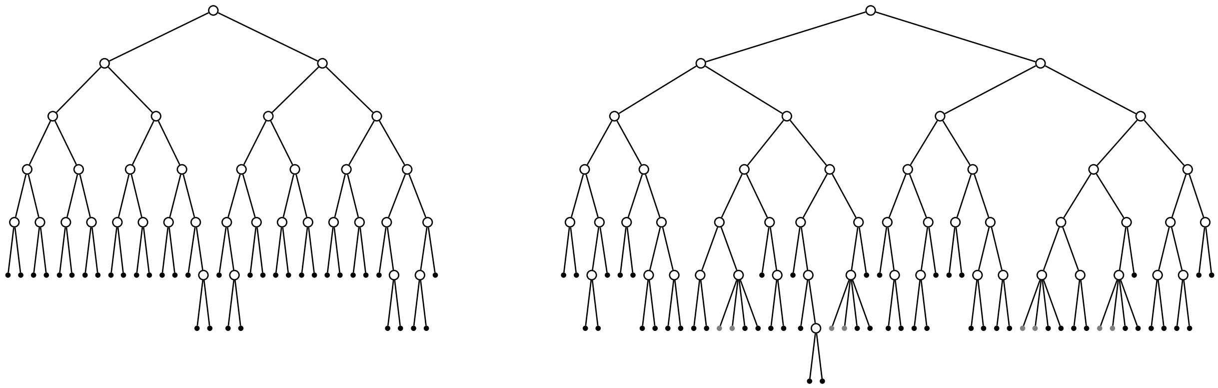


Figure 9: The segment trees streamed by Version Three for 50,000 boxes with cutoffs set to 1,000 (left) and 750 (right.) The segment tree nodes at the first level are colored white and the scanning nodes are colored black and gray for levels one and two, respectively.

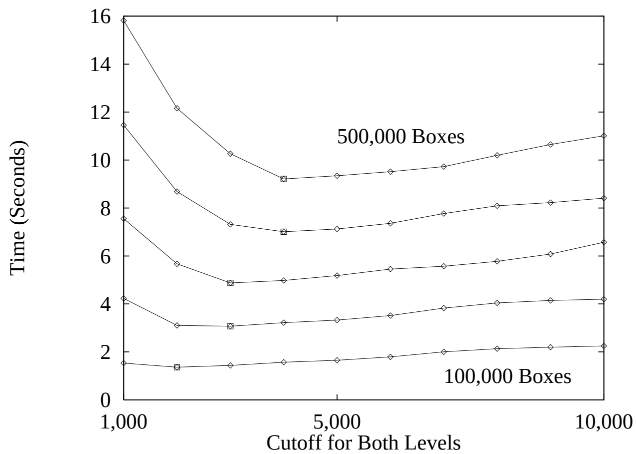


Figure 8: Performance of Version Three for input size from 100,000 to 500,000. The best time is boxed on the graph for each input.

set of thousands of points. By lowering the cutoff, we have added a level to the tree with potentially twice as many leaves resulting in up to four times as many calls to scanning. This inefficiency leads to the behavior we see in Figure 8. The optimal cutoff occurs just before calls are made to the secondary segment trees. This behavior, however, is directly related to the distribution: the probability of boxes intersecting is rather low in each dimension, leading to small I_m for most nodes in the tree.

Skewed distributions. Our observation prompts us to change the distribution so that $|I_m|$ is large. To do so, we define a family of distributions by extending our method for describing the balanced distribution.

Let $0 \leq p_0, p_1, p_2 \leq 1$ be real numbers so that for all n,

$$\frac{n^{p_0}}{n} \cdot \frac{n^{p_1}}{n} \cdot \frac{n^{p_2}}{n} \quad = \quad \frac{1}{n}. \tag{1}$$

Equivalently, $p_0 + p_1 + p_2 = 2$. For n boxes, we will choose the length of the projection of a box in dimension i uniformly at random from 1 to $\lfloor n^{p_i} \rfloor$ so that the probability of two boxes intersecting is about n^{p_i}/n. Equation (1) states that the probability of any two boxes intersecting in all dimensions is $1/n$, leading to about $n/2$ intersecting pairs in the complete case and about $n/4$ for the bipartite case as in the balanced distribution. Having defined this family of distributions, we look at Hybrid's performance for several highly skewed distributions (Table 1.) In each case, we show the performance for the optimal cutoff c_{opt} for that distribution, as well as the optimal cutoff (2,000) for the balanced distribution.

#	p_0	p_1	p_2	c_{opt}	$t_{c_{opt}}$ (s)	t_{2000} (s)
1	1	1	0	8	1.10	26.31
2	1	0	1	scan	991.60	?
3	0	1	1	scan	0.61	6.08
4	1	1/2	1/2	150	2.28	16.85
5	1/2	1	1/2	750	0.88	0.89
6	1/2	1/2	1	scan	1.81	7.44

Table 1: Performance of Version Three on 100,000 boxes for various distributions. c_{opt} is the optimal cutoff and t_c is the time in seconds for Hybrid running with cutoff $= c$.

Distributions 1–3 represent the worst case scenario for Hybrid. Every pair of boxes generated from these distributions intersect in two dimensions. As Hybrid scans in dimension zero and places the first level segment tree in dimension two, it will only be able to perform well if it can access the differentiating dimension where the probability of boxes intersecting is just $1/n$.

This is true for distribution 1 and 3. By using a very low cutoff, we take advantage of the first level segment tree for distribution 1, eliminating non-intersecting boxes. For distribution 3, scan is the optimal algorithm as dimension zero has the lowest number of intersections.

Distribution 2, however, is problematic. Generally, each node of the segment tree lowers the number of points and intervals. This is not true for this distribution as the number of intervals remains high throughout the first level. As such, the segment tree becomes extremely inefficient. Scanning is also very inefficient, as $k' = O(n^2)$. The only way of achieving reasonable performance for this distribution is detecting that the segment tree is failing to divide the set of intervals and restarting the algorithm in another dimension. Prior knowledge about the box distribution would tell us to scan in dimension two.

Distributions 4–6 represent scenarios we would more likely encounter. Note that Hybrid performs quite well but is sensitive to the cutoff used.

The sphere distribution. Having examined highly skewed distributions, we would like to know how Hybrid would perform in our application domains. We are mainly interested in using our box intersection tools on surfaces embedded in $\mathbb{R}^3$. We test Hybrid by generating boxes from the following *sphere distribution*: we uniformly sample the surface of a sphere of radius $n/2$ and center a box at each sample, choosing the length of each side independently and uniformly at random from 1 to $\lfloor\sqrt{n}\rfloor$. This gives us $O(n)$ box intersections because of the manifold property of the sphere. As Figure 10 demonstrates, Hybrid performs extremely well for a large range of cutoffs. Both Figure 8 and Figure 10 show that Hybrid is very forgiving when it comes to cutoffs for box distributions that have $o(n^2)$ box intersections in each dimension.

Conclusion. The most important conclusion we have derived from our experiments is that Hybrid is very sensitive to distribution of intersections in each dimension. While Hybrid is unable to perform well for highly skewed distributions, it performs extremely well for distributions we would likely encounter in practice. Any prior knowledge of the box distribution would help us in setting cutoffs for optimal performance.

6 Version One and Sorting

Having examined cutoffs, we now turn our attention to the details of our four implementations. Our implementations are in strict `ANSI-C`. We use GNU's `gcc` version 2.7 for compiling and the `-O3` and `-funroll-loops` options for final optimization. We give details about the machines we use for our timing tests in Table 2. For measuring memory usage, we assume we are using 32-bit machines, currently dominant in the industry, with 4-byte integers and 4-byte pointers.

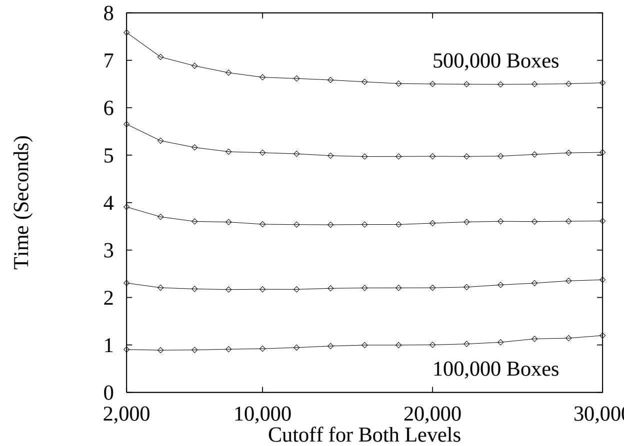


Figure 10: Performance of Version One for input size from 100,000 to 500,000 generated from the Sphere Distribution.

Version One. Our first implementation is for the complete case of box intersections. Our data structure for a box simply stores the six integer endpoints using 24 bytes per box. We give our declarations in Figure 11. To use Hybrid, we need to distinguish between intervals and points. We do so with two arrays of pointers to the box data structure for an additional 8 bytes per box.

```
typedef struct {
  int lo, hi;
} intervalT;

typedef intervalT boxT[3];
```

Figure 11: Structure declarations for Version One.

In Section 4, we pointed out that our first hybridization strategy, using scanning in dimension zero, entailed having the intervals and points in sorted order and we had a choice of two options to do so. We decided to adopt the first option for this implementation, that is, we sort our sets at the beginning in the dimension zero and keep the various sublists sorted throughout the calls to our procedure by stable partitioning and merging. Our scanning procedures then do not need to sort the intervals and points again.

While our pseudo-code in Hybrid uses set notation for describing the subsets of intervals and points, in actuality we use in-place partitioning to avoid using extra

Maker	Name	Processor	Level-1 Cache		Level-2 Cache		Main Memory	
			Size	AT	Size	AT	Size	AT
Sun	Ultra-30	300 MHz UltraSPARCII	16/16 KB	29 ns	2 MB	60 ns	128 MB	134 ns
SGI	Indigo-2	200 MHz MIPS R4000	16/16 KB	31 ns	1 MB	93 ns	128 MB	307 ns
Sun	Ultra-1	143 MHz UltraSPARC 1	16/16 KB	60 ns	512 KB	127 ns	64 MB	174 ns
Sun	Sparc-5	170 MHz TurboSPARC	unified 16 KB	65 ns	none	n/a	80 MB	168-501 ns

Table 2: Machines used for our timing experiments. Level-1 Cache sizes refer to separate instruction and data caches. AT is average time for a read and write to an integer found at that level of memory hierarchy. See Section 6 for discussion.

memory. However, to do stable partitioning, we need a temporary array of pointers for an additional 4 bytes per box. Also, we need to put both sets of intervals and points back in sorted order at the end of our recursive procedure by merging the sublists we have created. That is, we add an additional step to Hybrid:

8. Put I in sorted order by merging its sublists.
   Put P in sorted order by merging its sublists.

In total, our data structure for Version One uses 36 bytes per box. We show the data structure for four boxes in Figure 12.

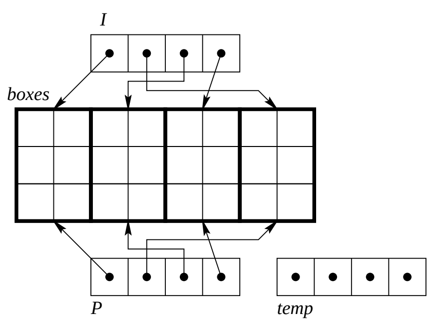


Figure 12: Version One data structure for four boxes.

Sorting. One issue we still have not fully discussed is the method to sort the intervals and points. At the beginning of this project, we sorted the pointers in the array I and copied them to P. This was in keeping with traditional Computer Science teachings as copying 24-byte structures is obviously slower than copying 4-byte pointers. However, we discovered that sorting the box structures was, in fact, faster than sorting the pointers (31% faster for 500,000 boxes.) More significantly, our implementation of Hybrid, unchanged otherwise, ran up to 39% faster after sorting structures as shown in Figure 13.

Sorting structures is faster than sorting pointers because the former generates a pattern of memory access which exhibits more spatial locality than the latter. Modern memory hierarchies are organized around

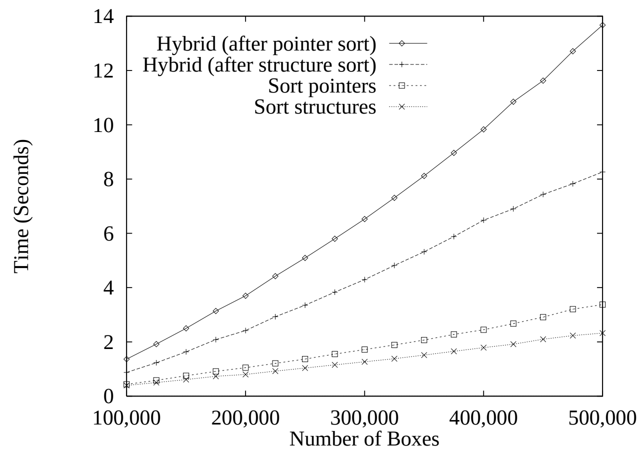


Figure 13: Partial times of Hybrid (Version One) for sorting structures versus sorting pointers.

the principle of *locality of reference* and provide us with fast multi-level caches and increasingly wider and faster memory buses [10]. These combine to allow for fast copying of large structures. Sorting by pointers causes nonlocal access to memory which disallows taking full advantage of the memory system. Even though the program has to do less copying (4 bytes versus 24 bytes in our case), it copies at a slower rate, yielding an overall slower program.

One may argue that once structures are large enough, sorting by pointers will be faster as the time used to copy the structures will dominate the sorting time. Figure 14 confirms this conjecture. Note that the time for sorting pointers is essentially constant for a fixed number of structures regardless of the structure size as we always copy about the same number of pointers while sorting. The sorting time consists mostly of cache miss penalties. The time for sorting structures increases linearly with the size of the structure as expected.

Cross-overs. The graphs cross at a *cross-over point* after which sorting pointers becomes faster. We call the maximum size for which sorting structures is faster *cross-over size*. We compute the cross-over size for our platforms on idle machines as shown in Figure 15.

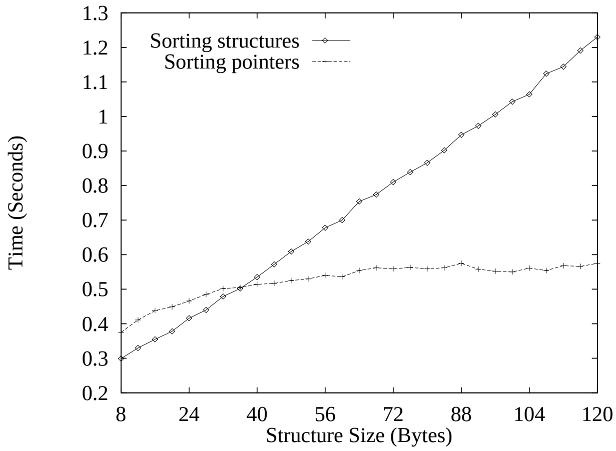


Figure 14: Sorting 100,000 structures. The time is the average of 10 runs using structures with randomly chosen keys.

Observe that the cross-over size increases with the size

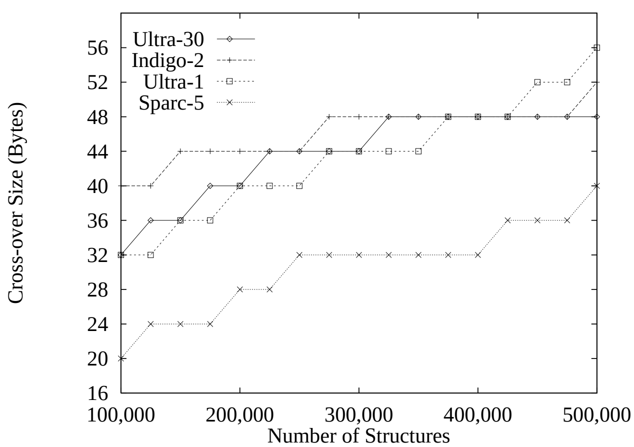


Figure 15: Cross-over size for our platforms.

of the input as pointers to the structures are wider apart, making pointer sorting slower. Also, for a fixed structure size, if sorting structures is faster than sorting pointers for some input size n, then sorting structures will be faster for all input size $m \geq n$. In fact, it will be increasingly faster. This was already demonstrated in Figure 13. What determines the cross-over size on different platforms for a fixed number of structures is the performance of memory hierarchy on each platform. To evaluate the behavior of the memory systems of our platforms, we timed reading and writing to an integer on each system. We had a program stride through memory to invoke the different levels of the hierarchy, using a variant of the program given for Exercise 5.2 in [10]. We include the resulting data for a stride of 24 bytes in Table 2. Observe that different platforms have wildly different cache-miss penalties. As the Indigo-2 is a much older machine, its main memory is more than twice as slow as the Ultra-30. This is the reason the cross-over size for Indigo-2 is higher than for the Ultra-30 in Figure 15. It is more important for the Indigo-2 to find its data in its caches than it is for the Ultra-30.

Consequences. We noted that HYBRID runs faster after sorting structures than sorting pointers, as shown in Figure 13. The significant speedup is caused once again by more efficient cache usage. After we sort the structures, our I and P pointers reference memory in a cache-coherent fashion, as shown in Figure 16 (compare with Figure 12.) This has two consequences:

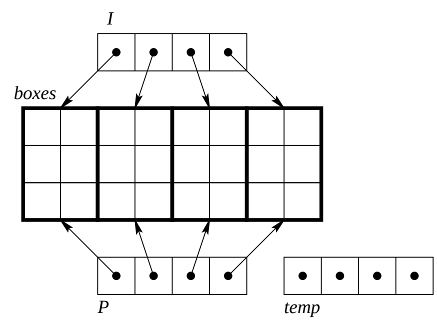


Figure 16: Version One data structure for four boxes after structure sort.

1. The initial partitioning steps are cache-coherent and therefore faster. Without a structure sort, the partitioning would access the entire *boxes* array in a more or less arbitrary fashion.
2. While partitioning moves the pointers, it is stable. The pointers always access memory in a monotonic fashion. In other words, within each sublist, the arrows in Figure 16 never cross. The monotonic access to memory allows for better cache usage than the random access.

As a result of the structure sort (a change of about three lines of `C` code), HYBRID runs almost twice as fast as before.

7 Versions Two, Three, Four

The crucial lesson we learned from our experiments with sorting and our quantitative studies was that non-local access can slow any algorithm significantly. As all non-local access is through pointers, we would like to eliminate all pointers to take full advantage of fast caches.

Designing Version Two. We used pointers in our data structure for Version One to represent intervals and points with a common array of boxes (see Figure 12.) Eliminating the pointers means we have to duplicate the data structure for boxes for an additional 24 bytes per box, although this duplication would obviate the need for the two arrays of pointers, saving 8 bytes per box. But our temporary array would also need to be an array of box structures for another 24 bytes per box.

There are further complications. In our first version, we did not number the boxes because they never moved in memory and we could identify them by their index in the *boxes* array. With the duplication of boxes and movement of data, we also need to number each box for yet another 4 bytes per box. The total memory usage, $28*3 = 74$ bytes per box, compared to 36 bytes per box in Version One, is not feasible.

One way to reduce the memory usage is the elimination of the temporary array. Without the temporary array, we can no longer keep the box array sorted through stable partitioning and merging. Recall that we need the intervals and points sorted in order to use scanning (see discussion in Section 4.) Therefore, we have to switch to our second option: we sort just before scanning. Having eliminated the temporary array, our data structure takes 56 bytes per box, a 56% increase relative to Version One. We give the new box declaration in Figure 17.

```
typedef struct {
  intervalT projections[3];
  int num;
} boxT;
```

Figure 17: Structure declarations for Version Two. Each box is stored in two such structures.

Consequences. While the second option for scanning is theoretically slower than the first, it highly simplifies the program. Partitioning points and intervals does not need to be stable any longer and can be implemented simply as Quicksort's partition step. Our partitioning is much like PARTITION in [5]. We do need to swap entire structures, however, when partitioning the sets. On the other hand, we do not need to merge either set, eliminating Version One's step 8 from the algorithm. Overall, our new approach reduces the code size for the segment tree to half its size in Version One.

Another consequence of the new approach is that HYBRID becomes tail recursive. We may apply Sedgewick's optimizations for Quicksort [18] to the algorithm, removing the tail recursion and guaranteeing $O(\log n)$ stack depth. In practice, we found these not to speed up the program, probably because the compiler already removes the tail recursion.

Comparison. We designed Version Two to fully use (in a sense, abuse) fast caches in a memory hierarchy. We compare the two versions on all our platforms in Figure 18. We computed a performance ratio t_1/t_2 for each platform, where t_i denotes the time taken by version i.

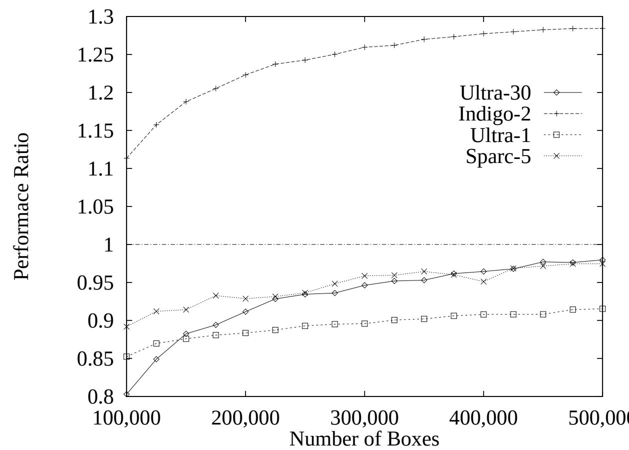


Figure 18: Ratio of time for Versions One and Two.

Version Two performs much better than Version One on one platform: the SGI Indigo-2. This stems from the high cache-miss penalty on this machine. On the Sun Ultra-1, Version Two performs much worse, probably because the machine has a relatively small Level-2 cache. On the other two platforms, initially the second version performs worse but achieves about 97% of the performance of Version One for 500,000 boxes.

Version Three. We may modify either Version One or Two easily for the bipartite case. We choose to modify Version Two as it seems to have good performance and it will have lower memory usage for the bipartite case than Version One.

For the bipartite case, we have two different input sets of boxes which do not need to be duplicated. Therefore, we only need 28 bytes per box for storage. We need to modify the program to have two initial calls to HYBRID (see Section 4.) We call this implementation *Version Three* and will give some timed results for it later in this section.

Version Four. When our boxes are cubes, we may take advantage of the additional symmetries cubes offer to reduce memory usage. For each cube, we store only the vertex with the lowest coordinate in each dimensions along with the length of the cube's side. We give the

```
typedef struct {
  int lo[3];
  int side;
  int num;
} cubeT;
```

Figure 19: Structure declaration for Version Four.

`C` declaration in Figure 19. We only need 20 bytes per cube or about 29% less memory than Version Three. We modify Version Three to create *Version Four* for intersecting cubes.

To compare Version Four with Version Three, we create a bipartite set of n cubes using the same distribution we used for boxes. We use the cubes as input to both versions, viewing them as cubes and boxes respectively. Figure 20 shows that the memory savings in Version Four translate into time savings, as less memory is traversed and copied.

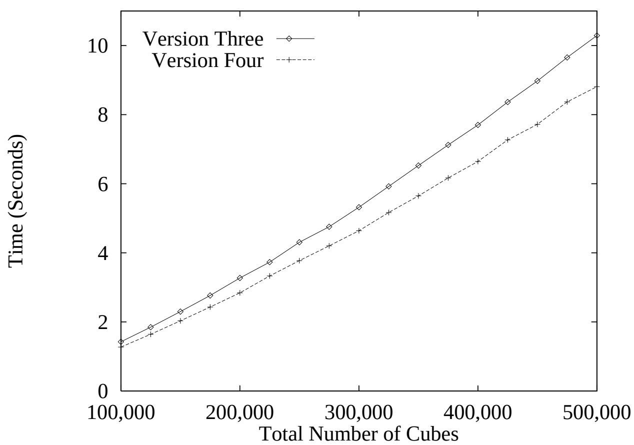


Figure 20: Comparison of Versions Three and Four for cubes.

As always, the running time is averaged over 10 runs for each n. Version Four is about 14% faster than Version Three.

Bounding cubes. We may use Version Four even when our input sets consist of boxes by enclosing our boxes with bounding cubes. This strategy is really a continuation of our original strategy: putting bounding volumes around geometric primitives. This approach clearly leads to more intersections, however, leading to slower performance for Version Four. We compare Version Three intersecting sets of boxes and Version Four intersecting bounding cubes for the same sets of boxes in Figure 21. In our applications, we will often perform a calculation for the geometric primitives inside the intersecting boxes. More intersections mean more

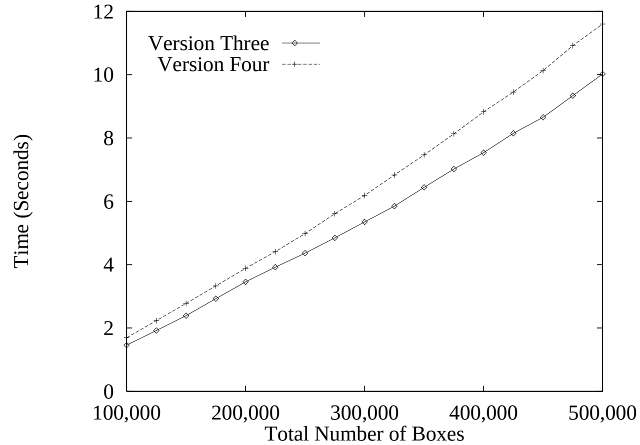


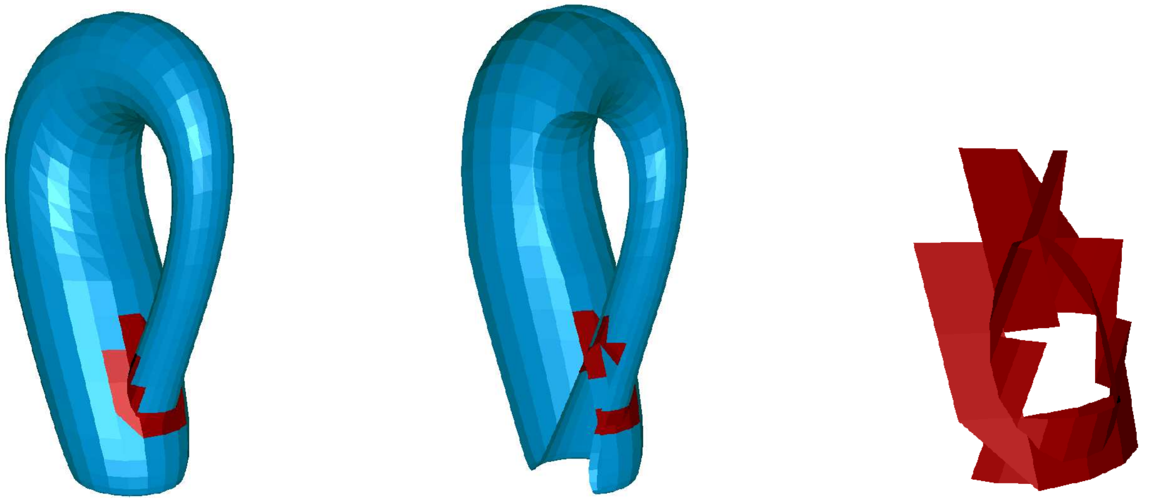
Figure 21: Comparison of versions three and four for boxes and their bounding cubes.

calculations, leading to slower overall programs. When memory is limited, however, Version Four could be a useful tool.

8 Two Applications

Having detailed our implementations, we return to our original motivation for this work: solving problems involving geometric primitives using our box intersection tools as fast engines. In this section, we demonstrate the utility of our work by briefly discussing two immediate applications of our implementations.

Surface self-intersection. Within the fields of Solid Modeling and Computer Graphics, a three-dimensional surface is often represented by a *triangulation* or a patchwork of *NURBS* [7]. Both fields often use heuristics for reconstructing surface models from point sets and cases unaccounted for often lead to the creation of surface self-intersection. Surface self-intersection is also caused by surface modification algorithms at times. We desire to detect such self-intersections. Computing whether two surface elements intersect, however, is expensive and thus motivates us to take a hierarchical approach using bounding volumes. We reduce the number of potential candidates for intersection by enclosing the geometric primitives by bounding boxes and intersecting the boxes. This problem involves reporting pairwise intersections between boxes in a single set and falls into the complete case in our categorization. We use a variant of Version One to find the intersecting box pairs and then test the geometric primitives inside each pair for intersection. This algorithm has been implemented as part of *geomagic® Studio*, a software product from Raindrop Geomagic. We show our algorithm at work in Figure 22.



(a) Entire model (b) Model cut in half. (c) The intersecting triangles.

Figure 22: The Klein bottle illustrates our first application of box intersection. All immersions of the Klein bottle in three dimensions, including the one shown above have self-intersections. The 66 intersecting triangles, rendered in dark gray, were detected by our algorithm in *geomagic® Studio*.

Distance computation. Algorithms for surface reconstruction have become increasingly robust recently and have led to the generation of large triangle meshes. The complexity of these meshes has outstripped the improvements in hardware and has led to substantial interest and research in simplification of triangle meshes with minimum geometric (and in some cases topological) change. Furthermore, other modification algorithms have been proposed for *refinement*, *thickening*, and *relaxation* of triangle meshes. One approach to objectively assess the performance of these algorithms is to find the "distance" between the original mesh and the modified mesh. That is, we find the Euclidean distance between the vertices of one mesh and the elements describing the surface in the other mesh. We have developed a fast three-phase algorithm which computes the distance between two surfaces and uses box intersection in each phase. In particular, we use Version Three as the problem falls into the bipartite category. We briefly describe the algorithm below, as a full exposition is beyond the scope of this paper.

The algorithm takes two meshes $M_1 = (V_1, E_1)$ and $M_2 = (V_2, E_2)$ as input, where V_i denotes the vertices and E_i the elements of M_i. We note, however, that the algorithm requires only V_1, so a point set may be substituted for V_1 for certain applications such as determining the quality of a surface reconstruction from a point set. For each $v \in V_1$, the algorithm finds a good upper bound estimate $D(v)$ to the distance $d(v) = \min_{x \in |M_2|} \|v - x\|$ in Phases I and II, where $|M_2|$ denotes the set of all points defined by the mesh M_2. This estimate is used to find $d(v)$ in Phase III.

Phase I. We enclose points in V_1 with infinitesimal axis-aligned cubes C_1 and enclose elements in E_2 with axis-aligned cubes C_2. We use a variant of Version Three to intersect C_1 and C_2 and compute for each intersecting pair $(c_1, c_2) \in C_1 \times C_2$ the distance of the vertex inside c_1 to the element inside c_2, keeping the minimum distance $D(v)$ for each vertex $v \in V_1$.

Phase II. Let $R \subseteq V_1$ be the set of vertices which did not obtain an estimate in Phase I. Enclosing R with infinitesimal cubes C_R and doubling the length of the sides of the cubes C_2, we intersect C_R and C_2 and compute distances as in Phase I. We repeat this phase until all vertices $v \in V_1$ have assigned estimates $D(v)$.

Phase III. We define C_1 to be a set of cubes created by centering a cube with side length $2D(v)$ around each $v \in V_1$. Enclosing E_2 with axis-aligned boxes B_2, we intersect C_1 and B_2 using Version Three and compute distances as in the previous phases, obtaining $d(v)$ for each $v \in V_1$.

We show the performance of this algorithm on the Stanford dragon in Figure 23. The dragon mesh has 437,645 vertices and 871,414 triangles. We decimated the mesh using *geomagic® Decimate* and computed the distance of the vertices of the decimated mesh to the original surface using our algorithm. Note that the algorithm is slower for the 1% than the 5% mesh, as there are more vertices requiring distances in Phase II. In Figure 24, we compare the Stanford bunny and its decimated

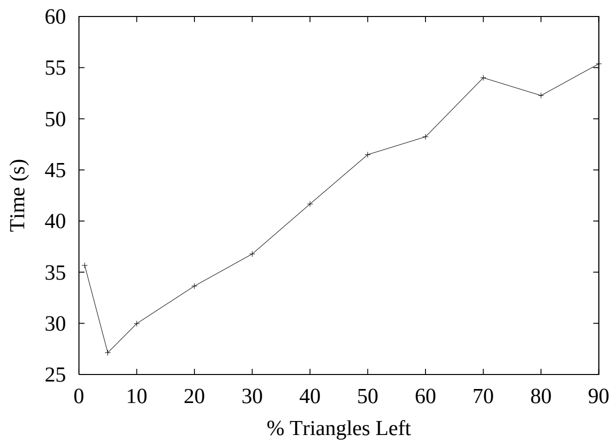


Figure 23: Performance for computing distances between a series of simplified versions of the Stanford Dragon and the original surface.

version using our algorithm.

9 Summary and Conclusions

Table 3 gives a listing of our implementations, when they are useful, what they intersect, and how much memory they consume per box. As our implementations

Version	Case	Objects	Bytes Per Object
One	Complete	Boxes	36
Two	Complete	Boxes	56
Three	Bipartite	Boxes	28
Four	Bipartite	Cubes	20

Table 3: Our four implementations of HYBRID.

take about the same time for the balanced distribution, we include the performance of just one version, Version One, on all of our platforms in Figure 25.

To conclude, we believe the contributions of this paper are:

1. an approach emphasizing both theory and practice toward design and implementation of fast algorithms,
2. industrial strength implementations of a fast algorithm to be used as tools for geometric computing,
3. evidence of the viability of the algorithm through large suites of tests on multiple platforms,
4. an in-depth look at the behavior of the segment tree and its hybrids with input generated from various distributions,

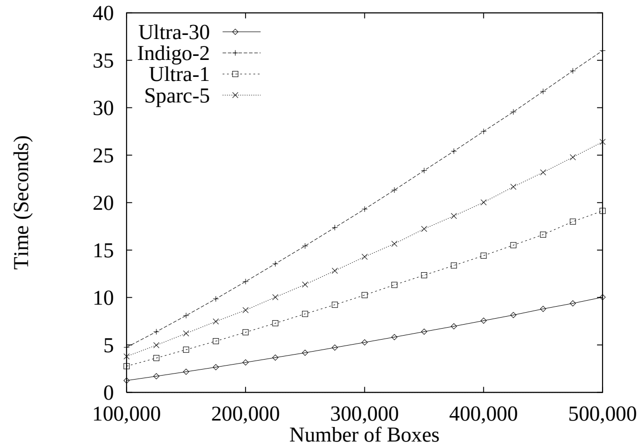


Figure 25: Performance of Version One on four platforms.

5. quantitative analyses leading to faster sorting.

References

[1] BENTLEY, J. L. Multidimensional search trees used for associative searching. *Comm. ACM* **18** (1975), 509–517.

[2] BENTLEY, J. L. Multidimensional divide and conquer. *Comm. ACM* **23** (1980), 214–229.

[3] BENTLEY, J. L., AND WOOD, D. An optimal worst case algorithm for reporting intersections of rectangles. *IEEE Trans. Comput.* **C-29**, 7 (1980), 571–577.

[4] CLARKSON, K. L., EPPSTEIN, D., MILLER, G. L., STURTIVANT, C., AND TENG, S.-H. Approximating center points with iterative radon points. *Internat. J. Comput. Geom. Appl.* **6**, 3 (1996), 357–377.

[5] CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. *Introduction to Algorithms.* The MIT Press, Cambridge, MA, 1994.

[6] EDELSBRUNNER, H., AND OVERMARS, M. H. Batched dynamic solutions to decomposable searching problems. *J. Algorithms* **6** (1985), 515–542.

[7] FOLEY, J. D., VAN DAM, A., FEINER, S. K., HUGHES, J. F., AND PHILLIPS, R. L. *Computer Graphics: Principles and Practice*, second ed. Addison Wesley, Reading, MA, 1996.

[8] GOTTSCHALK, S., LIN, M., AND MANOCHA, D. OBB-tree: A hierarchical structure for rapid interference detection. In *SIGGRAPH 96 Conference Proceedings* (1996), pp. 171–180.

[9] GUTTMANN, A. R-trees: a dynamic index structure for spatial searching. In *SIGMOD Conference Proceedings* (1984), pp. 47–57.

[10] HENNESSY, J. L., AND PATTERSON, D. A. *Computer Architecture: A Quantitative Approach*, second ed. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1989.

[11] Knuth, D. E. Theory and practice, IV. In *Selected Papers on Computer Science*. CSLI Publications, Stanford, CA, 1996.

[12] Krishnan, S., Pattekar, A., Lin, M., and Manocha, D. Spherical shell: A higher order bounding volume for fast proximity queries. In *Proceedings of the Third International Workshop on Algorithmic Foundations of Robotics* (1998), pp. 122–136.

[13] Latombe, J. C. *Robot Motion Planning*. Kluwer Academic Publishers, Norwell, MA, 1991.

[14] Lin, M. C., and Gottschalk, S. Collision detection between geometric models: a survey. In *Proceedings of the IMA Conference on Mathematics of Surfaces* (1998).

[15] Matoušek, J. Geometric range searching. *ACM Comput. Surveys* **26**, 4 (1994), 21–461.

[16] Naylor, B., Amanatides, J., and Thibault, W. Merging BSP trees yield polyhedral modeling results. In *SIGGRAPH 90 Conference Proceedings* (1990), pp. 115–124.

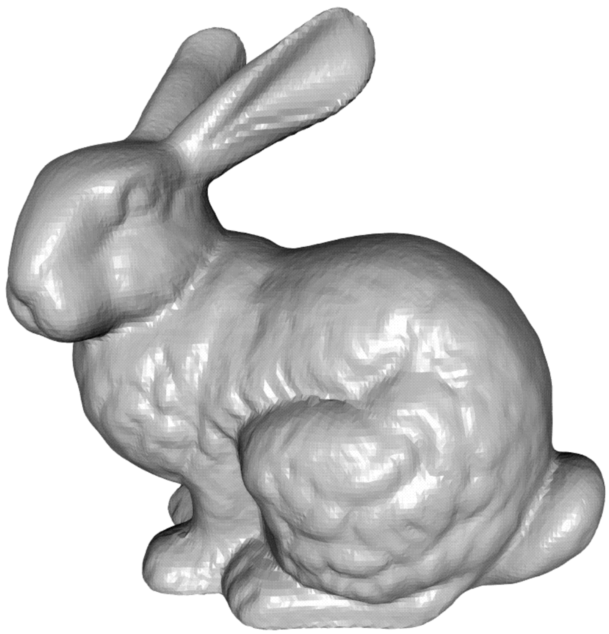
[17] Samet, H. *Spatial Data Structures: Quadtree, Octrees, and Other Hierarchical Methods*. Addison Wesley, Reading, MA, 1989.

[18] Sedgewick, R. *Algorithms in C: Fundamentals, Data Structures, Sorting, Searching*. Addison Wesley, Reading, MA, 1997.

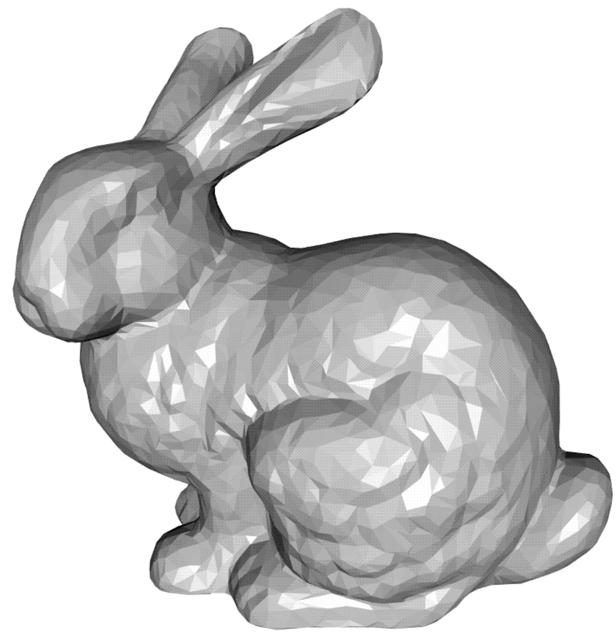
[19] Shamos, M. I., and Hoey, D. Geometric intersection problems. In *Proc. 17th Ann. IEEE Sympos. Found. Comput. Sci.* (1976), pp. 208–215.

[20] Singleton, R. C. An efficient algorithm for sorting with minimal storage (Algorithm 347). *Comm. ACM* **12** (1969), 185–187.

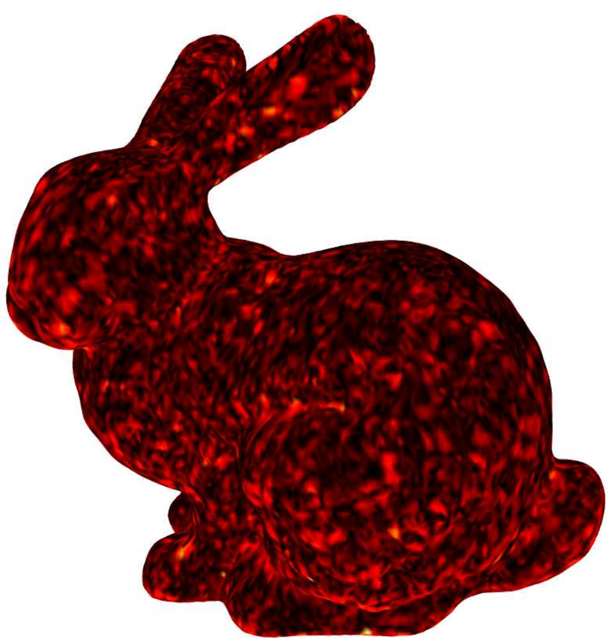
[21] Zhou, Y., and Suri, S. Analysis of a bounding box heuristic for object intersection. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms* (1999).



(a) Original mesh (34,834 vertices and 69,451 triangles)



(b) Mesh decimated to 10% using *geomagic*® *Decimate* (3,581 vertices and 6,945 triangles)



(c) Distance of vertices of original mesh to decimated mesh. The algorithm ran in 1.48 seconds.

(d) Distance of vertices of decimated mesh to original mesh. The algorithm ran in 1.41 seconds.

Figure 24: The Stanford bunny illustrates our second application of box intersection. The distance is mapped using the *hot* colormap, where the zero distance is black, the maximum distance for the mesh is white, and the object gets "hotter" with increasing distance, first turning red, then yellow, and finally white. Note: the mesh is not a closed manifold.