Discrete Comput Geom 8:335–360 (1992)

Discrete & Computational
Geometry

Dynamic Point Location in Arrangements of Hyperplanes*

Ketan Mulmuley[1] and Sandeep Sen[2]

[1] Computer Science Department, University of Chicago, Chicago, IL 60637, USA

[2] I.I.T. Delhi, India

Abstract. We present algorithms for maintaining data structures supporting fast (polylogarithmic) point-location and ray-shooting queries in arrangements of hyperplanes. This data structure allows for deletion and insertion of hyperplanes. Our algorithms use random bits in the construction of the data structure but do not make any assumptions about the update sequence or the hyperplanes in the input. The query bound for our data structure is $\tilde{O}(polylog(n))$, where n is the number of hyperplanes at any given time, and the $\tilde{O}$ notation indicates that the bound holds with high probability, where the probability is solely with respect to randomization in the data structure. By high probability we mean that the probability of error is inversely proportional to a large degree polynomial in n. The space requirement is $\tilde{O}(n^d)$. The cost of update is $\tilde{O}(n^{d-1} \log n)$. The expected cost of update is $O(n^{d-1})$; the expectation is again solely with respect to randomization in the data structure. Our algorithm is extremely simple.

We also give a related algorithm with optimal $\tilde{O}(\log n)$ query time, expected $O(n^d)$ space requirement, and amortized $O(n^{d-1})$ expected cost of update. Moreover, our approach has a versatile quality which is likely to have further applications to other dynamic algorithms.

For $d = 2, 3$ we also show how to obtain polylogarithmic update time in the CRCW PRAM model so that the processor-time product matches (within a polylogarithmic factor) the sequential update time.

1. Introduction

Maintaining data structures that allow periodic updates has received much attention in the past and in recent years. Typical operations include insertion and

* Ketan Mulmuley was supported by NSF Grant CCR 8906799 and a Packard Fellowship. Part of this work was done when Sandeep Sen was in AT&T Bell Laboratories, Murray Hill, NJ 07974, USA.

deletion of elements from a given universe like points, segments, etc., and at any given stage we may have to answer queries about the present set of elements. One of the challenging goals in designing data structure for such dynamic environments is to be able to match the query time with that of the static case (one in which the set of elements remains fixed but each instance of query could be different). At the same time it is also critical that we do not expend too much space for the data structure and also keep the update time minimal. Balanced binary trees supporting dictionary operations is perhaps the most commonly used dynamic data structure and it also matches the asymptotic performance of searching in an ordered set. In order to compete with the static case, the dynamic data structures typically need to be more sophisticated and sometimes turn out to be prohibitively difficult to implement. Examples of some sophisticated dynamic data structures include data structures for planar point location [5], [14], [18], [19].

A more recent line of attack for designing dynamic data structures has been the use of randomization. The term randomized algorithms in this paper refers to algorithms that do not assume any distribution of the input but use random bits to make choices at different stages of the algorithm for any input. Skip Lists [20] and Randomized Search Trees [2] are examples of dynamic data structures recently proposed and use randomization. Their performance bounds compare very favorably with their deterministic counterparts (that is the balanced binary trees) and are much simpler to implement. The obvious tradeoff is that the performance bounds are guaranteed with certain probabilities which in spite of being less than 1 are usually acceptable for most applications. In particular, if we can guarantee performance bounds with probability $1 - 1/n^{\alpha}$ for a large enough $\alpha > 1$, where n is the input size, then even for moderate values of n this is very close to 1. Bounds of this form are often referred to in the literature as *high-probability* bounds. These are stronger than bounds on the mean behavior, which cannot predict the probability of deviation from the *expected* behavior. The following notation is used in this paper. We say that a function $f(n) = \tilde{O}(g(n))$ if, for every $\alpha \geq 1$, there is a constant $c > 0$ such that $f(n) \leq c \cdot g(n)$ with probability at least $1 - 1/n^{\alpha}$.

In this paper we investigate further the use of randomization for searching in arrangements of hyperplanes in a dynamic environment. The static point-location problem for arrangements of hyperplanes has been satisfactorily solved [7] by making use of randomization and then subsequently derandomizing it efficiently [3], [4]. Our results are as follows. We give a very simple dynamic point-location algorithm with $\tilde{O}(polylog(n))$ query time and $\tilde{O}(n^d)$ space requirement and $\tilde{O}(n^{d-1} \log n)$ update time, for arbitrary d. The expected update time is $O(n^{d-1})$ and the expected space requirement is $O(n^d)$. In dimension two we obtain the optimal $\tilde{O}(\log n)$ query time; the other bounds are the same as before. We can reduce the query time to $\tilde{O}(\log n)$, with $O(n^d)$ expected space and $O(n^{d-1})$ expected, amortized update time, for arbitrary d. However, the bounds on space and update time for this algorithm are only expected and do not hold with high probability.

For $d = 2, 3$ we also show how to obtain polylogarithmic update time in the CRCW PRAM model so that the processor-time product matches (within a polylogarithmic factor) the sequential update time.

Our data structures can also be used for dynamic ray shooting with the same query time. See the note at the end of this paper.

Random sampling results in [7], [15], and [22] have contributed significantly toward our arrangement searching algorithms.

Notation. In this paper we use $|\ |$ to denote the size operation. Thus if N is a set, $|N|$ denotes its size, if f is a convex polytope, $|f|$ is the number of all its subfaces, and so on.

2. The Basic Algorithm

In this section we present a high-level, dimension-independent description of our basic approach. Some steps of our algorithm are dependent on the dimension. We present the implementation of these steps in later sections where we instantiate our basic algorithm in various dimensions.

We begin by describing a procedure for building a point-location data-structure in the static case and subsequently argue that its extension to the dynamic situation is straightforward. The static algorithm is reminiscent of an algorithm due to Clarkson [7] turned upside-down. Given a set N of hyperplanes in R^d, we denote the induced arrangement by $G(N)$. The d-cells of $G(N)$ can have an unbounded number of facets and this turns out to be problematic. Hence, we work with a certain triangulation $H(N)$ of $G(N)$ that is obtained by decomposing each d-cell of $G(N)$ into simplices or, in general, cells with a bounded number of facets. We leave the exact nature of $H(N)$ completely abstract at this point, except that it is assumed to satisfy the following condition: each d-cell f of $G(N)$ is decomposed into $O(|f|)$ simplices, or in general cells, each of which is "defined" by a bounded number of hyperplanes. As an abuse of notation, we refer to the d-cells of $H(N)$ as d-simplices, even though, strictly speaking, they need not be simplices.

The following basic algorithm builds a point-location structure $\tilde{H}(N)$ that can be used to locate the d-simplex of $H(N)$ containing any query point $p \in R^d$.

Let $N = N_1$. $\tilde{H}(N) = \tilde{H}(N_1)$ is defined recursively as follows:

1. Build the triangulation $H(N_1)$.
2. For each hyperplane in N_1, toss an unbiased coin. Let N_2 be the set of hyperplanes in N_1 for which the toss turned out to be a head. Build $\tilde{H}(N_2)$ recursively.
3. Associate with each d-simplex Δ of $H(N_2)$ a list $L(\Delta)$ of hyperplanes in $N_1 \backslash N_2$ that intersect Δ and conversely with each hyperplane in $N_1 \backslash N_2$, we associate a list of d-simplices in $H(N_2)$ that it intersects. We also say that the hyperplanes in $L(\Delta)$ conflict with Δ and $L(\Delta)$ is called its conflict list.
4. Build a data structure *Descent*(2, 1) that provides a "descending link" between the second level and the first level. This structure is used in point-location queries, in a manner to be described soon. At this stage we leave the nature of this descent structure completely abstract.

An important fact regarding our point-location structure is that, for every $l > 1$, N_l is a random sample of N_{l-1} of roughly half the size. Hence, the random

sampling results in [7] and [15] imply that, with very high probability, for every d-simplex Δ of $H(N_l)$ and every $l > 1$, $|L(\Delta)| = \tilde{O}(\log n)$. In what follows we denote the size of N_l by n_l.

Now let us see how to answer point-location queries. Let $p \in R^d$ be any fixed query point. Our goal is to locate p in $G(N) = G(N_1)$, the arrangement associated with the first level. Toward this end we recursively "locate" p in the second level. We assume that we are given a *descent oracle* so that, given how p is located in the second level, p can be located in the first level quickly, i.e., in polylogarithmic time, assuming that the oracle has the descent structure *Descent*(2, 1) as well as the conflict information at its disposal. As the number of levels in $\tilde{H}(N)$ is easily seen to be $\tilde{O}(\log n)$, this implies $\tilde{O}(polylog(n))$ bound on the query time. Of course, we have proven this bound for a *fixed* query point, but as we shall see later, this easily translates into a polylogarithmic bound for any query point, because there will be only polynomially many distinct search paths in our data structure. To get a tighter bound on the query time, such as $\tilde{O}(\log n)$ bound in dimension two, we need to use refined random sampling results that are proven later in this paper.

So far we have deliberately not stated in precise terms what is meant by locating p in the ith level. There seem to be several ways of defining what this means. The first possibility is to define locating p in the ith level as simply determining the d-cell in the arrangement $G(N_i)$ containing p, but then it is easy to see that descending from the ith level to the $(i-1)$st level is going to be difficult in general, because the d-cells of $G(N_i)$ can have an arbitrarily large number of facets. The second possibility is to define locating p in the ith level as locating the d-simplex of the triangulation $H(N_i)$ containing p. This notion is stronger than the first notion, because given the d-simplex of $H(N_i)$ containing p, we can immediately determine the d-cell of $G(N_i)$ containing p. The third notion is to define locating p in the ith level as determining the hyperplanes in N_i above and below p with respect to, say, the x_d-coordinate. At this point, let us keep the notion of locating p in the ith level completely abstract.

To make our data structure dynamic, we adopt the following scheme. Our procedures for addition and deletion of a hyperplane are such that, at any given time, the state of our data structure is independent of the actual sequence of updates that built it. Thus if N were to denote the set of currently existing hyperplanes that have been added but not deleted so far, then $\tilde{H}(N)$ will be as if it were built by the above static procedure applied to N. This ensures that the random sampling results that are crucial to analyze our static data structure carry over, more or less unaffected, to the dynamic setting.

Let us now see how to add a new hyperplane h to $\tilde{H}(N)$. We first toss an unbiased coin successively until we get a tail. Let j be the number of heads obtained before getting a tail. We simply "add" h to levels 1 through $j+1$. For $1 \leq l \leq j+1$, let $\bar{N}_l$ denote $N_l \cup \{h\}$. Addition of h to the lth level is carried out in three steps.

1. Update $H(N_l)$ to $H(\bar{N}_l)$.
2. Construct conflict lists of the new d-simplices in $H(\bar{N}_l)$.
3. Update *Descent*$(l+1, l)$.

The third step is dependent on the exact nature of the descent structures. Hence, we only elaborate the first two steps.

The zone of a hyperplane (in d dimension) is defined as follows. Let h_0 be a hyperplane in an arrangement $G(H)$. A k-face f for $0 \leq K \leq d-1$ is said to be visible from h_0 if there is a line segment s that connects f and h_0 such that the interior of s is contained in h_0 or in a cell of $\mathscr{A}(H)$. The *zone* of h_0 is the set of k-faces that are visible from h_0. Define $Zone(N_l, h)$, the *Zone* of h in the arrangement $G(N_l)$. The Zone Theorem in [13] states that

Theorem 1 (Zone Theorem). *The maximum cardinality of* $Zone(N_l, h)$ *is* $O(n_l^{d-1})$, *where* n_l *is the size of* N_l, *and, moreover,* $Zone(N_l, h)$ *can also be determined in* $O(n_l^{d-1})$ *time.*

Let f be any d-cell in $Zone(N_l, h)$. We remove all d-simplices in the old triangulation of f. Next we split f along h into two d-cells f_1 and f_2 and triangulate f_1 and f_2 all over. All triangulation schemes to be considered in this paper are simple enough so that triangulation of f_1 and f_2 can be carried out in $O(|f_1| + |f_2|) = O(|f|)$ time.

We also need to construct conflict lists of all d-simplices in the triangulation of f_1 and f_2. Let h' be a hyperplane in $N_l - N_{l-1}$ that intersects f. From the old conflict information, we can figure out all 1-faces (edges) of f intersecting h'. Hence, by a straightforward search in the new triangulations of f_1 and f_2, we can determine all d-simplices within f_1 and f_2 that intersect h' in time proportional to their number. Because the size of every conflict list, new or old, is $\tilde{O}(\log n)$, with high probability, it follows that the total cost of updating the conflict lists is $\tilde{O}(\sum_f |f| \log n) = \tilde{O}(n_l^{d-1} \log n)$, where f ranges over all d-cells of $G(N_l)$ intersecting h.

To summarize:

Lemma 1. *The cost of inserting a new hyperplane in* $\tilde{H}(N)$ *is* $\tilde{O}(n^{d-1} \log n)$, *ignoring the cost of updating the descent structures.*

Deletion is the exact reversal of addition, that is, the cost of deletion is no more than inserting the hyperplane immediately afterward. Hence, we merely state:

Lemma 2. *The cost of deleting any hyperplane from* $\tilde{H}(N)$ *is* $\tilde{O}(n^{d-1} \log n)$, *ignoring the cost of updating the descent structures.*

Our broad objective in dimension d is to obtain a polylogarithmic search time and $O(n^{d-1} \cdot polylog(n))$ update time, where n denotes the number of hyperplanes currently in the data structure. In the next section we consider the simplest case, $d = 2$, to bring out the basic ideas in a simplest setting. We show how to achieve $\tilde{O}(\log n)$ query time and $\tilde{O}(n \log n)$ update time for $d = 2$. In Section 4 we consider the general dimension. The case $d = 2$ is a little bit different from the general dimension in that we can make use of the low dimensionality of the problem to achieve optimal $\tilde{O}(\log n)$ time in a very simple way.

3. Two-Dimensional Arrangements

Let N be a set of n lines in R^2 and let $G(N)$ denote the induced arrangement. The convex regions of $G(N)$ need not have a bounded number of sides. Hence, using a well-known scheme, we decompose each convex region of $G(N)$ into vertical trapezoids. From each vertex of the convex region (polygon) extend a vertical ray directed toward the interior until it meets an edge of the polygon (see Fig. 1(a) and (b)). This partitions the convex polygon into trapezoids. (If required these

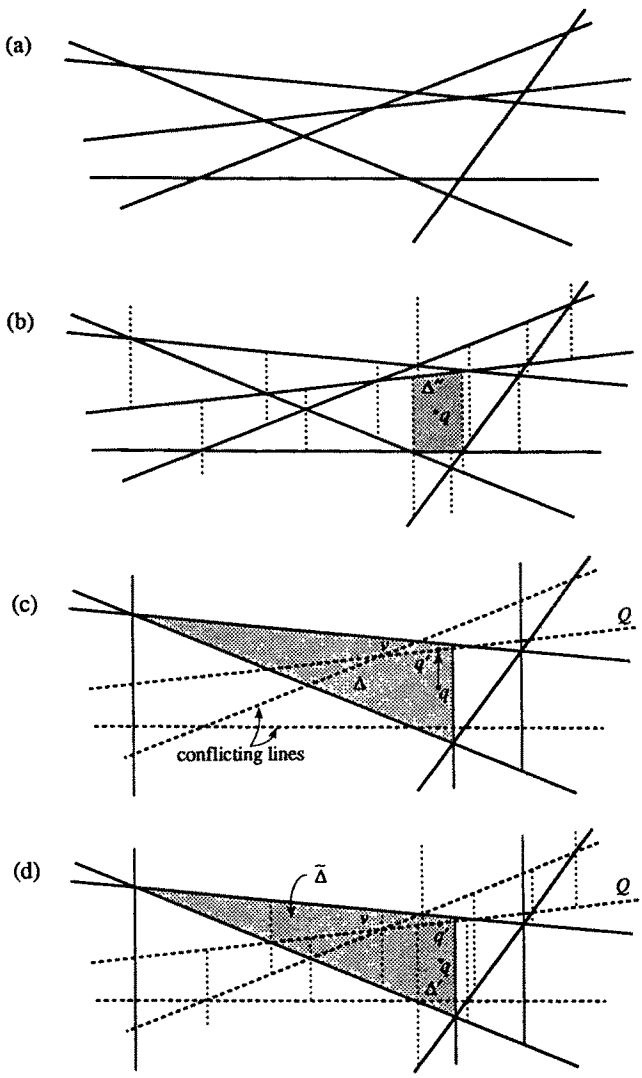

Fig. 1. (a) N_l, (b) $H(N_l)$, (c) $H(N_{l+1})$, and (d) *Descent*$(l+1, l)$.

trapezoids can be triangulated by drawing a diagonal, but this is not necessary, since each trapezoid is obviously "defined" by a bounded number of lines.) When the above procedure is repeated for all convex regions of $G(N)$ we get the triangulation $H(N)$ that we use in our basic algorithm.

The only thing that remains to be specified in the definition of our search structure $\tilde{H}(N)$ is the nature of the descent structures. The descent structure *Descent*$(l+1, l)$ between two successive levels $l+1$ and l is defined as simply the superposition of the triangulations $H(N_l)$ and $H(N_{l+1})$; see Fig. 1. We also denote this superposition by $H(N_l) \oplus H(N_{l+1})$. We also associate with each trapezoid in *Descent*$(l+1, l)$ a pointer to the trapezoid in $H(N_l)$ containing it.

Let us turn to point location. Let $q \in R^2$ be a fixed query point. Let r be the last level in our data structure, which means that N_r is empty. Thus locating q in $H(N_r)$ is trivial. Inductively assume that we have located q in $H(N_{l+1})$, $1 \leq l < r$. Making use of the descent structure defined above, it is really easy to descend from level $l+1$ to level l. Let $\Delta = \Delta_{l+1}$ be the trapezoid in $H(N_{l+1})$ containing the query point q. We determine the first line Q in N_l that intersects the vertical ray from q directed upward; see Fig. 1(c). Obviously Q is either the line bounding the upper side of Δ or it belongs to $L(\Delta)$. Let q' be the point of intersection of the vertical ray with Q. Let v be the intersection of Q with either a line in $L(\Delta)$ or the boundary of Δ, which is nearest to q' on its left side. It is easy to determine v in $O(|L(\Delta)| + 1)$ time. Next we walk within *Descent*$(l+1, l)$ from v to q'; see Fig. 1(d). The cost of this walk is again $O(|L(\Delta)| + 1)$. At the end of this walk, we have determined the trapezoid $\Delta' \in$ *Descent*$(l+1, l)$ containing q; see Fig. 1(d). The required trapezoid $\Delta'' = \Delta_l \in H(N_l)$ containing q is the one containing Δ'; see Fig. 1(b).

Thus we can descend from level $l+1$ to level l in $O(|L(\Delta_{l+1})| + 1)$ time. With high probability, $|L(\Delta_l)| = O(\log n)$ for all l, and the number of levels is $O(\log n)$. It follows that the time required to locate a fixed point q is $\tilde{O}(\log^2 n)$. The following theorem shows that the query time is, in fact, $\tilde{O}(\log n)$.

Theorem 2. *For a fixed query point* q, $\sum_{i \geq 1} |L(\Delta_i)|$ *is* $\tilde{O}(\log n)$, *where* Δ_i *is the trapezoid containing* q *in* $H(N_i)$.

Proof. Let $NB(s)$ denote the random variable that is equal to the number of tails obtained before obtaining the sth head in a sequence of binomial trials with a fair coin. $NB(s)$ is the familiar negative binomial distribution. When $s = 1$, it is the geometric distribution. We show that, for all i, $|L(\Delta_i)| = O(NB(a))$ for some fixed constant a. Because the coin tosses at each level, used in the definition of data structure, are independent from the coin tosses used in the preceding levels, it then follows that, for any fixed constant c, $\sum_{i \leq c \log n} |L(\Delta_i)| = O(NB(ca \log n)) = \tilde{O}(\log n)$, using Chernoff bound [6] for negative binomial distributions. As the number of levels is $\tilde{O}(\log n)$, this proves the theorem. □

So fix a level i. Also fix the set N_i of lines occurring in the ith level of the data structure. The set N_{i+1} is determined by flipping a fair coin for each line in N_i and retaining those lines for which the toss was head. We prove that:

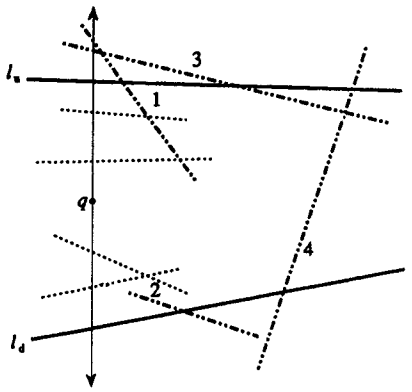

Fig. 2. ·····, lines in U and D; ···—···, lines in R_q (numbers indicate ordering on R_q).

Lemma 3. *There is an imaginary, on-line ordering $h_1, h_2, \ldots$ of all lines in N_i such that the set of lines "defining" or intersecting the trapezoid Δ_{i+1} always occurs as an initial subsequence of $h_1, h_2, \ldots$. By on-line ordering we mean that h_{k+1} can be chosen on the basis of the known coin toss results for $h_1, \ldots, h_k$. Note that Δ_{i+1} is not known to us a priori, because it depends on the results of coin tosses for the lines in N_i.*

As the number of lines defining any trapezoid is at most four, it follows from the lemma that $|L(\Delta_{i+1})|$ is $O(NB(4))$.

Proof of the lemma. Consider the ordered set V_u of lines (in the increasing Y direction) in N_i intersecting the vertical line extending upward from query point q. See Fig. 2.

Initially we toss coins for these lines in V_u, in the increasing Y direction away from q, until we obtain a head, and then (temporarily) stop. Let l_u be the line for which we obtained a head. Let $U \subseteq V_u$ denote the set of lines before l_u for which we obtained tails. Clearly, $l_u \in N_{i+1}$, whereas no line in U belongs to N_{i+1}. Thus l_u is obviously going to be bounding the top of the trapezoid Δ_{i+1}, which we do not know completely as yet. Moreover, all lines in U obviously conflict with Δ_{i+1}.

Now we resume our coin tossing, in a symmetric manner, for the lines in N_i intersecting the vertical line extending downward from q, until we obtain a head, and then we again stop temporarily. Let D be the set lines for which we obtained tails and let l_d be the line for which we obtained a head. Obviously, l_d is going to be bounding the bottom of the trapezoid Δ_{i+1}, which we know partially by now.

Now discard (hypothetically) the lines in U and D and consider the intersections of the remaining lines with l_u and l_d. Let R_q be the set of remaining lines that intersect either l_u or l_d to the right of the vertical line through q. We order R_q as follows. Given two lines l_1 and l_2 in R_q, we say that $l_1 \ll l_2$, if the y-coordinate of either $l_1 \cap l_u$ or $l_1 \cap l_d$ is less than the y-coordinates of both $l_2 \cap l_u$ and $l_2 \cap l_d$. Figure 2 shows ordering of R_q. Now we resume tossing coins for the lines in R_q

in the increasing order, until we obtain a head. Let l_r be the line for which we obtained a head. It is then clear that l_r defines the right-hand side of Δ_{i+1} in the sense that the right-hand side of Δ_{i+1} extends from the intersection of either l_u or l_d with l_r. Moreover, all lines for which we obtained tails conflict with Δ_{i+1}.

Now discard (hypothetically) the lines in R_q too. Let L_q be the set of remaining lines intersecting either l_u or l_d to the left of the vertical line through q. We order L_q in a symmetric fashion, and resume tossing coins for the lines in L_q in the increasing order (away from q) until we get a head and then temporarily stop. Let l_l be the line for which we obtained a head. It is clear that it "defines" the left-hand side of the trapezoid Δ_{i+1}, and all lines for which we obtained tails conflict with Δ_{i+1}.

At this point the trapezoid Δ_{i+1} containing q in the $(i+1)$st level has been completely determined. Indeed l_u, l_d, l_r, l_l are the lines defining Δ_{i+1} and the lines for which we obtained tails so far are precisely the lines in conflict with Δ_{i+1}. (We did not take into account the exceptional cases such as when Δ_{i+1} is unbounded or when it is, in fact, a triangle. However, a slight modification to the argument will cover these cases too.)

We can now toss coins for the remaining lines in any order whatsoever. It follows that the above on-line sequence of tosses has the desired property. □

In the above theorem we showed that the query time is $\tilde{O}(\log n)$ for a fixed query point. We further note that there are only polynomially many distinct combinatorial search paths for a given data structure. By combinatorially distinct, we imply a different sequence of triangles in the search path. More precisely, let $\tilde{G}(N)$ be the refinement of $G(N)$ obtained by passing infinite vertical lines through all intersections among the lines in N. Then, for a fixed region R in $\tilde{G}(N)$, it is easy to see that the search path in $\tilde{H}(N)$ remains the same if the query point lies anywhere in R. This implies that the cost of locating any point is $\tilde{O}(\log n)$.

To bound the space requirement of our search structure, first note that, for any $l \geq 1$, the size of the descent structure $H(N_{l+1}) \oplus H(N_l)$ is $O(n_{l+1}^2)$. This follows because, by the Zone Theorem, each line in N_l intersects $O(n_{l+1})$ trapezoids in $H(N_{l+1})$. Thus the total space requirement of our data structure is easily seen to be $O(\sum_l n_l^2) = \tilde{O}(n^2)$. Actually it is easy to ensure that the space requirement is $O(n^2)$ (worst case) without affecting the query time. For this we ensure that r, the number of levels in the data structure, is such that $\sum_{n=1}^{r} n_i^2 \leq b \cdot n^2$ for some large enough constant b; the levels (if any) higher than the maximum permissible value of r are not maintained. During point location we locate the query point in $H(N_r)$ trivially (in $O(|N_r|)$ time) and then descend through the data structure as before. If b is chosen large enough, with high probability this "clipped" data structure coincides with the nonclipped data structure, as defined before, and hence, with high probability, the query time also remains unaffected.

Now let us estimate the cost of adding or deleting a line. We only consider addition, because deletion is the reversal of addition. By Lemma 1, we only need to worry about the cost of updating the descent structures *Descent*$(l+1, l)$, $1 < l \leq j+1$, where j is the number of successive heads obtained. Consider first the simpler case when $l = j+1$. In this case the line h is to be added to N_l but

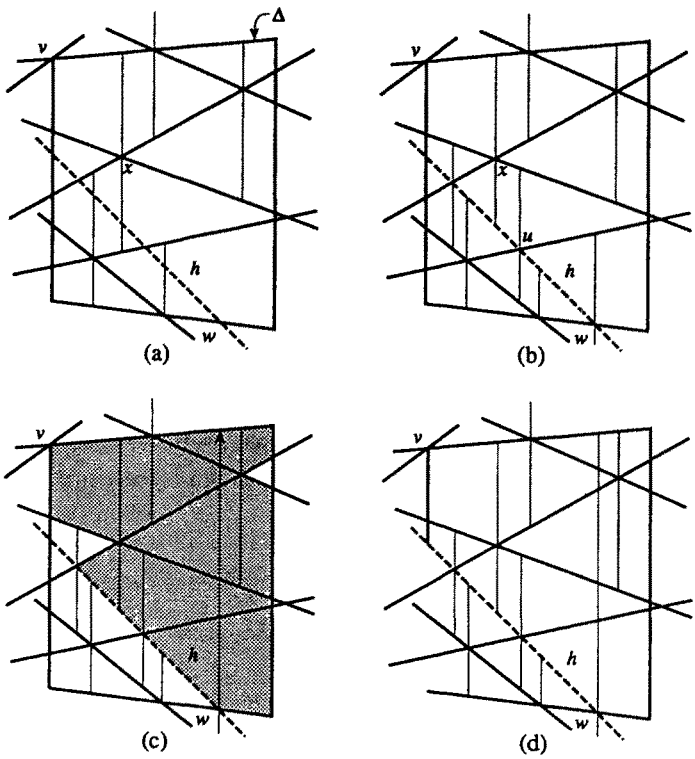

Fig. 3. Addition of a new line h.

not to N_{l+1}. Let $\Delta \in H(N_{l+1})$ be any trapezoid intersecting h. For every such trapezoid Δ, we need to update the restriction $\tilde{\Delta}$ of *Descent*$(l+1, l)$ to Δ. This is done as follows (see Fig. 3). First we add h to the trapezoidal decomposition $\tilde{\Delta}$. This means all vertical attachments in $\tilde{\Delta}$, such as the one through the intersection x in Fig. 3(a), are split and vertical attachments through new intersections on h, such u and w in Fig. 3(b), are added. The cost of adding h to $\tilde{\Delta}$ is $O(|L(\Delta)| + 1)$. This easily follows by applying the Zone Theorem to the restricted arrangement $G(N_l) \cap \Delta$.

Consider now the case when $l < j + 1$. In this case, h is to be added to N_l as well as N_{l+1}. To take this into account, we only need to extend the above procedure of adding h to $\tilde{\Delta}$ as follows. First, if h intersects the lower (or upper) border of Δ, then the vertical attachment through this point of intersection w must extend to the opposite border of Δ (Fig. 3(c)). This is because w belongs to the new trapezoidal decomposition associated with the $(l+1)$st level and hence the vertical attachment through w cuts through the lines in $L(\Delta)$ which do not belong to this level. The cost of adding such an extended vertical attachment through w is clearly dominated by the size of its zone in the restricted arrangement $G(N_l') \cap \Delta$

(shown shaded in Fig. 3(c)), where $N_l' = N_l \cup \{h\}$. By the Zone Theorem this cost is $O(|L(\Delta)| + 1)$. Finally, if h intersects the left (or the right) border of Δ, then that border has to split appropriately as shown in Fig. 3(d). The reason is this border corresponds to a vertical attachment through some intersection v in the old decomposition $H(N_{l+1})$, and hence occurs in the new decomposition associated with the $(l + 1)$st level in a split form.

It follows that the cost of updating the restriction of *Descent*$(l + 1, l)$ to any trapezoid $\Delta \in H(N_{l+1})$ is $O(|L(\Delta) + 1) = \tilde{O}(\log n)$. By the Zone Theorem the number of trapezoids in $H(N_{l+1})$ intersecting h is $O(n_{l+1})$. Hence, the total cost of updating *Descent*$(l + 1, l)$ is $\tilde{O}(n_{l+1} \log n)$. Summing over all levels, it follows that the total cost of updating the descent structures is $\tilde{O}(\log n \sum_l n_l) = \tilde{O}(n \log n)$. Using the results in [8], we can show that, for all i, the average conflict size of the trapezoids in $H(N_i)$ intersecting h is $O(1)$ (the average is taken over all trapezoids in $H(N_i)$ intersecting h). This immediately implies that the expected cost of update is $O(n)$.

There is another method for bounding the cost of updating *Descent*$(l + 1, l)$, which is interesting in its own right. It is based on the observation that the cost of updating *Descent*$(l + 1, l)$ is obviously bounded, up to a constant factor, by the total number of vertices in $G(N_l)$ which lie in the trapezoids of $H(N_{l+1})$ intersected by h. It follows from the following lemma that this number is $\tilde{O}(n_l \log n)$. This lemma turns out to be useful later in Section 6.

Lemma 4. *The total number of vertices in $G(N_l)$, which lie in the trapezoids of $H(N_{l+1})$ intersected by h, is $\tilde{O}(n_l \log n)$.*

Proof. Define the *vertical distance* of any vertex $v \in G(N_l)$ from h as the number of lines in N_l that intersect the open vertical segment joining v and h. It is clear that the vertical distance of any vertex of $G(N_l)$, lying in a trapezoid of $H(N_{l+1})$ intersected by h, is bounded by the conflict size of that trapezoid, which is $\tilde{O}(\log n)$. Hence, it suffices to bound $\sum_{j=1}^{k} s_j$, where $k = O(\log n)$ and s_j denotes the number of vertices in $G(N_l)$ at a vertical distance j from h. By the Zone Theorem, $s_0 = O(n_l)$. This, in conjunction with the results in [8] on abstract k-sets, implies that $\sum_{j=1}^{k} s_j = O(kn_l)$. □

We summarize our main result as follows:

Theorem 3. *Let $G(N)$ be an arrangement of n lines in a plane. There exists a dynamic point-location data structure of $O(n^2)$ size allowing $\tilde{O}(\log n)$ query time which also allows insertion/deletion of lines in $\tilde{O}(n \log n)$ time. The expected cost of update is $O(n)$.*

Remark. Using the best-known deterministic schemes for dynamic point location [5], [19], we can achieve $O(\log^2 n)$ and $O(n \log n)$ bounds for search and update times, respectively. These are considerably more involved procedures.

4. Extension to Higher Dimensions

In this section we extend the algorithm in Section 3 to arbitrary dimension. Our algorithm works by induction on the dimension d. For the basis case $d = 2$, we use the algorithm in Section 3.

In general dimension the algorithm follows the same basic scheme as in Section 2. However, the triangulation scheme that is used is somewhat different in nature. Let N be a set of hyperplanes in R^d, and let $G(N)$, as before, denote the induced arrangement. The triangulation $H(N)$ of $G(N)$ that we use is the so-called bottom-vertex triangulation that is defined as follows. We triangulate the j-faces of $G(N)$, $j \leq d$, by induction on j. If $j = 1$ this is trivial. Otherwise, let f be any j-face of $G(N)$, $j \geq 2$. Let v denote the vertex of f with the smallest x_d coordinate; it is possible that v lies at "infinity."[1] By our inductive hypothesis, all facets of f have been triangulated. So we simply extend the "simplices" on the boundary of f to cones with apex at v. This gives us a simple triangulation of f. When all j-faces of $G(N)$ are triangulated in this fashion, we get the triangulation $H(N)$ that we sought. $H(N)$ is called an x_d triangulation or bottom-vertex triangulation of $G(N)$. The reader might wonder why we did not use this triangulation scheme in Section 3. The reason is that, for some subtle reasons, the superposition scheme used there for defining the descent structures fails. It is possible to define alternative descent structures in this case, but then the cost of point location goes up to $\tilde{O}(\log^2 n)$.

Let us now define the point-location structure $\tilde{H}(N)$ to be associated with the arrangement $G(N)$. We use induction on d. When $d = 2$ we use the point-location structure in Section 3. For $d > 3$ we apply the scheme in Section 2, with the triangulation $H(N)$ as defined above. We only need to describe the descent structures. *Descent*$(i, i-1)$ contains a dynamic point-location structure for the $(d-1)$-dimensional arrangement $G(N_{i-1}) \cap Q$ for each hyperplane $Q \in N_{i-1}$. We denote this structure by $\tilde{H}(N_{i-1}, Q)$. When $d = 3$ the planar point-location structure $\tilde{H}(N_{i-1}, Q)$ is defined as in Section 3, with some minor modifications described below. For $d > 3$ it is the $(d-1)$-dimensional point-location structure that has already been defined, because of our inductive hypothesis. Maintenance of $H(N_{i-1}, Q)$ is done by recursively applying our lower-dimensional point-location algorithm.

It immediately follows from Lemma 1, Lemma 2, and simple recurrence equations that the cost of adding or deleting a hyperplane is $\tilde{O}(n^{d-1} \log n)$. $O(n^{d-1})$ bound on the expected cost of update easily follows if we were to use, as in Section 3, the results in [8] on average conflict size. Arguing as in dimension two, it is easily seen that the size of our data structure is $\tilde{O}(n^d)$. In fact, we can also ensure that the worst-case space requirement is $O(n^d)$, by applying the *clipping* procedure described in Section 3.

It remains to see how to answer point-location queries. Let $p \in R^d$ be a fixed query point. We cannot use our point-location structure to locate the d-simplex of $H(N)$ containing p. However, remember that our main goal is to locate only

[1] We can assume that N contains symbolically defined $2d$ hyperplanes bounding a cube (not parallel to the x_d axis) approaching infinity, and then confine our attention within this cube.

the d-cell of the arrangement $G(N)$ containing p. This can be done as follows. We determine not just the d-cell of $G(N)$ containing p but the full "antenna" [4] of p in $G(N)$. Antenna of the query point p in $G(N)$ is defined as follows. Let p_1 (resp. p_2) be the point of intersection of the vertical line through p and the hyperplane in N immediately above (resp. below) p. Then *antenna* (p) is defined to be the union of the segment $[p_1, p_2]$ together with the recursively defined *antenna*(p_1) and *antenna*(p_2). Note that the algorithm for $d = 2$ in Section 3, with minor modifications, also tells us the antenna of the query point in $G(N)$. Indeed, once we know the trapezoid in the trapezoidal decomposition of $G(N)$ containing p, we also know the lines in N above and below p. If we additionally maintain, for each line $Q \in N$, the ordered list of all intersections on it in the form of a balanced binary tree, we can easily determine the full antenna of p in $G(N)$.

Let us get back to the general dimension. Our goal is to determine the antenna of the query point p in $G(N) = G(N_1)$. Inductively, assume that we have determined the antenna of the query point p in $H(N_i)$. The descent from level i to level $i - 1$ is carried out as follows. Let Q_1 (resp. Q_2) be the hyperplane in N_i immediately above (resp. below) p. Let p_1 (resp. p_2) be the point of intersection of the vertical line through p with Q_1 (resp. Q_2). Then *antenna*(p) in $G(N_i)$ is the union of the segment $[p_1, p_2]$ together with the recursively defined *antenna*(p_1) and *antenna*(p_2). Let $v_1, v_2, \ldots, v_{2^d}$ be the terminal points of this antenna. We assume here that the antenna is bounded. This can be ensured by adding to all levels of our data structure $2d$ hyperplanes bounding a large cube approaching infinity, and restricting everything within this cube.[2]

Lemma 5. *The hyperplane in N_{i-1} that is immediately above (below) p is either Q_1 (Q_2), or it must intersect some d-simplex in $H(N_i)$ adjacent to some v_i, $1 \le i \le 2^d$.*

Proof. We proceed by induction on d. For the sake of induction we prove a somewhat stronger statement. We prove that any hyperplane $Q \in N_{i-1} \setminus N_i$ that intersects *antenna*(p) must intersect a d-simplex in $H(N_i)$ adjacent to some v_i, $1 \le i \le 2^d$.

If Q intersects either *antenna*(p_1) or *antenna*(p_2), then by the inductive assumption it must intersect some $(d-1)$-simplex Δ in the restriction of $H(N_i)$ to Q_1 or Q_2, and hence it intersects the two d-simplices in $H(N_i)$ adjacent to Δ. Otherwise, Q intersects $[p_1, p_2]$, but neither *antenna*(p_1) nor *antenna*(p_2). This means *antenna*(p_1) (resp. *antenna*(p_2)) lies completely above Q (resp. below Q). Let f be the d-cell in $G(N_i)$ containing p and let v be the x_d-minimum on f. Assume, without loss of generality, that v lies below Q, the other case being symmetric. Then, because *antenna*(p_1) lies completely above Q, it is clear that Q must intersect all d-simplices in $H(N_i)$ adjacent to the terminals of *antenna*(p_1). □

By Lemma 5, if we simply check through the conflict lists of all $O(1)$ d-simplices adjacent to the terminals of *antenna*(p), we can easily determine the hyperplanes Q_1' and Q_2' in N_{i-1} that are immediately above or below p. As conflict lists have

[2] The hyperplanes bounding this cube are to be defined symbolically.

$\tilde{O}(\log n)$ size, this takes $\tilde{O}(\log n)$ time. If we appropriately use the recursively defined dynamic point-location structures $\tilde{H}(N_{i-1}, Q_1')$ and $\tilde{H}(N_{i-1}, Q_2')$, we can determine the whole antenna of p in $H(N_{i-1})$ in polylogarithmic time, with high probability.

From the fact that our data structure has $\tilde{O}(\log n)$ levels, it easily follows by induction on d, that the query time for a fixed query point is $\tilde{O}(\log^{d-1} n)$. As the number of distinct search paths in the data structure is easily seen to be polynomial in n, it follows that this bound holds for any query point. To summarize:

Theorem 4. *Let $G(N)$ be an arrangement of n hyperplanes in R^d. There exists a dynamic point-location structure of $O(n^d)$ size allowing $\tilde{O}(\log^{d-1} n)$ query time which also allows insertion/deletion of hyperplanes in $\tilde{O}(n^{d-1} \log n)$ time. The expected cost of update is $O(n^{d-1})$.*

5. Top-Down Dynamic Sampling

It is also of theoretical interest if the query time can be brought down to $\tilde{O}(\log n)$ in arbitrary dimension. In this section we give an alternate dynamic point-location algorithm with $\tilde{O}(\log n)$ query time in arbitrary dimension. The expected space requirement of our algorithm is $O(n^d)$, and the expected amortized cost of update is $O(n^{d-1})$. We do lose something as far as the cost of update is concerned, because our bound holds only in the expected sense (where expectation is solely with respect to randomization in the data structure), whereas for the algorithm given in the last section we could prove a high probability bound for the cost of update.

The organization of this new algorithm is quite different. Roughly speaking, the data structure in Section 4 is defined in a bottom-up fashion, whereas the data structure in the present section is defined in a top-down fashion, very much like the previous static data structures based on random sampling [7], [15].

Given any set N of hyperplanes, our goal is to maintain a dynamic point-location structure so that, given a query point $p \in R^d$, we can quickly, i.e., in logarithmic time, locate the cell of the arrangement $G(N)$ containing p. It turns out to be convenient to solve a slightly more general problem, where we assume that we are given in addition a fixed d-simplex Γ in R^d, and the goal is to locate the cell in the intersection $\Gamma \cap G(N)$ containing the query point p. First we give an algorithm which guarantees $\tilde{O}(\log n)$ query time and $O(n^{d-1} polylog(n))$ expected (amortized) cost of update. We later remark how the expected cost of update can be brought down to $O(n^{d-1})$.

As usual, we first specify our data structure in a static setting and turn to its dynamization later. Hence, let N be a fixed set of n hyperplanes in R^d. Let Γ as before be a fixed d-simplex in R^d. We describe the data structure in a top-down recursive fashion.

At the root level of the data structure, we associate with Γ the entire restricted arrangement $G(N) \cap \Gamma$. This takes $O(|N|^d)$ time and space [13]. We also associate with Γ a coin with bias (probability of success of obtaining a head) $p = 1/n^{1-\delta}$. More precisely, we associate with Γ a *bias index* $i(\Gamma) = \lfloor \log_2 n^{1-\delta} \rfloor$. We then independently toss, for each hyperplane in N, a fair coin $i(\Gamma)$ times in a row. Let

R be the set of hyperplanes for which all $i(\Gamma)$ tosses were heads. The size of R, denoted by r, is roughly n^{δ}. By the Chernoff bound it is easy to see that $r = \tilde{O}(n^{\delta} \log n)$. Let $H(R)$ denote the top-bottom triangulation, as defined in Section 4, of the restricted arrangement $G(R) \cap \Gamma$. We construct $H(R)$ and also a *static* point-location structure for $H(R)$. This, by a conservative estimate, takes $O(r^{O(1)})$ time and space. If δ is small enough, this bound is $O(n^{d})$. For each d-simplex $\Delta \in H(R)$, let $N(\Delta)$ denote the set of hyperplanes in N intersecting (conflicting with) Δ. If $|N(\Delta)| \leq a \log n$, for a suitable *terminating constant* a to be chosen later, we build a simple point-location structure, guaranteeing $O(|N(\Delta)| + 1) = O(a \log n)$ query time, for the top-bottom triangulation $H(N(\Delta))$ of $G(N(\Delta)) \cap \Delta$. This is done by using the following lemma.

Lemma 6. *There exists a trivial static point-location structure for $G(N(\Delta))$ of size proportional to the size of $H(N(\Delta))$ with $O(|N(\Delta)| + 1)$ query time.*

Proof. The idea is to answer the point-location query in the following fashion (this point-location idea has also been used in [1] and [9]). Let $p \in \Delta$ be a query point. We find out in $O(|N(\Delta)| + 1)$ time the first hyperplane Q in $N(\Delta)$ that is hit by the vertical ray emanating from p; the case when this ray hits the boundary of Δ before any hyperplane in $N(\Delta)$ is handled with slight modifications. After this we recursively proceed in the lower-dimensional arrangement $Q \cap (N(\Delta))$, until we eventually locate a vertex of the d-cell R in $\Delta \cap G(N(\Delta))$ containing p. Our next goal is to locate the d-simplex in the x_d-triangulation (top-bottom triangulation) of R containing p. Let v be the bottom of R, i.e., the vertex with the smallest x_d-coordinate (which could possibly lie at infinity). Consider the ray emanating from v and passing through p. We can determine the facet of R hit by this ray in $O(|N(\Delta)| + 1)$ time, since R has only $O(N(\Delta) + 1)$ facets. Let p' be the point of intersection of the ray and the facet. Recursively we determine the $(d-1)$-simplex in the top-bottom triangulation of this facet containing p'. The d-simplex containing p is the cone over this $(d-1)$-simplex with apex at v.

It is clear that the cost of this whole procedure is $O(|N(\Delta)| + 1)$. □

If $|N(\Delta)| > a \log n$, we recur within Δ with respect to the set of hyperplanes $N(\Delta)$. We also associate with every d-cell of the restricted arrangement $G(N(\Delta)) \cap \Delta$, stored at Δ, a *parent pointer* to the d-cell of $G(N) \cap \Gamma$ stored at Γ. Point location is carried out in the obvious manner. To locate a point $p \in \Gamma$ in $G(N) \cap \Gamma$, we first locate the d-simplex of $H(R)$ containing p in $O(\log r)$ time using the point-location structure associated with $H(R)$. Then we recursively locate the d-cell of $G(N(\Delta)) \cap \Delta$ containing p. The parent pointer associated with this d-cell tells us the d-cell of $G(N) \cap \Gamma$ containing p. It is easy to see that the cost of point location is $\tilde{O}(\log n)$. (The cost of point location satisfies the following recursive equation: $Q(r) = r$, for $r \leq a \log n$, and $Q(n) = \log n + Q(r)$, where $r = \tilde{O}(n^{1-\delta}) \log n$.) Moreover, the number of distinct search paths in our data structure is easily seen to be polynomial in n with high probability. Hence, it follows that the cost of locating any point (not just a fixed one) is $\tilde{O}(\log n)$.

Let us denote our point-location structure above by *Sample*(N, Γ). We also let $N(\Gamma) = N$ by convention.

Lemma 7. *The depth of* *Sample*(N, Γ) *is* $\tilde{O}(\log \log n)$ *(assuming that the termination constant a can be chosen large enough).*

Proof. The depth satisfies the following recursive equation: $d(m) = 1$, for $m \leq a \log n$, and $d(m) = 1 + d(r)$, where $r = \tilde{O}(m^{1-\delta} \log n)$. □

Lemma 8. *Sample*(N, Γ) *can be built in* $O(n^d \text{polylog}(n))$ *expected time and space.*

This follows from Lemma 7 and the following slightly more general lemma. For a fixed integer $s > 0$, let $T_s(|N(\Gamma)|)$ denote the *clipped* cost of building *Sample*(N, Γ), ignoring the cost incurred at a depth higher than s, i.e., to say we only take into account the cost incurred up to depth s in the recursive definition of *Sample*(N, Γ). Then

Lemma 9. $E[T_s(N(\Gamma))]$, *the expected value of* $T_s(N(\Gamma))$, *is* $\leq n^d b^s$ *for some constant* $b > 0$ *that depends only on the dimension d.*

Proof. We use induction on s. The total clipped cost $T_s(|N(\Gamma)|)$ satisfies the following probabilistic recurrence equation:

$$T_s(N(\Gamma)) = O(|N(\Gamma)|^d + \sum_{\Delta \in H(R)} T_{s-1}(N(\Delta))) \qquad \text{and} \qquad T(N(\Delta)) = O(|N(\Delta)|^d)$$

$$\text{for} \quad |N(\Delta)| \leq a \log n.$$

The second equality follows from the fact that, for $|N(\Delta)| \leq a \log n$, the storage is proportional to the size of $H(N(\Delta))$ (Lemma 6). By induction hypothesis it follows that

$$E[T_s(N(\Gamma))] = O\left(|N(\Gamma)|^d + b^{s-1} E\left[\sum_{\Delta \in H(R)} |N(\Delta)|^d \right] \right).$$

By [8]

$$E\left[\sum_{\Delta \in H(R)} |N(\Delta)|^d \right] = O(|N(\Gamma)|^d).$$

If we choose b large enough, we are done. □

Now let us turn to the dynamization of our technique. Our procedures for updates are such that, at any time, our data structure is as if it were constructed by applying the above static procedure to the currently active set N of hyperplanes, but with one crucial difference. In the static definition of *Sample*(N, Γ), the bias

integer $i(\Delta)$ associated with a node, labeled with a d-simplex Δ in the data structure, was chosen to be equal to $\lfloor \log_2 n(\Delta)^{1-\delta} \rfloor$, where $n(\Delta) = |N(\Delta)|$. In a dynamic setting we work with a relaxed invariant. We only ensure that

$$|i(\Delta) - \lfloor \log_2 n(\Delta)^{1-\delta} \rfloor| \leq c,$$

where $c \geq 1$ is some predetermined constant; $c = 1$ will do.

Now let us see how to add a new hyperplane h to *Sample*(N, Γ). We give the algorithm in a recursive form. It is initially called with $\Delta = \Gamma$, where Γ denotes the d-simplex associated with the root of our data structure; generally $\Gamma = R^d$.

Procedure Add (*Sample*$(N(\Delta), \Delta)$, h):

1. Add h to $N(\Delta)$ and also to the arrangement $G(N(\Delta)) \cap \Delta$ stored at Δ—by [13], this takes $O(|N(\Delta)|^{d-1})$ time.
2. If $N(\Delta) < a \log n$, where $n = N(\Gamma)$, also update the trivial point-location structure associated with $G(N(\Delta)) \cap \Delta$ that allows $O(|N(\Delta)|)$ query time. Return.
3. If $|i(\Delta) - \log_2(|N(\Delta)|^{1-\delta})| > c$, where $N(\Delta)$ denotes the new set of hyperplanes associated with Δ, we construct new *Sample*$(N(\Delta), \Delta)$ from scratch, applying the static procedure in Lemma 8.[3] Return.
4. Toss a fair coin $i(\Delta)$ times in a row.
5. If not all tosses are heads, for all $\Delta' \in H(R(\Delta))$ intersecting h, call Add (*Sample*$(N(\Delta'), \Delta')$, h). Update the parent pointers. Return.
6. Otherwise, add h to $R(\Delta)$. Update $H(R(\Delta))$ and rebuild from scratch a static point-location structure for the new triangulation $H(R(\Delta))$—if the parameter δ is chosen small enough, the expected cost of this rebuilding, using a conservative estimate, is $O(|R(\Delta)|^{O(1)}) = O(|N(\Delta)|^{d-1})$.
7. Construct conflict lists of the newly created simplices in $H(R(\Delta))$ from the conflict lists of the destroyed d-simplices—this can be easily done in time that is linear in the total structural and conflict change in $H(R(\Delta))$.
8. For each newly created d-simplex Δ' in $H(R(\Delta))$, build *Sample*$(N(\Delta'), \Delta')$ from scratch, using the static procedure in Lemma 8. Also associate a parent pointer with each d-cell of $\Delta' \cap G(N(\Delta'))$.

The deletion operation is very much the reverse of the above addition operation, and hence is not discussed any further.

Theorem 5. *The expected amortized cost of addition or deletion is*

$$O(n^{d-1} polylog(n)).$$

Proof. If *Sample*$(N(\Delta), \Delta)$ is built from scratch in step 3, we amortize the expected cost of this rebuilding equally among the updates to $N(\Delta)$ since the last

[3] The integer n that occurs in the threshold bound $a \log n$ on the number of hyperplanes stored at the leaves of this static data structure is to be taken as $|N(\Gamma)|$, not $|N(\Delta)|$. It is easily seen that the bounds in Lemmas 8 and 9 still apply. We also build *Sample*$(N(\Delta), \Delta)$ from scratch, if the new size of $N(\Delta)$ exceeds the threshold $a \log n$.

such rebuilding took place; it is easy to see that the number of such updates is $\Omega(|N(\Delta)|)$. In what follows we forget about this amortization altogether and pretend the violation of the invariant in step 3 never takes place. (Amortization can be taken care of in a routine fashion.)

We only estimate the expected cost addition, deletion being analogous. Let $T(|N(\Delta)|, h)$ denote the expected cost of adding h to $Sample(N(\Delta), \Delta)$.

Estimating $T(|N(\Delta)|, h)$ directly is difficult. Hence, we use a *clipping* trick, as in the proof of Lemma 8. For every fixed integer let $T_s(|N(\Delta)|, h)$ denote the cost of addition ignoring the cost incurred at depth higher than s. This also means that if some subtree of our data structure at depth, say, s' is built from scratch, we take into account the cost of this static construction up to depth $s - s'$.

We show by induction on s that the expected value of $T_s(|N(\Delta)|, h) \leq c^s|N(\Delta)|^{d-1}$ for some fixed constant c that only depends on the dimension d. We distinguish between two cases.

Case 1: All $i(\Delta)$ tosses of the coin are heads. In this case, examining steps 1, 6, 7, and 8 in the algorithm carefully, we see that

$$T_s(|N(\Delta)|, h) = O(|N(\Delta)|^{d-1}) + \sum_{\Delta'} T_{s-1}(|N(\Delta')|),$$

where Δ' ranges over all newly created d-simplices in the new triangulation $H(R(\Delta))$ and $T_{s-1}(|N(\Delta')|)$ denotes the clipped cost of building $Sample(N(\Delta'), \Delta')$ from scratch. By Lemma 9, $T_{s-1}(|N(\Delta')|) = |N(\Delta')|^d b^{s-1}$ for some fixed $b > 0$. As $R(\Delta)$ is a random sample of $N(\Delta)$, it can be shown, using the results in [8] and the Zone Theorem, that the expected value of $\sum_{\Delta'} |N(\Delta')|^d$ is

$$O\left(\left(\frac{|N(\Delta)|}{|R(\Delta)|}\right)^d \cdot |R(\Delta)|^{d-1}\right) = O\left(\frac{|N(\Delta)|^d}{|R(\Delta)|}\right).$$

Hence, it follows that the expected value of $T_s(|N(\Delta)|, h)$, conditional on all $i(\Delta)$ tosses being heads, is $O(b^{s-1}|N(\Delta)|^d/|R(\Delta)|)$.

Case 2: Otherwise. If we examine steps 1 and 5 in the algorithm we see that, in this case,

$$T_s(|N(\Delta)|, h) = O(|N(\Delta)|^{d-1}) + \sum_{\Delta'} T_{s-1}(|N(\Delta')|, h),$$

where Δ' ranges over all d-simplices in $H(R(\Delta))$ intersecting h. By induction hypothesis, $T_{s-1}(|N(\Delta')|, h) \leq c^{s-1}|N(\Delta')|^{d-1}$. Again using the results in [8] it can be shown that the expected value of $T_s(|N(\Delta), h)$ in this case is $O(c^{s-1}|N(\Delta)|^{d-1})$.

Combining two cases, and noting that the probability of obtaining $i(\Delta)$ heads in a row is (roughly) $|R(\Delta)|/|N(\Delta)|$, it follows that the expected value of $T_s(|N(\Delta)|, h)$ is $O(c^{s-1} + b^{s-1})|N(\Delta)|^{d-1}$. If we choose $c > b$ large enough, it follows that $T_s(|N(\Delta)|, h) \leq c^s|N(\Delta)|^{d-1}$, thereby completing our induction.

By Lemma 7 the depth of our data structure is $O(\log \log n)$ with high probability (by choosing the terminating constant a large enough). It follows that, with high

probability, $T(|N|, h) = T(|N(\Gamma)|, h) = T_s(|N(\Gamma)|, h)$, where $s = O(\log \log n)$. Hence $T(|N|, h)$ is $O(n^{d-1}\, polylog(n))$. □

The point-location algorithm given above can locate only the d-cell of the arrangement $G(N)$ containing the query point p. It is also possible to modify the algorithm so that it can also locate, in addition, the hyperplanes above and below the query point. For this we let $H(N)$ be a *vertical decomposition* [4] of $G(N)$ instead of the x_d-triangulation, as used earlier. The vertical decomposition of $G(N)$ is obtained by passing a vertical wall through every pairwise intersection of the hyperplanes in N. Let l be any fixed pairwise intersection. For any fixed point $p \in l$, consider a vertical segment (parallel to the x_d-axis) that extends upward (and downward) until it hits the first hyperplane in N, and if no such hyperplane exists, it extends to infinity. The union of such vertical segments through all points on l defines a vertical wall through l. When such a wall is raised through all pairwise intersections, we get a decomposition of $G(N)$ into vertical cylinders, whose tops and bottoms are of the same shape (they can possibly touch each other). However, these cylinders can have arbitrarily large number of facets. To get around this phenomenon, we triangulate the top (equivalently bottom) of each cylinder, using the top-bottom triangulation scheme described earlier, and then extend this triangulation vertically to the whole of each cylinder. The resulting decomposition $H(N)$ of $G(N)$ is called its vertical decomposition. Its size is $O(n^d)$ [4].

If we substitute this new "triangulation" $H(N)$ in place of the top-bottom triangulation in the previous point-location structure, we get a new point-location structure. The advantage is that we can also determine in addition the hyperplane in $G(N)$ above (or below) the query point p. This is done as follows. Assume now that Γ associated with the root is a cylinder, e.g., it can be a cube approaching the whole of R^d. Let $Q(\Gamma)$ denote the hyperplane bounding the top of Γ. Let $R = R(\Gamma)$, as before, be the sample of the set $N = N(\Gamma)$ associated with the root of our data structure. Assume that recursively we have determined the hyperplane $Q' \in N(\Delta) \cup \{Q(\Delta)\}$ above the query point, where $\Delta \in H(R(\Gamma))$ is the 3-cell (3-cylinder) containing p, and $Q(\Delta)$ is the hyperplane bounding the top of Δ. By the very nature of a vertical decomposition, Q' is also the hyperplane in $N(\Gamma) \cup \{Q(\Gamma)\}$ above p.

In a similar fashion we can also determine the hyperplane in N below the query point p. If we associate, with each hyperplane $Q \in N$, a recursively defined point-location structure for the lower-dimensional arrangement $G(N) \cap Q$, we can thus determine the whole antenna of p in $G(N)$ in $\tilde{O}(\log n)$ time.

To summarize:

Theorem 6. *Let $G(N)$ be an arrangement of n hyperplanes in R^d. There exists a dynamic point-location structure of $O(n^d\, polylog(n))$ size with $\tilde{O}(\log n)$ query time. In the same time we can also determine the antenna of the query point in $G(N)$. The expected amortized cost of insertion/deletion of a hyperplane is $O(n^{d-1}\, polylog(n))$.*

Remark. There remains one theoretical issue as to whether the polylog factor in the update and the space requirement can be removed. We only indicate how this

is done in a static setting, because the dynamization technique in this section then becomes applicable with minor modifications. The idea is to bootstrap [4] the solution in Theorem 6 (twice), in conjunction with Lemma 5.[4] The translation of the bootstrapping argument of [4] to the present scenario is straightforward. Hence, we do not reproduce it here.

6. Parallel Algorithms for Updates

Since the cost of any update operation (insert or delete) for maintaining an arrangement is quite high, there is sufficient motivation to obtain faster algorithms by using parallelism. We first give a detailed description of a parallel algorithm for update in two dimensions and subsequently sketch its extension to three dimensions. The primary objective is to obtain $O(\log^c n)$ time (for some fixed c) complexity such that the processor-time product matches (within polylogarithmic factors) the sequential update time.

We describe our algorithm in the CRCW PRAM model. In this model we assume that processors can read simultaneously from a memory location and write conflicts are resolved arbitrarily. While describing our algorithm, we often make references to various standard parallel techniques like general and integer sorting, parallel-prefix, and list ranking. Below we state the results that we refer to frequently in our description.

Lemma 10 (General Sort). *N keys can be sorted in $O(\log N)$ time using N CRCW processors.*

Lemma 11 (Integer Sort). *N keys in the range $[0, N \log N]$ can be sorted in $\tilde{O}(\log N)$ time using $N/\log N$ CRCW processors.*

Lemma 12 (Parallel Prefix). *Let a_i, $1 \leq i \leq N$, be N elements from a domain D and let $\circ$ be an associative binary operation defined for elements in the domain. Then the N partial sums $a_1, a_1 \circ a_2, \ldots, a_1 \circ a_2 \circ \cdots \circ a_N$ can be computed in $O(\log N)$ time using $N/\log N$ CRCW processors.*

Lemma 13 (List Ranking). *Given a linked list of N elements, the distance of each element from the head of the list can be computed in $O(\log N)$ time using $N/\log N$) CRCW processors.*

6.1. *Two-Dimensional Updates*

Recall from Section 3 that our point-location structure consists of a hierarchy of levels. With each level we associate the arrangement formed by the lines stored in that level. We also have a descent structure between each pair of successive levels.

[4] Due to Lemma 5, the use of Aronov, Matousek, and Sharir's result in [4] becomes unnecessary.

For this section we adopt the following data-structure for storing an arrangement. For each line h_i, we maintain a sorted sequence of its intersections $v_i(j)$, $1 \leq j \leq n-1$, with all the other lines. In the sorted sequence we also store the (two) faces adjacent to $v_i(j)$, $v_i(j+1)$ with element $v_i(j)$. We store this sequence in a balanced tree (like the $BB(\alpha)$ tree) to facilitate fast searching and updates. We refer to the tree for line h_i as $S(h_i)$. We store each face of the arrangement in a *concatenable* queue data-structure (which is implemented as a balanced binary tree) that supports fast *union* and *split* operations. For face f, the tree $B(f)$ stores the label of the face in the root. All lines (from the previous level) intersecting (conflicting with) a trapezoid Δ are simply maintained in a list $L(\Delta)$. From our previous random sampling lemmas, $|L(\Delta)| = \tilde{O}(\log n)$. Note that for the same face a line can appear in more than one $L(\Delta)$ (i.e., it can intersect more than one trapezoid). It turns out that it is convenient to maintain an additional list $Q(f)$ of lines that intersect a given face f; here there are no multiple occurrences of the same line. This list is also convenient for processor allocation when we describe the details of the update procedure.

For our sequential algorithm it sufficed to bound the total number of update operations; the changes could be made incrementally by following and modifying pointers. However, for our parallel algorithm we need fast access and update abilities which are supported by these data-structures. We confine our discussion to inserting a new line; deletion can be treated in an identical fashion.

Lemma 14. *The operations search, insert, delete, union, and split can be performed in* $O(\log n)$ *time using the* $BB(\alpha)$ *tree.*

The main steps in the update procedure at any fixed level l (in the $O(\log n)$ level hierarchy of random samples) are:

(1) Find out the set of faces $\mathcal{F}$ (zone-faces) that are split by the new line h. Note that, for each $f \in \mathcal{F}$, this creates two faces $\hat{f}_1$ and $\hat{f}_2$. As a consequence new trapezoids are created and some existing ones destroyed.
(2) Reallocate the lines that intersect the affected faces to appropriate new faces.

Step 1 can be implemented in our data-structure by inserting two new vertices (the intersections of the new line h with the face) followed by the split and join operations. From Lemma 14 it can be implemented in $O(\log n)$ time. To find the set of faces $\mathcal{F}$ that are split by the new line we first determine all n intersections and insert them in the corresponding $S(h_i)$ for each line h_i. We also create an $S(h)$ for the new line h. Let us denote the ordered (on line h) set of intersections with line h as x_i where $i \leq n$. Two consecutive intersections x_i and x_{i+1} span a face $f \in \mathcal{F}$ that has been split. The identity of this face can be determined from the labels stored in $S(h_{i_1})$ and $S(h_{i_2})$ such that x_i is $h_{i_1} \cap h$ and x_{i+1} is $h_{i_2} \cap h$. From Lemma 14, face splitting can be done by a single processor in $O(\log n)$ time.

Next we reallocate the lines intersecting the split faces. For each of the new faces created (i.e., $\hat{f}_1$ and $\hat{f}_2$), we also keep track of the leftmost and the rightmost vertices. Note that the leftmost (rightmost) vertex of the new face is either the leftmost (rightmost) vertex of its parent face or one of the new intersections. Also,

we can sort the vertices of each face using $|\hat{f}_i|$ processors in $O(\log n)$ time. (We use $\hat{f}$ to refer to both $\hat{f}_1$ and $\hat{f}_2$.) This allows us to keep track of the sorted sequence of vertices (and hence the trapezoids formed by the consecutive vertical attachments). For each line that intersects a face, we can determine the trapezoids the line intersects once we know where the extreme points (the intersection of the line with this face) lies. If Δ_k and Δ_l are the two bounding trapezoids, then it intersects all the trapezoids in between. Assume that we have determined the new trapezoids a line intersects; we return to this issue after Lemma 15.

If we also know the rank (in the x direction) of the vertical attachments in a face, this immediately gives us the number of trapezoids it intersects. Let δ_i denote the number of trapezoids line h_i intersects. Then we know that $\sum_i \delta_i = \tilde{O}(|\hat{f}| \log n)$. We would like to cluster together all the lines that belong to a trapezoid in a list. We also know that the size of each such list is $\tilde{O}(\log n)$. Assume that there are $|\hat{f}|$ processors available for face $\hat{f}$ (can be done by simple prefix sum since we know the size of each face). So there are $|\hat{f}| \log n$ elements, each of which belongs to a specific trapezoid, and $|\hat{f}|$ processors that have to allocate these elements to the appropriate list.

This is closely related to the following *assignment* problem.

Let U be a set $\{1, 2, \ldots, n\}$ of n indices where each index belongs to exactly one of m groups $G_1, G_2, \ldots, G_m$. Let g_i denote the number of indices belonging to group G_i, $i = 1, \ldots, m$. Given a sequence $N(1), N(2), \ldots, N(m)$ where $\sum_{i=1}^{m} N(i) = O(n)$ and $N(i)$ is an upper bound for g_i, $i = 1, 2, \ldots, m$. The problem is to find a permutation of $(1, 2, \ldots, n)$ in which all the indices belonging to G_1 appear first, all the indices belonging to G_2 appear next, and so on. (Assume that given an index i, the group $G_{i'}$ that i belongs to can be found in $O(1)$ time.)

Therefore we can use the following result [21]. For our case $N(i)$ is $O(\log n)$ and $|U| = |\hat{f}| \log n$.

Lemma 15 (Assignment Lemma). *The above assignment problem can be solved in $\tilde{O}(\log n)$ parallel time using $n/\log n$ PRAM processors.*

We now address the problem of how to determine the trapezoids a line intersects in $O(\log n)$ time using n processors. For this purpose we work with the list $Q(f)$ (recall that it is a list of lines intersecting a face f). Note that each line h_i that has to be reallocated is either split by h or they lie completely "above" or "below" h within the face. In the latter case we already know the intersection points and we can compute their position in the sorted sequence of trapezoids very easily from the updated $B(\hat{f})$. In case the line h_i is split, i.e., its intersection with h lies within the face, we have to determine its position in the sorted sequence of vertical attachments. Since there are at most n intersection points we can do a binary search. Although this is true globally, we may not have enough processors to do it for a given face. We have $|\hat{f}|$ processors and if we are unlucky we may have to do more than $|\hat{f}|$ binary searches for face $\hat{f}$. So for this step we have to reallocate processors globally which can be done with the help of $Q(f)$. We can use a marker bit to indicate whether line h_j intersects h in f. The sum total of marks over all $Q(f)$, $f \in \mathcal{F}$ is n. We accordingly allocate processors and do the binary searches in the

appropriate lists (sorted sequence of the vertices of the face the intersection lies in). Actually we have to do two binary searches—one for each of the split faces $\hat{f}_1$ and $\hat{f}_2$. The $Q(\hat{f}_1)$ and $Q(\hat{f}_2)$ can be constructed easily using the prefix sum.

We next turn to the update of descent structures, which allow us to descend from one level to the next during point location. Recall that the descent structure between level l and $l-1$ is defined as the superposition of the arrangement at level $l-1$ with the trapezoids at level l (Section 3). This amounts to maintaining additional information for the intersections of a vertical attachment (of a trapezoid) with the conflicting lines from the previous level. We maintain for each such intersection point in level l its left and right neighboring vertex in G_{l-1} along the line.

To update this information efficiently we further maintain a sorted list of a line's intersection with the vertical attachments within each face (that the line intersects with). In case the face is split, we have to update this sorted list. We denote this list for a line h_i (for some fixed face) by D_i and the new lists as $\hat{D}_i$. The line h_i can intersect several zone-faces (of h) and hence we use $D_i(g)$ to refer to D_i corresponding to face g (a zone-face of h_i). We use $\hat{D}_i$ to refer to both $\hat{D}(f_1)$ and $\hat{D}(f_2)$. To update the list, we allocate $\lfloor |D_i|/\log n \rfloor$ processors. Note that $\sum_i \sum_{f \in \mathscr{F}} D_i(f) = \tilde{O}(n \log n)$, because the first sum is also bounded by the sum of the conflict sizes of all $O(n)$ trapezoids in the zone of h_i (interchange the summation signs). Each processor is given $\log n$ consecutive elements of this list. If $|D_i| < \log n$, then we bunch them in groups such that each group has $O(\log n)$ elements and assign one processor to each group.

Each element is the intersection of a vertical attachment with line h_i. If the line h_i lies between the vertex (from which the vertical attachment originates) and h, then that element is marked with 1, else it is marked with 0. Now a prefix sum computation can be used to discard all the elements marked 0 and compress the list D_i into $\hat{D}_i$. For the groups that were bunched together because they had less than $\log n$ elements, the processor assigned simply executes the above algorithm sequentially.

This gives the sorted list for the upper face of the (two) faces created by h. A similar computation yields the sorted list for the lower face. There is one more detail that has to be taken care of. The line h inserts two new vertical attachments in a face (that is split). While constructing $\hat{D}_i$ for a face, the intersections (if there is one) of line h_i with the vertical attachments have to be inserted; let us denote these points by l and r (if they exist). Moreover, their positions have to be determined in $G(N_{l-1})$.

We sketch the method leaving out the exact implementation details. We do this step globally for all the lists $\hat{D}_i$. For $|\hat{D}_i| \geq \log n$, this can be done by a binary search by allocating a processor. However, the total number of such lists is not known to be bounded by n; so for the shorter lists we have to adopt a different approach. We denote the neighboring vertical attachments of l (r can be treated similarly) by v_1 and v_2. One of these vertices belong to a zone-trapezoid of the newly inserted line, say v_1. If we count the number of intersections on line h_i that lie in the interval $[v_1, l]$ (call this number $P(l)$ of $G(N_{l-1})$ and sum over all such l_i, then the total can be bound by the number of intersections of $G(N_{l-1})$ lying in

the trapezoids of $H(N_l)$ intersected by h. From Lemma 4 this is $\tilde{O}(n \log n)$. We then allocate processors to groups of l such that the sum of $P(l)$ for that group does not exceed $K \log n$ for some fixed K. We use integer sorting on the labels of the $\tilde{O}(n \log n)$ $\hat{D}_i$ elements to do the processor allocation. There are n labels for lines and so we can use the result of Lemma 11. We allocate one processor to each group and let it sequentially locate the position of l walking along line h_i in $G(N_{l-1})$ from v_1 to l. This way we effectively step through all the intersection points and hence the position of all l's in $G(N_{l-1})$ can be determined at the conclusion of this procedure, which takes $\tilde{O}(\log n)$ time.

We have described the algorithm at a fixed level for update. All the levels can be updated simultaneously once we know the result of the coin tosses. If there are l consecutive heads, then the line gets inserted up to level $l + 1$. To make the data structure updates at any level $j < l$, only information for levels j and $j - 1$ is needed. In particular, to update *Descent*$(l, l - 1)$ only information of $G(N_l)$ and $G(N_{l-1})$ is required. Since the sizes of the levels decrease geometrically, the total number of processors required is $O(n)$ and hence we can state the result as follows:

Theorem 7. *The dynamic point-location data structure for an arrangement of lines can be updated in* $\tilde{O}(\log n)$ *time on a CRCW PRAM model using n processors. The query time is* $\tilde{O}(\log n)$ *and the space complexity is* $\tilde{O}(n^2)$.

6.2. *Extension to Higher Dimensions*

It is not clear whether the update procedure for dimensions higher than two can be parallelized easily using the previous approach. However, in dimension three we can obtain a fairly efficient algorithm by basically extending the algorithm for two dimensions. For each plane h_i we maintain the data structure for storing the projected arrangement $G(N, h_i)$. Moreover, for each pair of planes h_i and h_j that intersect in a line L_{ij}, we maintain a sorted sequence S_{ij} of its intersections with the remaining planes. In the update procedure we update these data structures corresponding to each $G(N, h_i)$ and L_{ij}. These procedures are similar to the ones described in the previous section. The cells (convex polyhedra in this case) can be stored in any of the standard data structures, for example, as incidence graphs. For each face of $G(N, h_i)$, we store the labels of the cells that it bounds. For each edge, we store the labels of the four cells that are incident on it.

If a cell is split by a newly inserted plane, it can be quickly identified by the position of the intersection of this plane and L_{ij} in S_{ij}. With the number of processors proportional to the size of the cell, the data structures corresponding to the new cells can be formed. In addition we have to update the conflict lists for each 3-simplex [17] lying in the zone of the new (or deleted) plane. We use the triangulation scheme of Dobkin and Kirkpatrick [12]. The triangulation procedure can be parallelized using the algorithms of [10] or [23]. For each plane (lying in the zone) we have to determine the triangles (more precisely, 3-simplices) it intersects. The search procedure uses the hierarchical representation of convex polyhedra which is obtained from [12]. The search begins from the bottommost level where there is a constant number of triangles. The number of levels k, is

roughly $O(\log |C|)$, where C is the triangulated convex polytope the plane intersects. For each plane–triangle intersection we allocate a processor. In the next level of the hierarchy, each triangle of level k has only a constant number of neighboring triangles from level $k-1$ (see [10]). So in constant time, a processor can check the set of triangles in level $k-1$ the plane intersects. The total number of processors required is proportional to the number of plane–triangle interactions which is $\tilde{O}(n_l^2 \log n)$, where n_l is the number of planes in level l (from the random sampling property). In order to reallocate processors evenly, we spend an extra $O(\log n)$ to do dynamic load balancing by a simple prefix sum computation. Hence we can apply Brent's slow-down lemma to reduce the number of processors by a multiplicative factor of $O(\log n)$ for the intersection detection part without affecting the asymptotic time complexity. The remaining implementation details can be worked out easily to arrive at the following result.

Theorem 8. *The dynamic point-location data structure for an arrangement of n planes can be updated in $\tilde{O}(\log^2 n)$ time on a CRCW PRAM model using n^2 processors. The query time is $\tilde{O}(\log^2 n)$ and the space complexity is $\tilde{O}(n^3)$.*

Note that this is $O(\log n)$ factor away from an optimal speed-up algorithm. We suspect that a more careful reallocation procedure could lead to an optimal speed-up algorithm.

7. Conclusion

In this paper we have presented a very simple scheme for maintaining a dynamic point-location structure for arrangements of hyperplanes that guarantees poly-logarithmic query time. One of the main contributions of this paper has been adaptation of the skip-list methodology [20] for dynamic algorithms. We believe that this is a versatile and powerful tool which is likely to have further applications [16].

References

1. P. Agarwal *et al.*, Euclidean minimum spanning trees and bichromatic closest pairs, *Discrete Comput. Geom.*, **6** (1991), 407–422.
2. C. Aragon and R. Seidel, Randomized search trees, *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, 1989, pp. 540–545.
3. B. Chazelle and J. Friedman, A deterministic view of random sampling and its use in geometry, *Combinatorica*, **10** (1990), 229–249.
4. B. Chazelle and J. Friedman, Point location among hyperplanes, Manuscript.
5. S. Cheng and R. Janardan, New results on dynamic planar point location, *Proceedings of the IEEE Symposium on Foundations of Computer Science*, 1990, pp. 96–105.
6. H. Chernoff, A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations, *Ann. of Math. Statist.*, **23** (1952), 493–507.
7. K. L. Clarkson, New applications of random sampling in computational geometry, *Discrete Comput. Geom.*, **2** (1987), 195–222.
8. K. L. Clarkson and P. Shor, Applications of random sampling in computational geometry, II, *Discrete Comput. Geom.*, **4** (1989), 387–421.

9. M. Dyer and A. Frieze, A randomized algorithm for fixed-dimension linear programming, *Math. Programming*, **44** (1989), 203–213.
10. N. Dadoun and D. Kirkpatrick, Parallel construction of subdivision hierarchies, *J. Comput. System Sci.*, **39** (1989), 153–165.
11. D. Dobkin and D. Kirkpatrick, A linear time algorithm for determining the separation of convex polyhedra, *J. Algorithms*, **6** (1985), 381–392.
12. D. Dobkin and D. Kirkpatrick, Determining the separation of preprocessed polyhedra—a unified approach, *Proceedings of the 17th International Colloquium on Automated Language Programming*, 1990, pp. 400–413.
13. H. Edelsbrunner, R. Seidel, and M. Sharir, On the zone theorem for hyperplane arrangements, Rept. UIUCDCS-R-91-1655, Department of Computer Science, University of Illinois, Urbana, IL, 1991.
14. O. Fries, K. Mehlhorn, and S. Naeher, Dynamization of geometric data structures, *Proceedings of the First ACM Symposium on Computational Geometry*, 1985, pp. 168–176.
15. D. Haussler and E. Welzl, ε-nets and simplex range queries, *Discrete Comput. Geom.*, **2** (1987), 127–152.
16. K. Mulmuley, Randomized multidimensional search trees: dynamic sampling, *Proceedings of the Annual ACM Symposium on Computational Geometry*, June 1991, pp. 121–131.
17. K. Mulmuley and S. Sen, Dynamic point location in arrangements of hyperplanes, *Proceedings of the Annual ACM Symposium on Computational Geometry*, June 1991, pp. 132–141.
18. M. Overmars, *The Design of Dynamic Data Structures*, Lecture Notes in Computer Science, Vol. 156, Springer-Verlag, Berlin, 1983.
19. F. Preparata and R. Tamassia, Fully dynamic point location in a monotone subdivision, *SIAM J. Comput.*, **18** (1989), 811–830.
20. W. Pugh, Skip lists: a probabilistic alternative to balanced trees, *Comm. ACM*, **33**, (1990), 668–676.
21. S. Rajasekaran and J. H. Reif, Optimal and sub-logarithmic time randomized parallel sorting algorithms, *SIAM J. Comput.*, **18** (1989), 594–607.
22. J. H. Reif and S. Sen, Optimal randomized parallel algorithms for computational geometry, *Proceedings of the 16th International Conference on Parallel Processing*, 1987. Revised version in *Algorithmica*, **7** (1992), 91–117.
23. J. Reif and S. Sen, Polling: a new random sampling technique for computational geometry, *Proceedings of the ACM Symposium on the Theory of Computing*, 1989, pp. 394–404.

Received August 25, 1991.

Note added in proof. The data structure in Section 4 can also be used for ray shooting as follows. Let N be the set of hyperplanes as in that section. Given a query point $p \in R^d$, and a ray originating from p, the goal in this problem is to determine quickly the first hyperplane in N that is hit by this ray, if any. Choose a z-coordinate such that the given ray becomes parallel to the z-axis. This coordinate depends on the query. It is completely independent of the x_d-coordinate that was used in the bottom-vertex triangulation. Define *antenna*(p) with respect to the z-coordinate just as in Section 4: Now "above" means in the positive z-direction and "below" means in the negative z-direction. The query procedure in Section 4 correctly determines *antenna*(p) for any choice of the z-coordinate. This is because Lemma 5 holds for any z-coordinate.

Once we know *antenna*(p) with respect to the z-direction we automatically know the first hyperplane hit by the query ray. (If there is no such hyperplane, the corresponding terminals of the antenna lie at infinity.)

Thus we get a dynamic ray-shooting algorithm with the same performance as in Theorem 4.