Optimal In-Place and Cache-Oblivious Algorithms for 3-d Convex Hulls and 2-d Segment Intersection

Timothy M. Chan   Eric Y. Chen*

December 24, 2009

Abstract

We describe the first optimal randomized in-place algorithm for the basic 3-d convex hull problem (and, in particular, for 2-d Voronoi diagrams). The algorithm runs in $O(n \log n)$ expected time using only $O(1)$ extra space; this improves the previous $O(n \log^3 n)$ bound by Brönnimann, Chan, and Chen [SoCG'04]. The same approach leads to an optimal randomized in-place algorithm for the 2-d line segment intersection problem, with $O(n \log n + K)$ expected running time for output size K, improving the previous $O(n \log^2 n + K)$ bound by Vahrenhold [WADS'05]. As a bonus, we also point out a simplification of a known optimal cache-oblivious (non-in-place) algorithm by Kumar and Ramos (2002) for 3-d convex hulls, and observe its applicability to 2-d segment intersection, extending a recent result for red/blue segment intersection by Arge, Mølhave, and Zeh [ESA'08]. Our results are all obtained by standard random sampling techniques, with some interesting twists.

Keywords: In-place algorithms, Convex hulls, Voronoi diagrams, Segment intersection, Cache-oblivious algorithms

1 Introduction

In this paper, we address the basic theoretical question of how much space is needed to solve fundamental problems in computational geometry. In the usual setting, we assume the input is given in an array, where elements in the input array may be permuted; the measure of interest is the amount of working space excluding the input array (and the output stream, if the problem requires one). For example, standard $O(n \log n)$-time algorithms for the 3-d convex hull problem [38, 20] require $O(n)$ space; Chazelle and Edelsbrunner's $O(n \log n + K)$-time algorithm for 2-d segment intersection [16] requires $O(n + K)$ space (K denotes the output size), whereas Balaban's $O(n \log n + K)$-time algorithm [4] requires only $O(n)$ space. Generally, throughout the literature, an underlying goal has always been to lower the space complexity of algorithms whenever possible, if the running time does not have to be sacrificed.

For two very basic geometric problems, this paper provides the ultimate in space reduction—we prove that there is essentially no inherent limit in how far we can reduce space. More precisely, we show that, in the standard algebraic decision tree model, there are asymptotically time-optimal algorithms for 3-d convex hulls and 2-d segment intersection using only $O(1)$ space. The algorithms

*Cheriton School of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada ({tmchan, y28chen}@uwaterloo.ca). Work supported by NSERC.

are randomized. For the former problem, given n points in an array, the algorithm can, in $O(n \log n)$ expected time, permute the array so that the vertices of the convex hull appear in a prefix of the array; if the facets or edges of the convex hull are desired, they can be printed to an output stream. The running time can be further reduced to $O(n \log h)$ if the output has h vertices. For the latter problem, given n line segments in an array, the algorithm can, in $O(n \log n + K)$ expected time, print the K intersecting pairs to the output stream.

Since 2-d Voronoi diagrams reduce to 3-d convex hulls, we can also print the vertices of the Voronoi diagram of n given points in $O(n \log n)$ expected time with $O(1)$ space. A few applications follow; for example, we can compute the closest red/blue pair between n red points and n blue points, or compute the largest circle enclosing n points whose center lies within a given triangle Δ_0, in $O(n \log n)$ expected time with $O(1)$ space.[1] The techniques we use are general enough that they can likely be applied to the construction of other linear-size geometric structures.

Previous work. Algorithms using $O(1)$ extra space in a model where elements may be permuted in the input array are commonly called *in-place* algorithms; heapsort is a classical example. There has been renewed interest in in-place algorithms in the last few years. For example, the problem of in-place sorting in $O(n \log n)$ time with $O(n)$ swaps was resolved only in 2003 by Franceschini and Geffert [25]. See [29, 28] for other recent examples of in-place algorithms, and [27, 26] for examples of in-place, also called *implicit*, data structures where the structure is represented entirely within a permutation of the elements in the input array.

The more sophisticated in-place algorithms and implicit data structures typically build on the following trick (e.g., see [36]): we can encode a bit implicitly without extra space by simply permuting a consecutive pair (p, q) of elements in the input array, for example, by letting the encoded bit be 0 (resp. 1) if p is lexicographically smaller (resp. larger) than q. Using this trick, a pointer can be encoded by permuting $O(\log n)$ pairs.

In computational geometry, a series of papers has recently appeared on in-place algorithms. Some simple results were described by Brönnimann *et al.* [11] (on 2-d convex hulls) and Bose *et al.* [7]. Chen and Chan [18] obtained a segment intersection algorithm running in $O((n+K)\log^2 n)$ time and $O(\log^2 n)$ space, by reworking Bentley and Ottmann's sweep-line algorithm using known implicit data structures (for the search tree and priority queue). Brönnimann and Chan [9] obtained an $O(n)$-time in-place algorithm for computing the convex hull of a 2-d polygonal chain, by reworking Melkman's algorithm using implicit data structures (for the deque). The most substantial previous paper on the topic was by Brönnimann, Chan, and Chen [10], who gave a collection of nontrivial results on both in-place geometric algorithms and implicit geometric data structures. Of particular relevance is the main result on 3-d convex hulls: the paper presented the first efficient in-place algorithm, running in $O(n \log^3 n)$ time, which is obtained by first designing a new sweep-line $O(n \log^2 n)$-time algorithm and then making it in-place using the implicit pointer encoding trick mentioned above.

Vahrenhold [42] later improved Chan and Chen's result on segment intersection and obtained an $O(n \log^2 n + K)$-time in-place algorithm, this time, by a careful reworking of Balaban's algorithm [4]. Turning to other problems, Blunck and Vahrenhold [5] gave an optimal $O(n \log n)$-time in-place

[1] For the former problem, observe that the optimal pair must form a Delaunay edge; we can examine the Delaunay edges in $O(1)$ space as they are generated one at a time. For the second problem, observe that the optimal center must be a vertex v in the *clipped* Voronoi diagram, i.e., the intersection of the Voronoi diagram with Δ_0; the radius of the largest empty circle centered at v can be determined from the features defining the vertex (e.g., 3 points, or 2 points and an edge of Δ_0). It is known that the 2-d clipped Voronoi diagram can be transformed into a 3-d convex hull [15].

algorithm for 2-d layers of maxima and 3-d maxima. Chan and Chen [13] presented a new implicit data structure for 2-d nearest neighbor search with $O(\log^{1.71} n)$ query time.

Despite all these previous efforts, optimal in-place algorithms for the basic 3-d convex hull and 2-d segment intersection problems have remained elusive. Our results provide the last word on these open questions.

Our techniques. Any sufficiently involved pointer-based structure we want to maintain would require the above-mentioned bit encoding trick. However, the time to encode/decode a single pointer is already logarithmic; this is one of the reasons why achieving optimal running time is often difficult in the in-place setting.

Our overall idea is to use a divide-and-conquer strategy where the subproblem sizes drop dramatically. This way, pointer sizes ($\log n$) would also decrease sufficiently, so that the extra logarithmic cost would dissipate at the end. We will aim roughly for a recurrence of the form $T(n) \leq (cn/\log n)\,T(\log n) + O(n \log n)$, which would indeed yield $T(n) = O(n \log n)$. (In fact, once we get such a recurrence, the subproblems are small enough that they can be solved directly without recursion.) To accomplish the divide-and-conquer, we apply standard random sampling techniques. The division into subproblems requires point location queries, and the key lies in a method (Lemma 2.4) to answer a batch of n point location queries, in a subdivision of sublinear size, in $O(n \log n)$ time without the extra logarithmic penalty.

Of course, countless papers in computational geometry since Clarkson and Shor [20] have used randomized incremental construction and the random sampling paradigm [35] before. In particular, several works have used sampling to link the complexity of problems like 3-d convex hulls to that of offline point location in different (e.g., transdichotomous or cache-oblivious) models [14, 33]. However, our work is not just another routine application of random sampling. Several sampling steps with different sample sizes are needed to put together the final result, though the algorithm can still be described concisely. In some ways, our algorithm is perhaps more related to (or generalizes) Franceschini and Geffert's usage of deterministic samples in their in-place 1-d sorting algorithm [25]. In any case, we can now add one more reason to study randomized techniques in computational geometry—that they can not only lead to time-optimal algorithms, but to space-optimal ones as well.

On cache-oblivious algorithms. Although we start off with theoretical questions, the ideas in our solution have led us to some new observations on a topic of perhaps more practical significance: the design of cache-oblivious algorithms. We will not survey the large body of work in this area; see [8, 22, 32], for instance. Let N, M, and B denote the input size, memory size, and block size respectively (in the cache-oblivious model, the algorithm does not know M and B). We assume that $M > B^{1+\gamma}$ for some constant $\gamma > 0$ (the "tall cache assumption"). For 3-d convex hulls, Kumar and Ramos [33][2] have already given an optimal randomized cache-oblivious (non-in-place) algorithm using $O((N/B)\log_M N)$ memory transfers. For 2-d segment intersection, Arge, Mølhave, and Zeh [3] recently presented an optimal cache-oblivious algorithm using $O((N/B)\log_M N + (K/B))$ memory transfers but only for the special red/blue case where the input consists of two collections of disjoint segments. Apparently, an optimal cache-oblivious algorithm for general segment intersection has been left open.

[2] Their algorithm is described for 2-d Voronoi diagrams, but should also work for 3-d convex hulls.

We observe that Kumar and Ramos' convex hull algorithm can be simplified (specifically, the extra "pruning steps" turn out to be unnecessary after all). Secondly, we observe that their algorithm can be adapted to yield an optimal cache-oblivious algorithm for general segment intersection.

The overall approach is again a sampling-based divide-and-conquer strategy. The new idea is quite simple: this time, we will aim for a recurrence of the form $T(n) \leq cn^{1-\varepsilon}T(n^{\varepsilon}) + O(n \log n)$ for an absolute constant c and a sufficiently small constant $\varepsilon > 0$. If ε is small enough (as compared to $1/c$), the recurrence would indeed have solution $T(n) = O(n \log n)$. Somehow, this simple approach was overlooked by previous researchers. This type of recurrence adapts well to the cache-oblivious settings. The approach suggests that a variant of the standard randomized incremental construction or random sampling method may be advantageous, where we stop the process once we have reached a sample size of $n^{1-\varepsilon}$, reorganize the array, and then recurse. (See [2, 1, 21] for other previous work on practical and I/O-efficient variants of random sampling methods.)

2 In-Place 3-d Convex Hulls

2.1 Preliminaries

In all our algorithms, we assume that the input is in general position.

We describe our 3-d convex hull algorithm in dual space. In the equivalent dual problem, we are given a set H of n planes in 3-d and want to construct its lower envelope $\mathcal{E}(H)$. Slightly generalizing the problem, we want the construct the portion of $\mathcal{E}(H)$ inside a fixed tetrahedron Δ_0 with one vertex at $(0, 0, -\infty)$.

Randomized divide-and-conquer. Given a subset R, let $\mathcal{T}(R)$ denote the *canonical triangulation* [19] of the region underneath $\mathcal{E}(R)$, defined as follows. For each face f of $\mathcal{E}(R) \cap \Delta_0$, we triangulate the convex polygon f by connecting the lowest vertex of f to all other vertices; for each resulting triangle f', we take the region underneath f' (a tetrahedron with one vertex at $(0, 0, -\infty)$) to be a cell of $\mathcal{T}(R)$. The total number of cells is $O(|R|)$. For each cell $\Delta \in \mathcal{T}(R)$, let H_Δ denote the subset of all planes in $H \setminus R$ intersecting Δ (the *conflict list* of Δ). To compute $\mathcal{E}(H)$, we can compute $\mathcal{E}(H_\Delta)$ inside each cell $\Delta \in \mathcal{T}(R)$. The following standard sampling lemma by Clarkson and Shor [20] is our starting point:

Lemma 2.1 *Given a set H of n planes, take a random sample R of size r. Then*

(i) $E\left[\sum_{\Delta\in\mathcal{T}(R)} |H_\Delta|^b\right] \;=\; O(r \cdot (n/r)^b)$ *for any constant $b \geq 0$;*

(ii) $\Pr\left\{\max_{\Delta\in\mathcal{T}(R)} |H_\Delta| > (n/r)\log t\right\} \;=\; O(r/t)$ *for any $t \geq r$.*

In particular, the following holds with probability at least $1/2$ for some absolute constant c:

$$\sum_{\Delta\in\mathcal{T}(R)} |H_\Delta| \;\leq\; cn \qquad \text{and} \qquad \max_{\Delta\in\mathcal{T}(R)} |H_\Delta| \;\leq\; c(n/r)\log r. \tag{1}$$

The permutation+bits model. We will work with an intermediate *permutation+bits model*, which is sufficient to capture the difficulty in the design of in-place algorithms, but allows algorithms to be expressed much more cleanly. Some form of this model has been used before in our previous paper [13] for implicit geometric data structures, and also in other previous in-place algorithms such as [25]. As usual, the input array stores a permutation of the input elements at any time, and we have $O(1)$ registers that can store indices and input elements. Standard operations on array elements and registers cost unit time as usual. In the new model, we are also allowed to work with an extra array of bits which we call the "buffer area". The number of bits may be large (possibly larger than linear), but there is a price to pay: the content of this extra space can only be accessed through bit probes, with each bit access costing unit time.

The buffer area can store pointers to input elements (with $O(\log n)$ bits) but cannot store input elements themselves (which we can think of as having infinite precision). For example, if the buffer area stores an array of pointers to the input elements, a binary search that normally takes $O(\log n)$ time would now costs $O(\log^2 n)$ in our model because reading a single pointer stored in the buffer area already costs $O(\log n)$.

Output requirement. The algorithm should identify (i) the vertices of the lower envelope (i.e., the hull facets in primal space), and (ii) the planes that participate in the lower envelope (i.e., the hull vertices in primal space). For (i), the vertices of the envelope are to be printed to an output stream, where each vertex is represented by its three defining planes. Note that if the edges of the envelope (i.e., hull edges in primal space) are desired, they can be printed as well, where each edge is represented by its two defining planes: For each vertex v printed, we just print the (at most two) edges formed by the three defining planes that are oriented rightward towards v. This way, each edge is printed once. The edges that intersect $x = \infty$ can be generated by a 2-d lower envelope algorithm [11].

For (ii), we assume that the buffer area stores an array of n "output bits", one corresponding to each input plane. At the end, we require that the algorithm turns the output bits to 1 for those input planes that participate in the envelope. Output bits that were set to 1 before we call the algorithm will remain set to 1. At the very beginning, we can initialize the output bits to 0.

In the model, we are allowed to permute elements in the input array during the execution of the algorithm. Though not explicitly stated in our algorithm descriptions below, whenever we swap elements in the input array, we will swap the corresponding output bits; this will not increase the asymptotic cost.

Reduction to the permutation+bits model. We now show that any efficient algorithm $\mathcal{A}$ in the permutation+bits model, using $O(n \operatorname{polylog} n)$ bits of space, can be converted to an efficient in-place algorithm for our problem. We apply the basic bit encoding trick mentioned in the introduction. The reduction is accomplished by two lemmas, both using random sampling with small sample sizes:

Lemma 2.2 *In the permutation+bits model, given a 3-d convex hull algorithm $\mathcal{A}$ requiring $T(n)$ expected time and $S(n)$ bits of space, we can obtain a 3-d convex hull algorithm $\mathcal{A}'$ requiring $O(T(n) + n \log n)$ expected time and $O(S(n \log\log n / \log n) + n)$ bits of extra space, assuming that $T(n)/n$ is increasing.*

Proof: Draw a random sample R of size $r = \log n$ and put it in a suffix of the input array. We compute $\mathcal{E}(R)$ and $\mathcal{T}(R)$ naively, in $r^{O(1)}$ time, and store it in the buffer area.

Take a cell $\Delta \in \mathcal{T}(R)$. In $O(n)$ time, we can determine which planes are in H_Δ, and move the planes of H_Δ to the front of the input array using a standard in-place partitioning subroutine. We run algorithm $\mathcal{A}$ on H_Δ to compute the part of the lower envelope contained in Δ. We then proceed to the next cell Δ. (The planes in $H \setminus R$ have been permuted, but this is allowed.)

Assuming (1), the running time is $\sum_{\Delta\in\mathcal{T}(R)} T(|H_\Delta|) + O(nr) \le O(T(n) + n\log n)$ and the space for the bits is $\max_\Delta S(|H_\Delta|) + O(n) = S((n/\log n)\log\log n) + O(n)$. If (1) fails, we can retry with a new sample; the expected number of trials is $O(1)$. $\square$

Lemma 2.3 *Given a 3-d convex hull algorithm $\mathcal{A}$ requiring $T(n)$ expected time and $O(n)$ bits of extra space in the permutation+bits model, we can obtain an in-place 3-d convex hull algorithm $\mathcal{A}'$ requiring $O(T(n))$ expected time.*

Proof: Draw a random sample R of size r where, this time, r is a constant made sufficiently large. We compute $\mathcal{E}(R)$ and $\mathcal{T}(R)$ naively, in $O(1)$ time and space. Before each iteration, we maintain the invariant that the input array is divided into two parts σ_1 and σ_0, where σ_1 holds the list of planes whose current output bit is 1 (i.e., planes that have been found to participate in the lower envelope) and σ_0 holds the list of remaining planes. Initially, σ_1 is empty.

Take a cell $\Delta \in \mathcal{T}(R)$. In $O(n)$ time, we can determine which planes are in H_Δ, and move the planes of $H_\Delta \cap \sigma_1$ and $H_\Delta \cap \sigma_0$ to the front of the input array, by in-place partitioning. We simulate algorithm $\mathcal{A}$ on H_Δ for the cell Δ. By (1), we can ensure that $|H_\Delta| \ll \varepsilon n$ and $\mathcal{A}$ uses εn bits of space for a some suitable constant $\varepsilon > 0$ (if (1) fails, we restart). Using the bit-encoding trick, we can simulate the buffer area by permuting $2\varepsilon n$ pairs at the back of the input array (the only one exception is the pair that crosses between σ_1 and σ_0). Notice that by choosing ε small enough, we can ensure that there is no overlap between the part of the array storing H_Δ and the buffer area. Before the simulation, we record the current output bits of H_Δ in the buffer area. After the simulation, we unsplit H_Δ into $H_\Delta \cap \sigma_1$ and $H_\Delta \cap \sigma_0$ based on the updated output bits from the buffer area, and then move blocks around, in $O(n)$ time (in-place), to restore the invariant that the input array is divided into the updated σ_1 and σ_0. We then proceed to the next cell Δ.

Since r is a constant, the total expected running time is $O(T(n))$. $\square$

It is not difficult to obtain a 3-d convex hull algorithm that runs in $O(n \operatorname{polylog} n)$ time using $O(n \operatorname{polylog} n)$ bits of space in the permutation+bits model: We can just simulate any known optimal $O(n\log n)$-time, $O(n)$-space algorithm, such as Preparata and Hong's divide-and-conquer algorithm [38]. (We may assume that this algorithm does not modify the input array, as we can work with pointers to the input points.) Each pointer operation now costs $O(\log n)$, so the running time becomes $O(n\log^2 n)$, and the number of bits of extra space is $O(n\log n)$.

Applying Lemma 2.2 to this algorithm with $T(n) = O(n\log^2 n)$ and $S(n) = O(n\log n)$ gives a new algorithm in the permutation+bits model with $O(n\log^2 n)$ running time and $O(n\log\log n)$ bits of extra space. Applying Lemma 2.2 once more gives $O(n\log^2 n)$ running time and $O(n\log^2\log n/\log n + n) = O(n)$ bits of extra space. Applying Lemma 2.3 finally yields an in-place 3-d convex hull algorithm with $O(n\log^2 n)$ expected time—this is already an improvement over Brönnimann, Chan, and Chen's $O(n\log^3 n)$ algorithm!

To obtain an $O(n\log n)$-time in-place algorithm, we can focus all our attention on designing an algorithm in the permutation+bits model that requires $O(n\log n)$ expected time and $O(n\log n)$ bits of extra space. Unfortunately, tasks that normally take $O(n)$ time, like merging two vertically separated convex hulls [38], no longer seem doable in $O(n)$ time in the permutation+bits model. Ironically, the difficulty in designing an optimal in-place algorithm lies in time complexity, not space.

2.2 An $O(n \log n)$ Algorithm

If we cannot lower the $O(n \log n)$ cost of simple tasks in the permutation+bits model, we will try to accomplish more in $O(n \log n)$ time—instead of dividing into 2 or a constant number of subproblems, we divide into a much larger number of subproblems at once, using random sampling. The computation of the conflict lists now requires an efficient algorithm to answer a batch of point location queries in a subdivision of sublinear size. Unfortunately, no implicit data structure for point location (or even 2-d nearest neighbor queries) attaining $O(\log r)$ query time is known [13, 6]. In our situation, however, the queries are given offline, and the array can be permuted during the algorithm. The following key lemma shows that in such a situation, the queries can be answered effectively with $O(\log r)$ cost each.

Lemma 2.4 *Suppose we are given a set P of n points in 2-d, stored consecutively in the input array, another set R of r points, stored consecutively in the input array, and a planar subdivision over the vertices in R of size $O(r)$, encoded in the buffer area.*

We can locate the cell γ_p in the subdivision containing each point $p \in P$, in $O(n \log r + r \log r \log n)$ time in total and using $O(r \log r \log n)$ bits of extra space in the permutation+bits model. The algorithm is allowed to permute P in the input array, and the γ_p's are to be stored as an array of $O(n \log r)$ bits in the buffer area (in the order corresponding to the permutation of P at the end).

Proof: We simulate a known planar point location method in the permutations+bits model. The most convenient for our purposes is Preparata's *trapezoid method* [37, 39]. The preprocessing time is $O(r \log r)$ time normally, but now costs $O(r \log^2 r)$ in our model. (We may assume that the preprocessing algorithm does not modify the array storing elements of R, as we can work with pointers to these elements.) The data structure is just a binary tree structure, with $O(r \log r)$ nodes and $O(\log r)$ height, where each node stores (pointers to) a constant number of elements of R. The query algorithm just traces down a path of this tree, where at each node we determine which child to descend to by performing a comparison involving the elements at the node.

If we naively simulate the query algorithm, each query would require $O(\log r)$ pointer operations and cost $O(\log^2 r)$. Our idea is to handle all the queries collectively as a batch. At the root, the query points consist of an array of n elements of P. We copy (the pointers to) the elements of R stored at the root node to the working registers, with $O(1)$ pointer operations, costing $O(\log n)$ time. We then perform n comparisons with the elements in the array and split the array into two subarrays where elements with the same outcome are put in the same subarray; this can be done in $O(n)$ time by the standard in-place partitioning subroutine. We then recurse on both children of the tree, with their associated subarrays. (We can simulate recursion using a stack, where each record stores two indices representing the boundary of a subarray, and a pointer to a node of the tree structure; the number of stack operations required is $O(r)$.) The total cost of the calls to the partitioning subroutine is $O(n)$ per level, i.e., $O(n \log r)$. The total cost of the pointer operations (and stack operations) is $O(r \log r \log n)$.

At the end, we can write the answers of the elements to the buffer area with a linear scan, where elements in the same subarray at the leaves have the same answers. (Note that the input array for P has been considerably rearranged, but this is allowed. In contrast, the array for R is not modified.) $\square$

The preceding algorithm can be viewed as a variant of quicksort, because of the recursive use of the partitioning subroutine. We remark that not all standard point location methods can be simulated as nicely as above; known optimal search structures with linear space (such as [24, 31, 40]) are DAGs, not binary trees (see [41] for an explanation). For the above, we only need optimal $O(\log r)$ query time, not optimal space, so the trapezoid method is ideal. (Another alternative is an r^{ε}-ary segment tree, which has $O(\log r)$ query time and $O(r^{1+\varepsilon})$ preprocessing time; because of the higher preprocessing time, this option would require more effort later in the recursion.)

With the above lemma on point location, we can now describe our optimal algorithm:

Theorem 2.5 *There is a 3-d convex hull algorithm that runs in $O(n \log n)$ expected time and uses $O(n \log n)$ bits of extra space in the permutation+bits model.*

Proof: Draw a random sample R of size $r = n/\log n$ and put it in a suffix of the input array. We compute $\mathcal{E}(R)$ and $\mathcal{T}(R)$, stored in the buffer area, in $T(r)$ time by some algorithm $\mathcal{A}$ to be specified later.

For each input plane $h \in H \setminus R$, we first compute a vertex of $\mathcal{E}(R)$ above h. It suffices to find the extreme vertex v_h in $\mathcal{E}(R)$ along the direction orthogonal to h. In the dual, $\mathcal{E}(R)$ corresponds to an upper hull of r points; planes tangent to $\mathcal{E}(R)$ corresponds to points on the upper hull; and planes parallel to h corresponds to points on a vertical line. Thus, finding v_h is equivalent to intersecting the upper hull with a query vertical line. This reduces to a point location query in the vertical projection of the upper hull, a planar triangulation with $O(r)$ vertices. By Lemma 2.4, the cost of the n offline point location queries is $O(n \log r + r \log^2 n)$. The input array for $H \setminus R$ has now been permuted (which is fine since so far we have not kept any pointer structure to the elements of $H \setminus R$); we will keep this permutation as the "home" permutation.

For each input plane h, compute the list V_h of all vertices of $\mathcal{E}(R)$ above h. This can be done by a depth-first search in $\mathcal{E}(R)$, starting at v_h; in the traditional model, this takes $O(|V_h|)$ time, since the graph associated with $\mathcal{E}(R)$ has maximum degree 3. From this list, we obtain the list of cells of $\mathcal{T}(R)$ intersected by h. From these lists, we obtain the conflict lists H_Δ for all $\Delta \in \mathcal{E}(R)$, stored in pointer structures in the buffer area. In the traditional model, the total time is proportional to the total size of the H_Δ's, which is $O(n)$ by (1) (if (1) fails, we restart). In the permutation+bits model, the cost becomes $O(n \log n)$ (all these operations are done in the buffer area, without permuting the input array).

Now, take a cell $\Delta \in \mathcal{T}(R)$. We move the planes of H_Δ to the front of the array; this requires $O(|H_\Delta|)$ swaps in the input array and $O(|H_\Delta|)$ pointer operations on the conflict lists, taking $O(|H_\Delta| \log n)$ time. We then solve the subproblem for H_Δ in $T(|H_\Delta|)$ time, by running algorithm $\mathcal{A}$. We assume that algorithm $\mathcal{A}$ does not change the input array. Afterwards, we move the planes of H_Δ back to their original position; for example, if we record a transcript of the preceding $O(|H_\Delta|)$ swaps, using $O(|H_\Delta| \log n)$ bits, we can perform these swaps backwards in $O(|H_\Delta| \log n)$ time. (It is important that we go back to the home permutation, so as not to disturb the pointer structures for the conflict lists.) We can then proceed to the next cell Δ.

For $r = n/\log n$, the total expected running time is

$$\sum_i T(n_i) \;+\; T(n/\log n) \;+\; O(n \log n)$$

for some n_i's with $\sum_i n_i = O(n)$ and $\max_i n_i = O(\log^2 n)$. We do not need recursion. For algorithm $\mathcal{A}$, we can just simulate a known $O(n \log n)$ algorithm; this requires $T(n) = O(n \log^2 n)$ time in the permutation+bits model. (With $O(n \log^2 n)$ time, algorithm $\mathcal{A}$ indeed does not need to change the input array, by working with pointers.) This yields a final time bound of $O(n \log^2 \log n + n \log n) = O(n \log n)$. □

By applying Lemma 2.2 twice and then Lemma 2.3, we reach our final conclusion:

Corollary 2.6 *There is an in-place 3-d convex hull algorithm that runs in $O(n \log n)$ expected time.*

2.3 Derandomization?

Derandomization of our in-place $O(n \log n)$-time algorithm appears difficult because of the large choice of sample size r, so we leave open the question of finding an optimal deterministic in-place algorithm for 3-d convex hulls. We remark, however, that the in-place $O(n \log^2 n)$-time algorithm mentioned after Lemmas 2.2–2.3 can be derandomized using *ε-approximations* [34].

Formally, a subset $A \subseteq H$ is a $(1/r)$-approximation of H iff $|H_\Delta|/|H|$ and $|A_\Delta|/|A|$ differs by at most $1/r$ for every simplex Δ. Assuming that a $(1/r)$-approximation A of size $r^{O(1)}$ is given, we can compute a sample R satisfying the properties in Lemma 2.1 in $r^{O(1)}$ time and space, as shown by Chazelle and Matoušek [17, proof of Corollary 4]. In the proof of Lemma 2.2, the extra time and space in bits are polylogarithmic, which are allowed. In the proof of Lemma 2.3, the extra time and space in words of words of space are $O(1)$, which are again acceptable. It remains to describe how to construct an $(1/r)$-approximation A.

There is a known algorithm $\mathcal{A}$ for computing $(1/r)$-approximations in $O(n \log r)$ time when $r \leq n^\delta$ for a sufficiently small constant $\delta > 0$ [34]. In the permutation+bits model, $\mathcal{A}$ would take $O(n \log r \log n)$ time and $O(n \log r \log n)$ bits of extra space. To obtain an in-place algorithm, we divide the array into k subsets $H_1, \ldots, H_k$, each of size n/k, with $k = c' \log r \log n$ for some constant c'. For each i, we compute a $(1/(2r))$-approximation A_i of H_i of size $r^{O(1)}$ by simulating $\mathcal{A}$ on H_i. By choosing c' large enough, this requires $O(n \log r \log n)$ total time and at most εn bits of space for an arbitrarily small constant $\varepsilon > 0$. Using the bit-encoding trick, we can simulate the buffer area by permuting $2\varepsilon n$ pairs anywhere in the input array outside of H_i. We move $\bigcup_i A_i$ to a prefix of the array and finish by computing a $(1/(2r))$-approximation A of $\bigcup_i A_i$ by simulating $\mathcal{A}$ once again. Since $\bigcup_i A_i$ has size $r^{O(1)} \log r \log n$, the time needed for this last step is negligible and the space needed in bits is less than εn, and so the bit-encoding trick can again be used. The result A is a $(1/r)$-approximation of H, by known properties about ε-approximations (closure under unions and compositions) [34]. We have thus described an in-place algorithm to compute $(1/r)$-approximations with running time at most $O(n \log r \log n) = O(n \log^2 n)$ for $r \leq n^\delta$.

Corollary 2.7 *There is a deterministic in-place 3-d convex hull algorithm that runs in $O(n \log^2 n)$ time.*

2.4 An Output-Sensitive Version

We observe that with additional known techniques, the $O(n \log n)$ running time can be reduced to $O(n \log h)$ if h is the output size. This improves the previous output-sensitive in-place algorithm by Brönnimann, Chan, and Chen [10], which runs in $O(n \log^3 h)$ time. The result extends the previous 2-d $O(n \log h)$-time algorithm by Brönnimann *et al.* [11].

Corollary 2.8 *There is an in-place 3-d convex hull algorithm that runs in $O(n \log h)$ expected time where h denotes the number of hull vertices.*

Proof: Note that for $h \geq n^\varepsilon$, our $O(n \log n)$-time algorithm already runs in $O(n \log h)$ expected time. We may thus assume $h < n^\varepsilon$. By Lemma 2.3, it suffices to give an $O(n \log h)$-time algorithm that uses $O(n)$ bits of extra space in the permutation+bits model for this case. (The proof of Lemma 2.3 is still valid in the output-sensitive setting.)

We adapt Chan's (deterministic) output-sensitive convex hull algorithm [12]. We divide the given array P of points into n/m subarrays P_i, each containing m points for some parameter m. For each subarray P_i, we build an implicit data structure to support *gift-wrapping queries* by permuting P_i. (In a gift-wrapping query, we want to find the first point hit when we rotate a given plane around a given line outside the interior of the convex hull; in the dual, this is equivalent to finding the first plane hit by a given ray originating from inside a halfspace intersection.) Brönnimann, Chan, and Chen [10] provided such data structures; the simplest option is an implicit partition tree, which achieves query time $O(m^{1-\alpha})$ for some constant $\alpha > 0$. The preprocessing time for each subarray is $O(m \log m)$, for a total of $O(n \log m)$. (The preprocessing algorithm is in-place.)

Chan's algorithm computes the convex hull using $O(h)$ gift-wrapping queries on P, which reduce to $O(h(n/m))$ queries on the P_i's. The algorithm maintains a queue storing at most $O(h)$ facets (to implement a breadth-first search), and a dictionary storing $O(h)$ facets (to ensure that each facet is processed once). In the permutation+bits model, the algorithm requires $O(h \log n)$ bits of extra space; this number is sublinear since $h < n^\varepsilon$. The total time for the queries is $O(h(n/m)m^{1-\alpha})$. Excluding the preprocessing phase, the total time for the dictionary/queue operations is $O(h \log h)$ in the traditional model, and $O(h \log h \log n)$ in the permutation+bits model, because of pointer operations; this cost is sublinear since $h < n^\varepsilon$. If h is known, we can set $m = h^{1/\alpha}$ and get an overall running time of $O(n \log h)$ and an overall space bound of $O(n + h \log n) = O(n)$ in bits. A standard guessing trick [12] can remove the assumption that h is given in advance. $\square$

3 In-Place Segment Intersection

The same approach as in Section 2 works for the segment intersection problem. To avoid repetition, we only point out the differences. Here, H is a set of line segments, and $\mathcal{T}(R)$ is the trapezoidal decomposition of the arrangement of the segments in R. A new complication is that the analysis depends on the output parameter K, the number of intersections in H. For a random sample R of H of size r, the expected size of $\mathcal{T}(R)$ is now $O(r + Kr^2/n^2)$, and consequently Lemma 2.1(i) changes to

$$E\left[\sum_{\Delta \in \mathcal{T}(R)} |H_\Delta|^b\right] = O((r + Kr^2/n^2) \cdot (n/r)^b).$$

In Lemma 2.2, suppose that algorithm $\mathcal{A}$ specifically takes $O(n \log n + K)$ time. Then algorithm $\mathcal{A}'$ takes expected time $O((r + Kr^2/n^2) \cdot (n/r) \log n + K) = O(n \log n + K)$.

In Lemma 2.3, the algorithm can be slightly simplified since there are no output bits.

In Theorem 2.5, we again use offline 2-d point location (Lemma 2.4), this time, to locate a cell in $\mathcal{T}(R)$ containing each left endpoint of H. For each segment h, we can compute the list of all cells in $\mathcal{T}(R)$ intersecting h by a a depth-first search, starting at the left endpoint; in the traditional model, this takes time proportional to the sum of $\deg(\Delta)$ over $\Delta \in \mathcal{T}(R)$ intersecting h, where $\deg(\Delta)$ denotes the number of cells adjacent to Δ. From these lists, we obtain all the conflict lists H_Δ, stored in pointer structures in the buffer area. In the traditional model, the total time is $\sum_{\Delta \in \mathcal{T}(R)} |H_\Delta| \deg(\Delta)$. An analysis by Clarkson and Shor [20, proof of Theorem 4.6, 3rd paragraph] bounds the expectation of this expression by $O((r + Kr^2/n^2) \cdot (n/r))$. Setting $r = n/\log n$ as before would lead to an optimal expected time bound, but the number of bits of extra space used may not be $O(n \log n)$. Instead, we proceed iteratively with different choices of r as follows:

We set $r = n^2/K_j$, where $K_j = 2^j n \log n$ in the j-th trial. If $\sum_{\Delta \in \mathcal{T}(R)} |H_\Delta| \deg(\Delta) \le c'n$ and $\sum_{\Delta \in \mathcal{T}(R)} |H_\Delta|^2 \le c'K_j$ for a sufficiently large constant c', we declare the j-th trial successful and solve the subproblems directly by brute force in expected time $O(K_j)$. The brute-force algorithm simply checks all pairs and needs no extra space.

If $K_j \ge K$, then $E\left[\sum_{\Delta \in \mathcal{T}(R)} |H_\Delta| \deg(\Delta)\right] = O((r + Kr^2/n^2) \cdot (n/r)) = O(n)$ and $E\left[\sum_{\Delta \in \mathcal{T}(R)} |H_\Delta|^2\right] = O((r + Kr^2/n^2) \cdot (n/r)^2) = O(K_j)$. So, if $K_j \ge K$, the probability that iteration j is not successful can be bounded by an arbitrarily small constant p. Letting f denote the final iteration, we have $\Pr\{K_f > 2^i \max\{n \log n, K\}\} \le p^i$, and so $E[K_f] = O(\max\{n \log n, K\})$. Thus, the expected total cost is $O(n \log n + K)$.

Note that we can suppress printing during all iterations and rerun the final iteration to print the intersections. We conclude:

Theorem 3.1 *There is an in-place 2-d segment intersection algorithm that runs in $O(n \log n + K)$ expected time, where K denotes the number of intersections.*

4 Cache-Oblivious 3-d Convex Hulls

In this section, we switch to the cache-oblivious model and give a simpler re-derivation of an optimal 3-d convex hull algorithm in this model, first obtained by Kumar and Ramos [33]. As in Section 2.2, the approach is based on random sampling. Again we need an efficient offline point location method as a subroutine. The lemma below serves this purpose.

Lemma 4.1 *Suppose we are given a set P of N points and an arrangement R of r hyperplanes in a constant dimension d. We can locate the cell γ_p in the arrangement containing each point $p \in P$, in $O((N/B) \log_M r + r^{O(1)})$ memory transfers.*

The proof of this lemma was given by Kumar and Ramos [33] and will not be reproduced here. Roughly, we can adopt, for example, the naive *slab method* [23] for point location, which normally has $O(\log r)$ query time and $r^{O(1)}$ preprocessing time. The key subproblem is locating a set of points against a set of nonintersecting hyperplanes inside a vertical slab. This subproblem can be solved by a variant of an optimal cache-oblivious sorting algorithm. (One cannot directly apply a sorting algorithm, however, since points are not comparable against each other, but one can still adopt a variant of the cache-oblivious *distribution sort* [30]. Our situation is actually a little simpler, since we do not need to sort all N elements but just distribute N elements into r "buckets", while tolerating an extra $r^{O(1)}$ term.)

Because of the larger overhead $r^{O(1)}$ term, our cache-oblivious convex hull algorithm will need recursion with a different sample size. The algorithm is still conceptually simple and is described in the following theorem.

Below, it is more convenient to switch to another common definition of random samples: a *p-sample* R of H refers to a subset obtained by including each element in H independently with probability p.

Theorem 4.2 *There is a 3-d convex hull algorithm that takes $O((N/B)\log_M N)$ expected number of memory transfers in the cache-oblivious model.*

Proof: Fix a sufficiently small constant $\delta > 0$. Given a set H of N planes, a parameter $r^* \le N$, and an (r^*/N)-sample R^* of H, we describe a procedure to compute $\mathcal{E}(R^*)$ along with the conflict lists H_{Δ^*} for all $\Delta^* \in \mathcal{T}(R^*)$.

Draw an (r/r^*)-sample R of R^*, with $r = \min\{N^\delta, r^*\}$. Compute $\mathcal{E}(R)$ and $\mathcal{T}(R)$ naively, say, with $O(r\log r)$ time/memory transfers. For each plane $h \in H$, compute the list V_h of all vertices of $\mathcal{E}(R)$ above h. In dual space, this is equivalent to finding all planes below each of N query points in an arrangement of $O(r)$ planes. This task reduces to N offline point location queries in the arrangement—elements h associated with the same cell have the same set V_h. By Lemma 4.1, the cost of locating the cells is $O((N/B)\log_M r + r^{c'})$ for some constant c'. For each of the $O(r^3)$ cells of the arrangement, we precompute the list of (at most $O(r)$) planes below the cell; this takes $O(r^4)$ time. By applying a cache-oblivious sorting algorithm to the results of the point location, using cells as the keys, and scanning the corresponding precomputed lists, we can then compute all the V_h's with $O((X/B)\log_M N)$ memory transfers, where X is the total size of the lists. By using vertices as the keys instead, we can compute the lists H_v of all planes below each vertex v of $\mathcal{E}(R)$. We can also generate the conflict list H_Δ for each $\Delta \in \mathcal{T}(R)$, by more sorting and scanning (since H_Δ is the union of H_v over three vertices v of Δ). We now recursively solve the subproblem for H_Δ and $H_\Delta \cap R^*$ for each $\Delta \in \mathcal{T}(R)$. As a result, we can obtain a list of all vertices of $\mathcal{E}(R^*)$ and and a list H_v of all planes below each vertex v of $\mathcal{E}(R^*)$. By applying a cache-oblivious sorting algorithm to the vertices of $\mathcal{E}(R^*)$, we can generate the list of all vertices of $\mathcal{E}(R^*)$ incident to each plane h, and thus compute the edges and faces of $\mathcal{E}(R^*)$, as well as $\mathcal{T}(R^*)$, with $O((r^*/B)\log_M r^*)$ additional memory transfers. We can also generate the conflict lists H_{Δ^*} for all $\Delta^* \in \mathcal{T}(R^*)$ from the lists H_v by sorting and scanning.

Since R is an (r/N)-sample of H, Lemma 2.1 implies that $E[\sum_{\Delta\in\mathcal{T}(R)}|H_\Delta|] = O(N)$, and thus $E[X] = O(N)$, and $\max_{\Delta\in\mathcal{T}(R)}|H_\Delta| = O((N/r)\log t)$ with probability $1 - O(r/t)$. Note that for any fixed h, we have $p' := \Pr\{h \in R^* \mid h \notin R\} = \Pr\{h \in R^* \setminus R\}/\Pr\{h \in R^*\} = \frac{r^*/N - r/N}{1-r/N} \le r^*/N$. Conditioned to a fixed R, H_Δ is fixed and $H_\Delta \cap R^*$ is a p'-sample of H_Δ. Note also that the $O(r^{c'}) = O(N^{c'\delta})$ term never dominates if δ is sufficiently small, since $N > M > B^{1+\gamma}$ by the tall cache assumption, implying $N/B > N^{1-1/(1+\gamma)}$. We thus have the following recurrence for the total cost of the above procedure:

$$T(N, r^*) \;=\; \begin{cases} \sum_i T(N_i, r^*N_i/N) + O((N/B)\log_M N) & \text{if } N > M \text{ and } r^* > N^\delta \\ O((N/B)\log_M N) & \text{else} \end{cases}$$

where the N_i's are random variables satisfying $E[\sum_i N_i \mid N] = O(N)$, and $\max_i N_i = O(N^{1-\delta}\log t)$ with probability $1 - O(N^\varepsilon/t)$.

So far, the above is just a reinterpretation of Kumar and Ramos' basic strategy. If we put $r^* = N$, the result would be suboptimal by logarithmic factors because of constant-factor blow-up ($c > 1$), as Kumar and Ramos noted. To get an optimal algorithm, they added more complicated "pruning steps", which involve solving 2-d subproblems. Our new idea is simple: set $r^* = N^{1-\varepsilon}$ instead. It is easy to see then that the recursion depth is constant, so we get $T(N, N^{1-\varepsilon}) = O((N/B)\log_M N)$ for any constant $\varepsilon > 0$ (where the constant factor depends on ε), with probability at least $1 - 1/N$, e.g., by setting t to be a polynomial in N.

For the overall algorithm, we draw an (r^*/N)-sample R^* with $r^* = N^{1-\varepsilon}$, run the above procedure, and then recursively compute the lower envelope for H_Δ inside Δ for each $\Delta \in \mathcal{T}(R^*)$. (Restart if (1) fails for R^*.)

The final recurrence for the expected cost is

$$T(N) \;=\; \begin{cases} \sum_i T(N_i) + O((N/B)\log_M N) & \text{if } N > M \\ O(N/B) & \text{else} \end{cases}$$

where $\sum_i N_i \le cN$ and $\max_i N_i = O(N^\varepsilon \log N)$. By choosing a constant $\varepsilon < 1/c$, we can verify by induction that the recurrence has solution $O((N/M)\log_M N)$, since

$$\begin{aligned} \left(\sum_i c'(N_i/B)\log_M N_i\right) + (N/B)\log_M N \;&\le\; \left(\sum_i c'\varepsilon(N_i/B)\log_M N\right) + (N/B)\log_M N \\ &\le\; (c'c\varepsilon+1)(N/B)\log_M N \;\le\; c'(N/B)\log_M N, \end{aligned}$$

for $c' > 1/(1-c\varepsilon)$. □

To summarize, our simpler algorithm is just a variant of a standard sampling strategy, where we repeatedly take samples of size $r = N^\delta$. We apply this strategy for only a constant number of layers, and then switch to recursion. In contrast, repeated application of this strategy alone would yield suboptimal cache-oblivious results because of the nonconstant number of layers, whereas repeated recursion alone would also yield suboptimal results because of the constant-factor blow-up.

5 Cache-Oblivious Segment Intersection

The same approach as in Section 4 works for the segment intersection problem in the cache-oblivious model. This time, we obtain a new result. We now point out the main differences.

The computation of conflict lists here can still be reduced to point location for hyperplanes (so Lemma 4.1 is still applicable), but we need to lift to a higher dimension: Specifically, map each input line segment h to a point $f(h) = (\alpha, \beta, \alpha', \beta', \xi, \eta) \in \mathbb{R}^6$, where (α,β) and (α',β') are the coordinates of the left and right endpoints, and $\xi x + \eta y = 1$ is the equation of the line through h. For each vertex of $\mathcal{T}(R)$ with coordinates (a,b), form the hyperplane $a\xi + b\eta = 1$. For each edge of $\mathcal{T}(R)$ with line equation $ax + by = 1$, form the hyperplanes $a\alpha + b\beta = 1$ and $a\alpha' + b\beta' = 1$. If $f(h_1)$ and $f(h_2)$ lie in the same cell of the arrangement of these $O(r)$ hyperplanes in $\mathbb{R}^6$, then the segments h_1 and h_2 have the same set of intersecting trapezoids in $\mathcal{T}(R)$, as required.

The first recurrence is changed to

$$T(N, r^*, K) \;=\; \begin{cases} \sum_i T(N_i, r^*N_i/N, K_i) + O((N/B)\log_M N) & \text{if } N > M \text{ and } r^* > N^\varepsilon \\ O((N/B)\log_M N) & \text{else} \end{cases}$$

where the N_i's and K_i's are random variables satisfying $E[\sum_i N_i \mid N, K] = O((r + Kr^2/N^2)\cdot N/r) = O(N + K/N^{1-\varepsilon})$, $\max_i N_i = O(N^{1-\varepsilon}\log t)$ with probability $1 - O(N^\varepsilon/t)$, and $\sum_i K_i = K$. For $r^* \le N^{1-\delta}$, the recursion depth is constant, and by induction, one can verify that $T(N, r^*, K) = O(\frac{N+Kr^*/N}{B}\log_M N)$.

The second recurrence becomes

$$T(N, K) \;=\; \begin{cases} \sum_i T(N_i, K_i) + O(\frac{N+K/N^\delta}{B}\log_M N) & \text{if } N > M \\ O(N/B) & \text{else} \end{cases}$$

where $\sum_i N_i \le c(r + Kr^2/N^2) \cdot N/r = c(N + K/N^\delta)$, $\max_i N_i = O(N^\delta \log N)$, and $\sum_i K_i = K$. By induction, one can verify that $T(N, K) = O((N/B) \log_M N + K/B)$ for $\delta < 1/c$. Similarly, the expected space is $O(N + K)$, which can be made worst-case by repetition.

The space bound can be reduced to $O(N)$ with additional steps. We consider two cases:

- $K \le N^{1+\delta}$. We pick an (r^*/N)-sample R^* with $r^* = N^{1-\delta}$. Generating the conflict lists costs $T(N, r^*, K) = O(\frac{N+Kr^*/N}{B} \log_M N) = O((N/B) \log_M N)$ and expected space $O(N)$. Let K_Δ denote the number of intersections in Δ. We solve the subproblems for H_Δ by the preceding algorithm. This requires expected total cost $O(E\left[\sum_{\Delta \in \mathcal{T}(R^*)}((|H_\Delta|/B) \log_M |H_\Delta| + K_\Delta/B))\right]) = O((N/B) \log_M N + K/B)$ and space $O(\max_{\Delta \in \mathcal{T}(R^*)}(|H_\Delta| + K_\Delta)) = O(\max_\Delta |H_\Delta|^2) = O(N^{2\delta} \log N) = o(N)$ with high probability. Whenever the space usage exceeds $c''N$ for a sufficiently large constant c'', we go to a new trial; the expected number of trials is $O(1)$.

- $K > N^{1+\delta}$. We pick an (r^*/N)-sample R^* with $r^* = N^2/K_j$, where $K_j = 2^j N^{1+\delta}$ in the j-th iteration. Generating the conflict lists costs $T(N, r^*, K) = O(\frac{N+Kr^*/N}{B} \log_M N) = O(\frac{K/N^\delta}{B} \log_M N)$; the total is $O(K/B)$ even if we sum over all possible j's. We stop the conflict-list computation and skip to the next iteration when the space usage exceeds $c''N$ for a sufficiently large constant c''. We also skip to the next iteration if $\sum_{\Delta \in \mathcal{T}(R^*)} |H_\Delta|^2 > c''K_j$. Otherwise, we declare iteration j successful and solve all the subproblems by brute force, with cost $O(K_j/B)$.

  If $K_j \ge K$, then the conflict-list computation takes expected space $O(N + Kr^*/N) = O(N)$, and $E\left[\sum_{\Delta \in \mathcal{T}(R^*)} |H_\Delta|^2\right] = O(r^* + Kr^{*2}/N^2) \cdot (N/r^*)^2) = O(K_j)$. So, if $K_j \ge K$, the probability that iteration j is not successful can be bounded by an arbitrarily small constant p. Letting f denote the final iteration, we have $\Pr\{K_f > 2^i K\} \le p^i$, and so $E[K_f] = O(K)$. Thus, the expected total cost in this case is $O(K/B)$.

We can run the algorithms in the two cases concurrently in the cache-oblivious model. We conclude:

Theorem 5.1 *There is a 2-d segment intersection algorithm that takes $O((N/B) \log_M N + (K/B))$ expected number of memory transfers and uses $O(N)$ space in the cache-oblivious model, where K is the output size.*

Acknowledgements

We thank the anonymous referees for their detailed and useful comments.

References

[1] P. K. Agarwal, L. Arge, and K. Yi. I/O-efficient construction of constrained Delaunay triangulations. In *Proc. 13 European Sympos. Algorithms*, Lect. Notes Comput. Sci., vol. 3669, Springer-Verlag, pages 355–366, 2005.

[2] N. Amenta, S. Choi, and G. Rote. Incremental constructions con BRIO. In *Proc. 19th ACM Sympos. Comput. Geom.*, pages 211–219, 2003.

[3] L. Arge, T. Mølhave, and N. Zeh. Cache-oblivious red-blue line segment intersection. In *Proc. 16th European Sympos. Algorithms*, Lect. Notes Comput. Sci., vol. 5193, Springer-Verlag, pages 88–99, 2008.

[4] I. J. Balaban. An optimal algorithm for finding segment intersections. In *Proc. 11th ACM Sympos. Comput. Geom.*, pages 211–219, 1995.

[5] H. Blunck and J. Vahrenhold. In-place algorithms for computing (layers of) maxima. In *Proc. 10th Scand. Workshop Algorithm Theory*, Lect. Notes Comput. Sci., vol. 4059, Springer-Verlag, pages 363–374, 2006.

[6] P. Bose, E. Y. Chen, M. He, A. Maheshwari, and P. Morin. Succinct geometric indexes supporting point location queries. In *Proc. 20th ACM-SIAM Sympos. Discrete Algorithms*, pages 635–644, 2009.

[7] P. Bose, A. Maheshwari, P. Morin, J. Morrison, M. Smid, and J. Vahrenhold. Space-efficient geometric divide-and-conquer algorithms. *Comput. Geom. Theory Appl.*, 37:209–227, 2007.

[8] G. S. Brodal. Cache-oblivious algorithms and data structures. In *Proc. 9th Scand. Workshop Algorithm Theory*, Lect. Notes Comput. Sci., vol. 3111, Springer-Verlag, pages 3–13, 2004.

[9] H. Brönnimann and T. M. Chan. Space-efficient algorithms for computing the convex hull of a simple polygonal line in linear time. *Comput. Geom. Theory Appl.*, 34:75–82, 2006.

[10] H. Brönnimann, T. M. Chan, and E. Y. Chen. Towards in-place geometric algorithms and data structures. In *Proc. 20th ACM Sympos. Comput. Geom.*, pages 239–246, 2004. Full version at `http://www.cs.uwaterloo.ca/~tmchan/inplace_jsub.pdf`

[11] H. Brönnimann, J. Iacono, J. Katajainen, P. Morin, J. Morrison, and G. T. Toussaint. Space-efficient planar convex hull algorithms. *Theoret. Comput. Sci.*, 321:25–40, 2004.

[12] T. M. Chan. Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discrete Comput. Geom.*, 16:361–368, 1996.

[13] T. M. Chan and E. Y. Chen. In-place 2-d nearest neighbor search. In *Proc. 19th ACM-SIAM Sympos. Discrete Algorithms*, pages 904–911, 2008.

[14] T. M. Chan and M. Pătraşcu. Transdichotomous results in computational geometry, I: Point location in sublogarithmic time. *SIAM J. Comput.*, to appear. Preliminary versions in *Proc. 47th IEEE Sympos. Found. Comput. Sci.*, pages 325–332 and 333–342, 2006.

[15] T. M. Chan, J. Snoeyink, and C.-K. Yap. Primal dividing and dual pruning: output-sensitive construction of four-dimensional polytopes and three-dimensional Voronoi diagrams. *Discrete Comput. Geom.*, 18:433–454, 1997.

[16] B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. *J. ACM*, 39:1–54, 1992.

[17] B. Chazelle and J. Matoušek. Derandomizing an output-sensitive convex hull algorithm in three dimensions. *Comput. Geom. Theory Appl.*, 5:27–32, 1995.

[18] E. Y. Chen and T. M. Chan. A space-efficient algorithm for segment intersection. In *Proc. 15th Canad. Conf. Comput. Geom.*, pages 68–71, 2003.

[19] K. L. Clarkson. New applications of random sampling in computational geometry. *Discrete Comput. Geom.*, 2:195–222, 1987.

[20] K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, II. *Discrete Comput. Geom.*, 4:387–421, 1989.

[21] A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, and E. Ramos. Randomized external-memory algorithms for line segment intersection and other geometric problems. *Int. J. Comput. Geom. Appl.*, 11:305–337, 2001.

[22] E. D. Demaine. Cache-oblivious algorithms and data structures. Manuscript, 2002. `http://erikdemaine.org/papers/BRICS2002/`

[23] D. Dobkin and R. J. Lipton. Multidimensional searching problems. *SIAM J. Comput.*, 5:181–186, 1976.

[24] H. Edelsbrunner, L. Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. *SIAM J. Comput.*, 15:317–340, 1986.

[25] G. Franceschini and V. Geffert. An in-place sorting with $O(n \log n)$ comparisons and $O(n)$ moves. *J. ACM*, 52:515–537, 2005.

[26] G. Franceschini and R. Grossi. Optimal implicit dictionaries over unbounded universes. *Theory of Comput. Sys.*, 39:321–345, 2006.

[27] G. Franceschini, R. Grossi, J. I. Munro, and L. Pagli. Implicit B-trees: a new data structure for the dictionary problem. *J. Comput. Sys. Sci.*, 68:788–807, 2004.

[28] G. Franceschini and S. Muthukrishnan. In-place suffix sorting. In *Proc. 34th Int. Colloq. Automata, Languages, and Programming*, Lect. Notes Comput. Sci., vol. 4596, Springer-Verlag, pages 533–545, 2007.

[29] G. Franceschini, S. Muthukrishnan, and M. Pătraşcu. Radix sorting with no extra space. In *Proc. 15th European Sympos. Algorithms*, Lect. Notes Comput. Sci., vol. 4698, Springer-Verlag, pages 194–205, 2007.

[30] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. 40th IEEE Sympos. Found. Comput. Sci.*, pages 285–297, 1999.

[31] D. Kirkpatrick. Optimal search in planar subdivisions. *SIAM J. Comput.*, 12:28–35, 1983.

[32] P. Kumar. Cache oblivious algorithms. *Algorithms for Memory Hierarchies*, Lect. Notes Comput. Sci., vol. 2625, Springer-Verlag, pages 193–212, 2003.

[33] P. Kumar and E. Ramos. I/O efficient construction of Voronoi diagrams. Manuscript, 2002. `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.59.6760`

[34] J. Matoušek. Derandomization in computational geometry. In *Handbook of Computational Geometry* (J. Urrutia and J. Sack, ed.), North-Holland, pages 559–595, 2000.

[35] K. Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms.* Prentice-Hall, Englewood Cliffs, N.J., 1994.

[36] J. I. Munro. An implicit data structure supporting insertion, deletion, and search in $O(\log^2 n)$ time. *J. Comput. Sys. Sci.*, 33:66–74, 1986.

[37] F. P. Preparata. A new approach to planar point location. *SIAM J. Comput.*, 10:473–482, 1981.

[38] F. P. Preparata and S. J. Hong. Convex hulls of finite sets of points in two and three dimensions. *Commun. ACM*, 20:87–93, 1977.

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

[40] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Commun. ACM*, 29:669–679, 1986.

[41] R. Seidel and U. Adamy. On the exact worst case complexity of planar point location. *J. Algorithms*, 37:189–217, 2000.

[42] J. Vahrenhold. Line-segment intersection made in-place. *Comput. Geom. Theory Appl.*, 38:213–230, 2007.