Algorithmica (1996) 16: 569–617

Algorithmica
© 1996 Springer-Verlag New York Inc.

A Nearly Optimal Deterministic Parallel Voronoi Diagram Algorithm[1]

R. Cole,[2] M. T. Goodrich,[3] and C. Ó Dúnlaing[4]

Abstract. We describe an n-processor, $O(\log(n) \log\log(n))$-time CRCW algorithm to construct the Voronoi diagram for a set of n point-sites in the plane.

Key Words. Voronoi diagram, Parallel algorithm.

1. Introduction. Outline of the Algorithm. The Voronoi diagram is a geometric structure of great computational interest: see [5] for a useful survey. This paper addresses the problem of constructing the diagram in parallel, given as input a set of n points ("sites") in the plane. The Voronoi diagram for a set of sites is the locus of points equidistant from two closest sites: Figure 1 illustrates a diagram with 32 sites.

The model of parallelism we assume is a CRCW PRAM, a system of independent processors accessing a shared random-access memory, where the same memory cell can be read by several processors simultaneously (concurrent read) and written by several processors simultaneously (concurrent write). Write-conflicts are resolved arbitrarily: the model of computation is an ARBITRARY CRCW PRAM.

Each processor is assumed capable of exact rational and integer arithmetic in unit time.

Earlier algorithms [1], [9] were presented for CREW[5] machines. The algorithm in [1] used n processors and took $O(\log^2(n))$ parallel time; that in [2] used $n \log(n)$ processors and took $O(\log(n) \log\log(n))$ parallel time. Our algorithm reduces the overall work (parallel time $\times$ number of processors) to $O(n \log(n) \log\log(n))$, while maintaining a runtime of $O(\log(n) \log\log(n))$. Both of these figures are within the factor $\log\log(n)$

[1] A preliminary version of this paper was presented at the 17th EATCS ICALP meeting at Warwick, England, in July 1990.
[2] Department of Computer Science, Courant Institute, 251 Mercer Street, New York, NY 10012, USA. cole@cs.nyu.edu. Supported by the US NSF under Grants CCR 890221 and CCR 8906949.
[3] Department of Computer Science, Johns Hopkins University, Baltimore, MD 21218, USA. goodrich@cs.jhu.edu. Supported by the US NSF under Grants CCR 8810568, CCR-9003299, and IRI-9116843, and by the NSF and DARPA under Grant CCR 8908092.
[4] School of Mathematics, Trinity College, Dublin 2, Ireland. odunlain@maths.tcd.ie. Supported by the EU Esprit program under BRAs 3075 (ALCOM) and 7141 (ALCOM II).
[5] Exclusive-write, that is, no write conflicts are allowed. We use the concurrent-write mechanism for forward chaining and integer sorting [6].

Received May 1, 1994; revised December 20, 1994. Communicated by K. Melhorn.

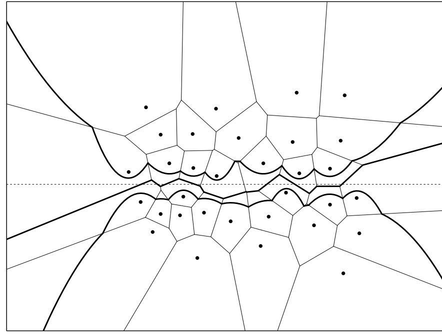

Fig. 1. Voronoi diagram formed from two sets P and Q to the top and bottom of the dashed horizontal line L. The (P, L)- and (Q, L)-beachlines are illustrated, and the (P, Q)-contour edges are darkened. Note that the contour lies between the two beachlines. The x-direction is upwards.

of optimal.[6] Our technique, like all[7] previous deterministic parallel algorithms, is based on the serial algorithm due to Shamos and Hoey [25]. The set of sites is initially sorted by x-coordinate; then the algorithm proceeds recursively:

- Partition the input set S of n sites into two sets P and Q of size $n/2$ by a vertical straight line L.
- Compute the Voronoi diagrams of the left and right half-sets recursively, with $n/2$ processors assigned to each; call these $\mathrm{Vor}(P)$ and $\mathrm{Vor}(Q)$ respectively.
- Compute the *contour*, the locus of all points in the plane equidistant from P and Q (the contour is illustrated in Figure 1).
- *Stitch* the diagrams together along the contour.

It was shown in [1] how to compute the contour and stitch the diagrams together in $\log(n)$ parallel time. This is done, roughly speaking, as follows:

- Each edge of $\mathrm{Vor}(P)$ can meet the contour at most twice; for simplicity we assume at most once: using suitable data structures and one processor per edge of $\mathrm{Vor}(P)$, those edges which meet the contour are identified. We call such edges "attachments," and the points where they meet the contour (necessarily vertices of $\mathrm{Vor}(S)$) their "ends." This takes $O(\log(n))$ parallel time. Likewise for $\mathrm{Vor}(Q)$.
- The edges of $\mathrm{Vor}(P)$ meeting the contour can be ranked and sorted according to the y-coordinates of their ends, without knowing these coordinates, in $O(\log(n))$ parallel steps.
- Among all the attachments in $\mathrm{Vor}(P)$, ranked along the contour, let e be the median attachment. Its end can be calculated using processors assigned to all attachments from Q in one parallel step; this subdivides the attachments from Q and permits, ultimately, calculation of all the ends from P, in $O(\log(n))$ time. Likewise for Q.

[6] This is substantially better than earlier deterministic algorithms, but randomized parallel algorithms have been described which achieve optimal *expected* time and work [24].

[7] All, that is, except the earliest [8], which ran in $O(\log^3(n))$ time, using a transformation to the three-dimensional convex hull problem.

- Once all the attachments and ends have been calculated, it is straightforward to complete construction of Vor(S) in $O(\log(n))$ steps.

In this paper we show that it is possible to compute the contour in $O(\log\log(n))$ parallel time, and stitch in constant parallel time. The ideas are derived from Valiant's [27] array-merging algorithm. In place of a data-structure which allows one processor to answer certain queries in serial time $O(\log(n))$, we have one allowing $\sqrt{n}$ processors to answer such queries in $O(1)$ parallel time. The structure involves what are called *beachlines*[8] and *fringes* [1]. Beachlines (but not fringes) are illustrated in Figure 1.

Given a set P of sites and a (vertical) line L which has all of P to its left (or right), the "beachline" between P and L is the set of points in the plane equidistant from L and P. It is a union of parabolic segments; the foci of the parabolas are in P and they all have the same directrix L. The beachline "cusps" (where adjacent parabolic segments meet) are the points where it crosses edges of Vor(P). The beachline thus divides Vor(P) into a part nearer P than L and a part further from P than L. The latter part is called the "fringe." It has the structure of a forest of trees; each tree contains exactly one unbounded edge of Vor(P). These trees are essentially binary trees, and the structure of the fringe can be used to build the contour in $O(\log\log(n))$ parallel time.

An important advantage of our structure is that not only are the attachments calculated, but also the ends, which allows two of the above steps to be bypassed. This avoids an $\Omega(\log(n))$ bottleneck caused by list-ranking [13].

Another feature of our methods is that the structures can have surplus processors allotted. The beachline B is a sorted list, and it would be convenient to store it in sorted order in an array. However, to avoid the need for excessive reorganization between phases of the algorithm, we allow B to be represented in an array A of size $O(n)$, in which an element from B might be represented by a block of several contiguous entries from A. The array A is then used to assign processors to the elements in B; only those processors assigned to the first copy of an element will be active. By this means we avoid compressing data by parallel prefix, another potential bottleneck [23].

It appears that the main difficulty in our approach is recursive construction of the beachlines, which would be trivial in the serial case. In [9] the beachlines were all precomputed independently in $O(\log(n))$ parallel steps using fractional cascading [3]. This allowed the algorithm to proceed without difficulty but was processor-inefficient, since the precomputed structures were built using $n\log(n)$ processors.

The improved method described here uses n processors to precompute not the beachlines but partial information about the beachlines. Specifically, given a set B of k points in the plane, sorted by y-coordinate, a *ruling* for B is a set of $O(k/\log(n))$ horizontal lines such that each horizontal strip contains $O(\log(n))$ points in B. We will see how to precompute rulings for the beachlines with n processors. During the algorithm, a linked-list structure for a beachline B can be used to build an array A covering B: A has $O(k/\log(n))$ blocks, each of size $O(\log(n))$, covering the strips of a given ruling of B. Each strip contains $O(\log(n))$ elements of B, which can be sorted in time $O(\log\log(n))$ by list-ranking.

---

[8] The notion of beachline, invented by Chee Yap, was first described in [1].

The contributions of this paper (aside from the result stated in the title) are:

(a) Detailed geometrical analysis of beachline and fringe.
(b) A special-purpose planar point location structure for the fringe.
(c) Usage of duplicate array entries to avoid array compression.
(d) An unusual application of stable integer sorting.

Sections 2–8 of this paper cover the material as follows. Section 2 introduces the Voronoi diagram together with the convex hull, and proves some facts about "beachlines" and "fringes." Section 3 reviews some parallel operations on arrays, including Valiant's merging technique and some results from [6]. Section 4 introduces the point-location structure definable from a fringe, and Section 5 shows how to use it to locate contour vertices. Section 6 shows how the Voronoi diagram itself can be built recursively during this process. Section 7 (which is long) shows how to build these fringe structures during the recursive processing, with the aid of precomputed "rulings," and Section 8 shows how to precompute the necessary "rulings." Concluding remarks are in Section 9.

REMARKS. It is implicit in our algorithm that most of the processing will take place in sorted arrays of data, and processors will be attached to entries in such arrays. The general steps involve processors attached to a block of entries in one array inspecting either an evenly spaced sequence of entries in another array, or a few contiguous intervals of entries in another array. These tasks do not involve processor-allocation problems such as occur in list processing; processor allocation involves simple arithmetic calculations and does not complicate our algorithm.

Most of the structures are defined relative to a vertical "reference line" L. In some of the diagrams, to save space, L will be shown as horizontal.

2. Definitions, Notations, and Terms.

In this section the notions of convex hull, Voronoi diagram, contour, beachline, and fringe are defined. Various geometrical properties are demonstrated; the section summary indicates where such properties will be useful.

2.1. We consider throughout a set S of *sites*, points in the plane; n will be the number of sites in S. For convenience, the following assumptions are made about S.

- n is a power of 2.
- For no two distinct pairs $\{p, q\}$, $\{p', q'\}$ of sites are the lines pq and $p'q'$ parallel.[9]
- No four sites are concyclic.
- No two sites have the same x-coordinate.

2.2. The *convex hull* $H(S)$ of S is the smallest convex set containing S (Figure 2). Its boundary is a convex polygon whose corners are sites of S. The distance $|x - y|$ between two points in the plane is Euclidean distance. For any point x, the distance of

[9] The distinct pairs could have one site in common, so this implies no three sites are collinear.

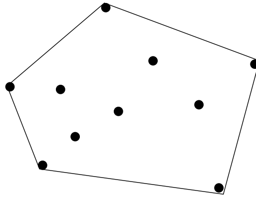

Fig. 2. Convex hull of a set of point sites.

x to a nonempty set T of points is

$$d(x, T) = \inf\{|x - t| \colon t \in T\}.$$

For any site p its *Voronoi cell* (with respect to S) is the set of points x in the plane such that $|x - p| = d(x, S)$, i.e., the set of points as close or closer to p than to any other site in S.

Let x be a point in the plane. Its *clearance circle* relative to S is the circle centred at x and of radius $d(x, S)$. This is the largest circle centred at x whose interior contains no site in S. The Voronoi cell of a site p can be defined as the set of all points whose clearance circle touches p.

The cell owned by p can be expressed as the intersection of $n - 1$ closed half-planes; because, for every other site q, the set of points equidistant from p and q is a straight line, and the set of points as close or closer to p than to q is a closed half-plane bounded by this line. Therefore the cell is a (topologically) closed convex region whose boundary is an open or closed polygon.[10]

DEFINITION 2.3. The *Voronoi diagram* Vor(S) is the union of all these cell boundaries.

Equivalently, the Voronoi diagram consists of every point in the plane whose clearance circle touches two or more sites.

2.4. The Voronoi diagram is a plane graph with n faces, one for each site, and hence it has $O(n)$ edges and vertices. The cells owned by the corners of $H(S)$ are unbounded, and all other cells are bounded. The unbounded edges are infinite rays perpendicular to the sides of $H(S)$ (and collinear with the midpoints of its sides, of course). The vertices are those points in the plane whose clearance circles touch at least three (hence, in view of 2.1, exactly three) sites.

We begin with a simple lemma about Voronoi vertices. It will be used in Lemma 2.20 below.

LEMMA 2.5. *Let v be a Voronoi vertex, and let V be any line through v. Then Voronoi edges from v extend on both sides of V.*

PROOF. See Figure 3. Without loss of generality, v is the only vertex, and V is vertical. If none of the incident edges extends rightwards from v, then one of the incident cells,

[10] The nondegeneracy assumptions 2.1 eliminate the possibility that its boundary is two parallel lines.

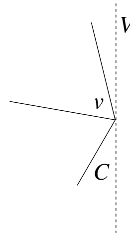

Fig. 3. Illustrating Lemma 2.5.

C say, contains all points to the right of V. This is impossible since C is convex with a corner at v. □

2.6. Construction of the diagram will be by divide-and-conquer. Initially, the sites are sorted by x-coordinate (which are all distinct, 2.1). The recurrence step involves partitioning S into two equal-size sets P and Q separated by a vertical line L passing between the two median elements.

DEFINITION 2.7. The (P, Q)-*contour* [25] is the set of points x in the plane such that $d(x, P) = d(x, Q)$.

(The (P, Q)-contour is an infinite zigzag line, monotonic in the y-direction [25]: see Figure 1.) Then Vor(S) is the union of the contour together with that part of Vor(P) to its left and of Vor(Q) to its right.

Throughout this paper, S, L, P, and Q play these rôles. Since P and Q play almost identical rôles, any statement involving P applies, suitably altered, to Q.

DEFINITION 2.8. (See Figure 4.) The (P, L)-*beachline* is the set of points x such that $d(x, P) = d(x, L)$.

LEMMA 2.9.

(i) *The* (P, L)*-beachline is infinite, piecewise parabolic, and monotonic in the* y*-direction, each segment is contained in a cell of* Vor(P), *and its cusps* (*points common to two adjacent segments*) *are on the edges of* Vor(P).

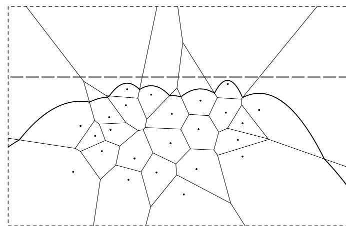

Fig. 4. Vor(P), reference line L, (P, L)-beachline and -fringe. The x-direction is upwards and L is shown horizontal.

(ii) *The beachline crosses each edge of* Vor(P) *at most twice*: *hence there are* $O(n)$ *beachline segments*.

(iii) *The contour lies strictly between the* (P, L)*- and* (Q, L)*-beachlines*.

PROOF. (i) For each site p in P, let R_p consist of all points closer to p than L. The boundary of this region is the parabola with focus p and directrix L. The beachline is the boundary of the union of all these regions R_p: hence it is infinite, monotonic in the y-direction, and piecewise parabolic.

If a point x on the beachline is interior to a segment with focus p, then its clearance circle touches p alone and hence x is in the cell for p. If it is on the boundary of two segments, then it is on the edge of Vor(P) separating the two associated foci.

(ii) An edge bounding the cell of the site p can meet the beachline only on the parabola bounding R_p, hence it can meet it at most twice.

(iii) If a point x is on the (P, Q)-contour, then its clearance circle touches both P and Q; hence this circle intersects L properly and x lies between the two beachlines. □

REMARK 2.10. We have assumed that the processors are capable of exact rational arithmetic. The sites, and any vertical separating line L, are assumed, of course, to have rational coordinates. Given two sites p and q, the perpendicular bisector (which contains the Voronoi edge separating them, if it exists) satisfies a rational linear equation: hence the Voronoi vertices have rational coordinates.

This does not generally hold for the beachline cusps. However, it is easy to see that their coordinates satisfy quadratic equations with rational coefficients, hence are of the form $a \pm \sqrt{b}$ where a and b are rational. Comparison between two quantities of this form, which is the only exact arithmetic operation needed, is easily accomplished with a few rational operations.

DEFINITION 2.11. The (P, L)*-fringe* is that part of Vor(P) to the right of the (P, L)-beachline.[11]

See Figure 4. From the above lemma, the contour can meet Vor(P) only in this fringe. We first note that

LEMMA 2.12. *A fringe edge cannot meet the beachline twice*.

PROOF. See Figure 5. Suppose that f is a fringe edge, on the bisector of two sites p and r; suppose that both its endpoints u and v were on the beachline. These endpoints are equidistant from p, r, and L, and this fixes them uniquely on the bisector between p and r. Then f is (by definition) the edge joining u and v: but points between u and v are to the left of the beachline, hence not on the fringe, a contradiction. □

It is possible that an edge of Vor(P) could cross the contour twice; if so, the beachline divides it into two fringe edges. That is a consequence of the following lemma.

---

[11] The (Q, L)-fringe lies to the left of the (Q, L)-beachline.

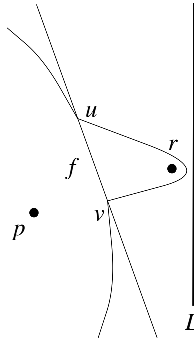

Fig. 5. Illustrating Lemma 2.12.

LEMMA 2.13. *Let X be a line-segment or ray with one end on the beachline at a point x and entirely contained in a cell of* $\mathrm{Vor}(P)$ *containing x. Then X cannot meet the contour more than once.*

PROOF. (See Figure 6.) Choose $p \in P$ so that X is in the cell of $\mathrm{Vor}(P)$ owned by p. We assume that X meets the contour at least once. Let y be the point on X, closest to x, where X meets the contour. Now, x is on the beachline: therefore x is not on the contour, and it is closer to p than to the closest site in Q (Lemma 2.9(iii)).

Let q be a site in Q such that y is equidistant from p and q. Since y is on the perpendicular bisector of the line pq, and x (being closer to p) is not, all points beyond y on X are closer to q than to p. Since they are all in the cell of $\mathrm{Vor}(P)$ owned by p, they are closer to the closest site in Q than in P, and are therefore to the right of the contour. Hence no point beyond y is on the contour. □

LEMMA 2.14. *A fringe is a forest of (free)*[12] *trees.*

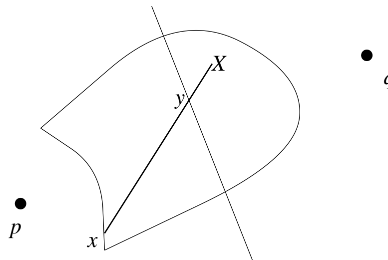

Fig. 6. Illustrating Lemma 2.13.

[12] A free tree is a connected undirected graph without cycles; it does not have a distinguished root node. In this paper "graph" has a wider meaning than usual, since edges can be unbounded.

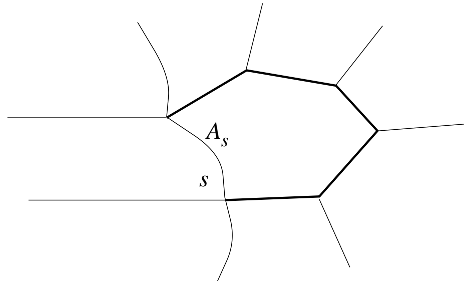

Fig. 7. A_s: beachline region owned by s (Definition 2.15). The open boundary bA_s is darkened.

PROOF. In other words, the (P, L)-fringe is an acyclic graph. The reason is as follows: any simple cycle of edges in Vor(P) must (being a Jordan curve[13]) enclose part of the plane. Its interior, being open, intersects the interior of the cell of at least one site p. However, the cycle is disjoint from the cell interior, so it encloses all of the cell, and hence encloses p. Since p is to the left of the beachline, the cycle cannot be entirely within the fringe. □

DEFINITION 2.15. Let s be a beachline segment, contained in the cell C of a site p in Vor(P). The *region* A_s owned by s is that part of C bounded by s on its left. The boundary of A_s (which includes s) is conventionally denoted ∂A_s. The open boundary[14] bA_s is that part of ∂A_s to the right of the beachline: $bA_s = \partial A_s \backslash s$.

See Figure 7. The open boundary of A_s is where A_s intersects the (P, L)-fringe.

2.16. It is easy to show that for every corner p of the convex hull $H(P)$ of P there exists a circle touching p and L and not touching or containing any other site in P or Q: in other words, each corner of the convex hull owns a beachline segment. The two infinite segments of the beachline belong to the parabola whose focus is the leftmost site in P (necessarily a corner of $H(P)$).

Consider one of the infinite edges of Vor(P). It lies along the perpendicular bisector B of two sites p, p' which are corners of $H(P)$. As a point z moves along B away from $H(P)$, the circle with centre z passing through p and p' increases in size; when z is sufficiently distant then that circle contains no other sites in P and intersects L (since the line pp' is not vertical by assumption 2.1). This implies that z is on the (P, L)-fringe. In other words (see Figure 8),

LEMMA 2.17. *Every unbounded edge of* Vor(P) *intersects the* (P, L)*-fringe, and the intersection is unbounded.*

[13] A Jordan curve is a curve in the plane topologically equivalent to a circle. The Jordan Curve theorem [16], [21] says that such a curve has a definite "inside" and "outside."
[14] "Open boundary" is a nonstandard term, used only in this paper.

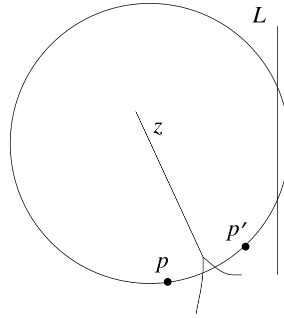

Fig. 8. Illustrating Lemma 2.17.

DEFINITION 2.18. A segment s is a *separating segment* if s is bounded but A_s is unbounded.

If s is a separating segment, then its two endpoints are on different, disjoint, unbounded components of bA_s:

LEMMA 2.19. *If s is a bounded nonseparating segment, then its endpoints are connected by bA_s. If it is a separating segment, then its endpoints are connected to different unbounded edges on the fringe.*

PROOF. Consider a point moving along bA_s, beginning at the lower endpoint of s. The moving point either goes to infinity, along an unbounded fringe edge, or returns to the beachline at the upper endpoint of s. In the first case A_s is unbounded. In the second case bA_s is bounded.

Thus if s is bounded (has two endpoints), but is not a separating segment, then its endpoints are connected along bA_s. If s is a separating segment, then its endpoints are not connected along bA_s, its lower endpoint is connected to an unbounded edge, and its upper endpoint is (by similar reasoning) connected to a different unbounded edge. □

LEMMA 2.20. *Each tree in the (P, L)-fringe contains exactly one unbounded edge from* $\text{Vor}(P)$.

PROOF. Let T be a tree in the (P, L)-fringe. Let h be the highest beachline cusp on T. The segment s whose lower endpoint is h cannot be a bounded nonseparating segment, since otherwise, by Lemma 2.19, its upper endpoint would also be in T. Therefore h is connected to an unbounded edge along bA_s. Therefore T contains at least one unbounded edge.

Next we show that every tree meets at most one unbounded edge; equivalently, there is no path within the (P, L)-fringe joining two unbounded edges.

Consider any simple path Π in $\text{Vor}(P)$ joining two infinite edges of $\text{Vor}(P)$. Let R be a large rectangular region containing all sites in P and all vertices in $\text{Vor}(P)$, so only the unbounded edges of the diagram intersect the boundary ∂R, and both L and the infinite

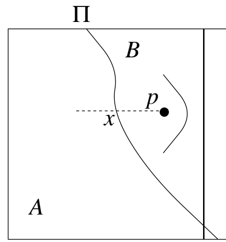

Fig. 9. Illustrating Lemma 2.20.

beachline segments pass through the top and/or bottom sides of R. Note that Π intersects ∂R at exactly two points. By a straightforward adaptation of the Jordan Curve Theorem, Π partitions R into two open connected sets A and B. See Figure 9.

Let x be a point on Π, not a vertex, so there are exactly two Voronoi cells incident to x, owned, say, by p and q, respectively. These sites are both inside R, and the line-segments px and qx are entirely within these respective cells, meeting Π only at x, so one of them is in A and the other is in B. This implies that both A and B contain sites from P.

Clearly, the outside of R is partitioned by Π into two connected regions, so we can say that Π divides the whole plane into two connected regions, still denoted A and B. We shall label A and B so that B contains a site p such that the horizontal ray extending leftwards from p intersects Π at a point x. Since the beachline must pass to the right of p, it will follow that x is not on the fringe, and the proof will be complete (the infinite edges belonging to different fringe trees).

If Π extends infinitely in both vertical directions, then let B be the region "to the right of" Π. Then Π passes to the left of all sites in B, and no more need be said. Otherwise, Π is bounded below, say, and unbounded above (by assumption 2.1, the infinite edges cannot be horizontal). Let B then be the region "above" Π. Then for every horizontal line ℓ, $B \cap \ell$ is bounded. Let p be a site in B, and let ℓ be the horizontal line through p. Let x be the leftmost point of $B \cap \ell$; then x is on Π and to the left of p. □

2.21. Let T be a free tree—a connected acyclic graph—in the (P, L)-fringe. Let E be its unique unbounded edge, with r its endpoint. Orient all edges of T by orienting E away from r and orienting all other edges towards r (i.e., any edge e is connected to r by a unique path in T: if $e \neq E$, then it is oriented towards r along this path). Regarding these edge orientations as from child to parent, T (or, more properly, $T \backslash E$) is now given the structure of a binary tree: the leaves of T are the cusps where T meets the beachline. Every internal node v of T has two incoming edges and one outgoing edge. Let e_3 be the outgoing edge. Then e_3 is oriented towards the parent of v (unless $v = r$). Let C_1 be the cell of $\mathrm{Vor}(P)$ to the left of e_3, and let C_2 be the other cell; the other two edges meeting at v are on the boundary of these cells; let e_1 be the edge on the boundary of C_1; similarly, e_2. Then the child node on e_1 (resp. e_2) is defined as the *left* (resp. right) child of v. See Figure 10.

LEMMA 2.22. *Let e be a fringe edge, on a fringe tree T. Let p_1* (*resp. p_2*) *be the site owning the cell to the left* (*resp. right*) *of e. Let x be a point on the interior of e. Let x_1* (*resp. x_2*) *be the unique point where the line-segment xp_1* (*resp. xp_2*) *crosses the*

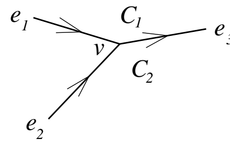

Fig. 10. Illustrating 2.21.

(P, L)-beachline. The beachline interval from x_1 to x_2 and the line-segments xx_1 and xx_2 together form a Jordan curve J. Suppose that v is the endpoint of e inside J (or on J if e meets the beachline). See Figure 11. Then:

(i) *All descendants of v in T are inside J and all nodes inside J are descendants of v.*
(ii) *x_1 is above x_2 on the beachline.*

PROOF. Let u be a node inside J. The path from u to infinity in the fringe must cross J, at some point not on the beachline; the only such point is x, and the path must therefore pass through v, so u is a descendant of v. If u is a node of the fringe not inside J, then the path from u to infinity cannot cross J, since if it entered J it would have to leave it at a different point, but it can only cross J at x. Therefore u is not a descendant of v. This proves (i).

(ii) Let y be a point beyond x on e, so the line-segment xy is oriented towards the infinite part of T. Travel around the Jordan curve J, beginning at x_1: from x_1 to x to x_2 and along the beachline back to x_1. At x, y is to the left of this path. If x_2 were above x_1, then this tour would be anticlockwise, with the interior of J on the left. In this case, y would be inside J. However, in this case a path from y to infinity in T would be forced to leave J, and therefore cross x, which is impossible. □

COROLLARY 2.23. *Let e be a fringe edge, with p_1 and p_2 its two adjacent sites. Then the orientation of e in the fringe depends only on p_1, p_2, and L.*

PROOF. Choose any internal point x on e, and calculate x_1 and x_2 as in Lemma 2.22. Assume that x_1 is the higher of the two points. Then e is oriented so that x_1 (and p_1) is to the left of e. □

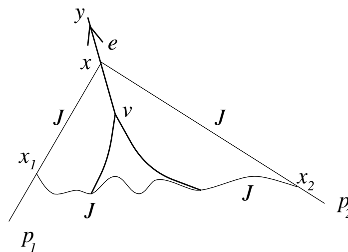

Fig. 11. Illustrating Lemma 2.22. The x-direction is upwards.

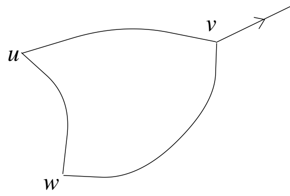

Fig. 12. Illustrating Lemma 2.25.

COROLLARY 2.24. *For any fringe node v, the leaf descendants of v form a contiguous interval of beachline cusps, with the leftmost descendant highest and the rightmost lowest.*

PROOF. From Lemma 2.22(i), and following the notation of that lemma, the leaf descendants are between x_1 and x_2 on the beachline; hence they form a contiguous interval. The leftmost leaf descendant is the closest cusp to x_1 in J: hence it is the highest cusp. Similarly, the rightmost cusp is the lowest. □

Consider a segment s of the beachline between these two cusps. It is a bounded nonseparating segment (Definition 2.18): let u and w be its upper and lower endpoints. Recall (Lemma 2.19) that bA_s connects u to w in T.

Now for any tree T a path joining two (adjacent) leaves u and w can be separated into two branches leading from the leaves to their lowest common ancestor v. In the present case, the branch from u to v (clockwise along bA_s) leads to a left child of v along a rightmost branch, and the branch from w to v (anticlockwise along bA_s) leads to a right child of v along a leftmost branch. See Figure 12. This implies that u and w are the inorder predecessor and successor, respectively, of v:

LEMMA 2.25. *If u and w are the endpoints of a nonseparating segment, so they are adjacent beachline cusps, and v is their lowest common ancestor, then u is the inorder predecessor and w is the successor of v.*

Another lemma with the same flavour as Lemma 2.22 is about edge orientations at the contour. It refers to the construction in that lemma.

LEMMA 2.26. *In Lemma* 2.22 *suppose that x is not to the right of the (P, Q)-contour (it might be on the contour); let v be as in the lemma. Then v and all its descendants are to the left of the contour.*

PROOF. For clarity, assume x is on the contour. By Lemma 2.13, the line-segments x_1x and x_2x meet the contour only at x, so the Jordan curve J meets the contour only at x. However, the beachline is to the left of the contour; hence so is J; hence so is v and all its descendants. □

Both the contour and the beachline are monotonic in the y-direction. The order of edges (resp. segments) is not necessarily the same as the vertical order of the sites owning

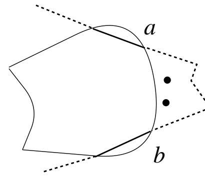

Fig. 13. Illustrating Lemma 2.27(i).

them. However, there is a way to match contour edges with some beachline segments so that vertical orders correspond. We call an edge which meets the contour an *attachment*.

In the following lemma we speak of beachline "features." This means "segments or cusps." If e is a fringe edge, its "child" endpoint v is either a beachline cusp, or comes between two adjacent beachline cusps u and w in inorder (Lemma 2.25). In the first case we consider the beachline cusp to "own" v and in the second case we consider the beachline segment between u and w to "own" v.

LEMMA 2.27.

(i) *For any beachline segment* s, A_s (2.15) *intersects the contour in a connected interval.*

(ii) *Let* A *be the set of attachments, let* U *be the sequence of beachline features owning the child endpoints of the edges in* A, *and let* V *be the sequence of contour vertices where they cross the contour. Then* U *and* V *are in the same vertical order.*

(iii) *Let* e *be a contour attachment, meeting the contour at a vertex* v, *and let* i_1 *and* i_2 *be the contour edges incident to* v, *with* i_1 *above* i_2. *Let* s_1 *and* s_2 *be the* (P, L)-*beachline segments such that* i_j *intersects* A_{s_j}, $j = 1, 2$. *Then* s_1 *is above* s_2.

PROOF. (i) If the intersection were disconnected, then there would be two points a, b common to the contour and the open boundary bA_s, with a above b and no other such points between. See Figure 13. The paths from a to b on bA_s and on the contour would form between them a simple cycle of edges in Vor(S), a Jordan curve, containing sites from S and lying between the (P, L)-beachline and the (P, Q) contour. Hence these sites are from P and are to the right of the beachline, which is impossible.

(ii) Let a and b be two points on the contour, where edges e_1 and e_2 respectively of the (P, L)-fringe meet the contour, with a above b and no other such point in between. Then the edges (truncated to a and b, respectively) and the contour interval between them are all on the boundary of the same cell, C, of Vor(S).

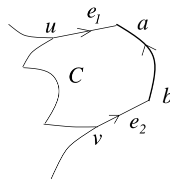

Fig. 14. Illustrating Lemma 2.27(ii).

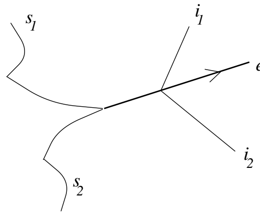

Fig. 15. Illustrating Lemma 2.27(iii).

Let u be the child vertex on e_1, v that on e_2 (Figure 14). The path $vbau$ (from v to b along e_2, then from b to a along the contour, and then from a to u) is in the anticlockwise sense around the boundary ∂C. So C is to the left of e_2 and to the right of e_1; the rightmost cusp descendant of u and the leftmost descendant of v are upper and lower ends respectively of a beachline segment in C; so the beachline feature owning u is above that owning v, as required.

(iii) See Figure 15. The attachment e crosses the contour from left to right; i_1 is left of e and i_2 is right of e; therefore s_1 and s_2 are respectively left and right of e, which implies that s_1 is above s_2. □

Summary. In this section the structures of the beachline and fringe have been analysed in detail. The results, summarized, are: each unbounded edge of Vor(P) separates the cells of sites which are adjacent corners of the convex hull $H(P)$. For every such edge there is an unbounded edge on the (P, L)-fringe.[15] The (P, L)-fringe is a disjoint union of free trees; every such tree contains a unique unbounded edge. These trees can be oriented naturally as binary trees, with the leaves on the beachline: with this orientation, the leftmost descendant of a node is the highest[16] on the beachline.

Since the fringe has quite a regular structure, it partitions the plane into regions which are well organized for planar point-location queries; this organization is discussed in detail in Section 4, leading to efficient calculation of the contour in Section 5.

3. Parallel Operations on Arrays. This section collects some standard techniques of parallel computation. Most of them are of the $\sqrt{n}$ divide-and-conquer style, leading to various $O(\log\log(n))$ parallel time algorithms.

3.1. We begin with a review of Valiant's CREW procedure to merge two sorted arrays A and B of sizes n and m respectively into another array C of size $n + m$. We assume there is a processor attached to each entry in A and B. For every item $A[i]$ we want to calculate the maximum index j (if any) such that $A[i] > B[j]$; for every item $B[j]$ we want to calculate the maximum index i (if any) such that $A[i] \leq B[j]$.

Let A be divided into $\sqrt{n}$ blocks of size $\sqrt{n}$. Correspondingly, B is divided into $\sqrt{n}$

[15] There are two such edges if P contains only two sites.

[16] Assuming P is to the left of L; otherwise the opposite holds.

(uneven) intervals, and each block of A is merged recursively with the corresponding interval from B, as follows.

Call the first elements in each block the *block leaders*. Let $n_i = 1 + (i-1)\sqrt{n}$: then $A[n_i \cdots n_{i+1} - 1]$ is the ith block of A. The corresponding interval $B[m_i \cdots m_{i+1} - 1]$ (which can be empty) satisfies: $B[m_i]$ is the first element no less than $A[n_i]$, if it exists. Calculating the partition of B is in two steps. First, the elements $B[m_i]$ matching the block leaders are identified; these elements define the partition of B. Second, all elements in $B[m_i \cdots m_{i+1} - 1]$ matching the ith block of A are informed.

Let B be divided into $m/\sqrt{n}$ blocks each of size $\sqrt{n}$. For each block leader $A[n_i]$ in A assign one processor from each block of B to compare the block leader, $B[\ell]$, say, with $A[n_i]$ and hence identify the largest block leader $B[\ell]$ of B (if any) such that $B[\ell] < A[n_i]$. Then the processors in $A[n_i \cdots n_{i+1} - 1]$ can inspect the block $B[\ell \cdots \ell + \sqrt{n} - 1]$ to ascertain the largest i' such that $B[i' - 1] < A[i]$: i' is the desired index m_i. This finishes the first step.

Next the correct indices i are first written to the block leaders of B as follows: if $B[\ell]$ is a block leader, then the processors in the block identify the corresponding block i of A, by finding the largest n_i such that $A[n_i] \leq B[\ell]$.

To clarify the second step, we define

DEFINITION 3.2. A block of processors in A is *short* if the corresponding interval of B is within a single block of B, otherwise it is *long*.

(Compare with 5.6.) Let J be the ith block of A, and let I be the corresponding interval $m_i \cdots m_{i+1} - 1$ of B. If J is short, then $|J| \geq |I|$ and the index i can be written into I by the processors in J. If J is long, the processors in J first write the index i to the leftmost and rightmost blocks of B intersecting I. Then the processors in I complete the calculation; for the only entries in I for which i is not yet calculated are those belonging to complete blocks contained in I, and they can copy i from their block leaders. This finishes the second step.

For a complete description, which does not assume $\log\log(n)$ is an integer, and which achieves optimal speedup ($n/\log\log(n)$ processors), see [18].

3.3. The following "broadcasting" problem occurs in the Voronoi diagram algorithm. Suppose that A is an array of bits with one processor attached to each array entry. Suppose that each 1-bit is intended to mark the beginning of a subinterval of the array, so we call the 1-bits the "interval leaders." The problem is to inform all the other processors of the nearest interval leaders, i.e., for each i, the maximum $j \leq i$ which carries a 1-bit.

An obvious way is to use parallel prefix. This would be a bottleneck in our algorithm [23]. When the interval leaders are linked together, however, it is possible to do this in $O(\log\log(n))$ time, as shown below.

LEMMA 3.4. *With A as above, and assuming that each interval leader knows the closest interval leader to its left and right, all of A can be informed of the appropriate interval leaders in $O(\log\log(n))$ parallel steps (CREW).*

PROOF. Without loss of generality n is a power of 2; let $n = 2^{k_1+k_2}$ where $k_1 \leq k_2 \leq k_1 + 1$. Consider A partitioned into 2^{k_1} blocks of size 2^{k_2}.

We have the "marked" processors, the interval leaders, and we have the processors at the beginning of each block, the block leaders. Each interval leader ascertains if there are any block leaders in its interval. If so, it informs the *leftmost* such block leader. All the processors in this block can consult the block leader, and then the interval leader, to ascertain the range of block leaders in the interval, and hence to inform them all (there are enough processors to do this).

Having done this, all processors in the same interval as their block leader or the leader of the *next* block can identify themselves by consulting the block leader. The only processors which remain in doubt are those in intervals which do not contain a block leader, and these can be informed by a recursive application of the same method within the blocks.

In summary: there are about $\log\log(n)$ rounds. In the ith round, B is partitioned into 2^{2^i} blocks of equal size. For each one of these blocks, if it is entirely contained in an interval of the partition, then the identity of the interval leader is written into each block entry. If a block does not fit into an interval of the partition in the ith round, then it is untouched during that round. □

Lemma 3.4 addresses the so-called *forward chaining problem* [6], in the special case where there are pointer-links between the marked elements. Actually, the same effect can be achieved by a subtle variation of the methods in the above two lemmas, on a CRCW machine, without assuming such links. We state the result (it is used in Sections 7 and 8):

PROPOSITION 3.5 [6]. *If A is an array of processors, some of them marked, in a CRCW machine, then in time $O(\log\log(n))$ the processors can attach to each array entry $A[j]$ indices i and k to the closest marked entries (if they exist) left and right of j.*

We also use the principal result from [6] in Section 8:

PROPOSITION 3.6. *Let K be an array of $k \leq n$ integers between 1 and n. Then K can be stably sorted in $O(\log(n)\log\log(n))$ time using $k/\log(n)$ processors (CRCW).*

PROOF. Theorem 1 of [6] provides for sorting K stably in time $t(k, n) = O(\log(k)/\log\log(k) + \log\log(n))$ with a time-processor product of $O(k \log\log(n))$. Using $k/\log(n)$ processors, the time would be $O(\log(n)\log\log(n))$, which is greater than $t(k, n)$, so the time is achievable. □

3.7. For completeness we include two other techniques: list-ranking and parallel prefix. These are applied in Sections 7 and 8. The techniques described here are inefficient in terms of processor allocation, but they are sufficient for our purposes.

Given a linked list F, the rank of a node in F is its distance from the last node in the list. Let $k = |F|$. If processors are assigned to every node in the list, this rank can be calculated in time $O(\log(k))$ as follows: there are about $\log_2(k)$ phases; in the $(r + 1)$st phase all nodes of rank less than 2^r know their rank, and the other nodes know the node at distance 2^r ahead of them in the list. A typical node p with a node q at distance 2^r

ahead on the list calculates the node at distance 2^{r+1} by pointer jumping, and, if this does not exist, stores its rank as $2^r + d$ where d is the rank of q.

This (exclusive write) algorithm will be sufficient for our purposes; $O(\log(k))$ is optimal for exclusive-write machines, as can be shown by a reduction from parity [13].

Parallel prefix is the recognized method of solving the following problem: given a list $x_1, \ldots, x_k$ of numbers stored in an array, compute all partial sums $\sum_1^r x_i$. It can be solved with $k/\log(k)$ processors (CREW), essentially by covering the array with a balanced tree structure and calculating partial sums of subintervals covered by nodes of the tree [19]. The runtime of $\log(k)$ achieved is more or less optimal [23].

4. Point Location Using the (P, L)-Fringe. From Section 2 we have a fairly complete picture of the (P, L)-fringe: it is a forest of free trees, exactly one tree for each unbounded edge of $\text{Vor}(P)$; the leaves of each tree form a contiguous interval of cusps along the (P, L)-beachline, and it has essentially the structure of a binary tree in which the leaf descendants of any node likewise forms a contiguous interval along the beachline, with the "leftmost" above the "rightmost" with respect to y-coordinates.

Throughout this section, n will be the number of sites in P. The beachline, fringe, etc., and the arrays covering these structures, all have size $O(n)$. It is convenient to speak as if n is the number of entries in an array covering the (P, L)-beachline.

4.1. We address the following *location* problem: given a point q in the plane, to decide whether it lies to the right of the (P, L)-beachline, and, if so, to return the cell of $\text{Vor}(P)$ containing q—that is, to return the site in P closest to q.

The aim, realized in Section 5, is to use search structures on the (P, L)- and (Q, L)-fringes to calculate the (P, Q)-contour vertices. We use methods analogous to parallel merging (3.1). Valiant's parallel algorithm to merge sorted arrays A and B is based on the observation that any item x can be located in the array A in two steps using $\sqrt{n}$ processors. Analogously, we describe a "skeleton tree" for the (P, L)-fringe which enables any point x to be located in the fringe region containing x using $\sqrt{n}$ processors. The skeleton tree will have size $\sqrt{n}$ and each node of the tree will be assigned a plane region intersecting $O(\sqrt{n})$ regions of the (P, L)-fringe.

4.2. Since the individual trees in the fringe correspond to sides of the convex hull $H(P)$, it is useful to have a description of this set (i.e., an array containing its corners in cyclic order) available during the algorithm. Fortunately it is not difficult to construct $H(S)$ from $H(P)$ and $H(Q)$ with n processors: it is necessary to calculate the two outer tangents common to $H(P)$ and $H(Q)$, and this can be done in constant parallel time using $\sqrt{n}$ divide-and-conquer [28], [4], [22]. Thus we have all relevant convex hulls constructed during the algorithm.

4.3. The fringe is laid out as a sequence of binary trees. To provide a uniform search-structure, it is convenient to embed the structure in a *single* full binary tree T. The structure of T will itself be mapped onto the beachline.

Let $T_1, \ldots, T_k$ be the sequence of fringe trees, with root nodes (meeting the infinite edges) $r_1, \ldots, r_k$. Let $s_1, \ldots, s_{k-1}$ be the sequence of beachline segments whose bounding cusps are on different trees (s_1 separates T_1 from T_2 and so on; we ignore the two unbounded segments owned by the leftmost site in $H(P)$).

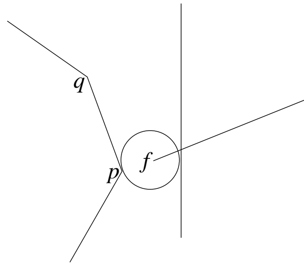

Fig. 16. Ray from f is in p's cell in Vor(P) (2.18).

These segments s_i are the separating segments (Definition 2.18). For the purpose of searching the beachline, it is useful to have sample points available on the separating segments. Such points can be calculated easily from $H(P)$; for definiteness, we fix the following method of calculating separating points: given a site p which is a corner of $H(P)$ but not the leftmost corner, let q be the corner next to p in anticlockwise order, and let f be the centre of the circle tangent to L and tangent to the side pq at p. Then f is on the separating segment associated with p, and the infinite ray extending outwards from f, in the direction pf, is entirely within the (P, L)-fringe, and entirely within p's cell in Vor(P). (See Figure 16.) It is convenient to fix and record descriptions of these rays and call them *separating rays*. There is one separating ray for each corner of $H(P)$ except the leftmost.

One associates nodes n_i corresponding to the $k-1$ separating segments $s_1, \ldots, s_{k-1}$; these are not associated with any vertex of the fringe, but, formally, the left child of n_i is r_i and its right child is n_{i+1} (except for n_{k-1} whose right child is r_k).

This defines a structure T which is a single full binary tree. The leaves of T correspond to the beachline cusps; the internal nodes correspond to the bounded beachline segments. The beachline segments (excluding the two unbounded segments) each "own" a unique internal node of T, namely, the least common ancestor (LCA) in T of its two bounding cusps. If s is a separating segment (2.18), then this LCA is an artificial node n_i introduced above. If s is a bounded nonseparating segment, then this LCA corresponds to a (P, L)-fringe vertex, using the natural binary-tree structure on the fringe (2.21). Sufficient information about the (P, L)-fringe to define the structure of T will be stored in an array A covering the (P, L)-beachline (4.7).

DEFINITION 4.4. If s is a bounded nonseparating segment,[17] the unique vertex v coming between its bounding cusps in inorder (Lemma 2.25) is called the *inner vertex* owned by s.

4.5. The regions A_s as defined in 2.15 partition the plane to the right of the (P, L)-beachline. By extending the ends of the open boundaries bA_s leftwards with infinite horizontal rays, we define a partition of the entire plane. This partition is subdivided as

[17] Vertices of Vor(P) are not associated with the separating segments, which separate different trees of the fringe.

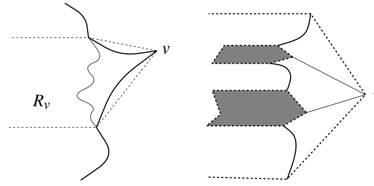

Fig. 17. Tube R_v; T-pocket at v.

follows. With each inner node v of T is associated a *tube*: this is a polygonal region, not necessarily convex, defined as follows. Let u and w be v's leftmost and rightmost leaf descendants in T. If u and w are in the same tree of the fringe, so v is a fringe vertex, then the tube is bounded by the lines uv and vw (possibly but not necessarily fringe edges), and the horizontal rays extending leftwards from u and w. See Figure 17.

Otherwise, u and w are the highest and lowest cusps respectively meeting two fringe trees T_i and T_j. If u is not the highest beachline cusp, then let the tube be bounded above by the separating ray directly above u (4.3), extended leftwards from where it meets the beachline by an infinite horizontal ray. If u is the highest cusp, then the tube is unbounded above. Similarly, the tube is bounded below by two infinite rays unless w is the lowest cusp in which case the region is unbounded below.

All tubes are closed, that is, they include their polygonal boundaries.

4.6. We assume there is an array A covering the (P, L)-beachline. Recall from the Introduction that this means that each array entry $A[i]$ contains, or points to, a data record $R(f)$ associated with one of the features f (a cusp or segment) of the beachline, and that all beachline features are thus accessed in vertically descending order in A, possibly with duplicate entries. We allow for duplicates to avoid a data-compression step between phases of the algorithm.

In general, the number of processors available will be a fixed fraction of the size of A. For simplicity, we assume that there are sufficiently many processors available to attach to all the elements of A.

4.7. Array entry $A[j]$ contains the following information about the beachline feature f which it covers.

- The interval $i \cdots k$ of entries in A covering f.

If $j \neq i$, then no other data need be stored with this entry: it is enough to store the information about f in $A[i]$.

If the feature f is a cusp, then:

- Its coordinates are stored in $A[i]$.
- Let e denote the edge of $\mathrm{Vor}(P)$ crossing the beachline at the cusp: a pointer to a record for e is stored with f. This is needed when combining $\mathrm{Vor}(P)$ with $\mathrm{Vor}(Q)$ (Section 6).

Otherwise, it is a segment. The two unbounded segments contribute little information. If the feature is a bounded segment s, let u and w be the bounding cusps, and let v be

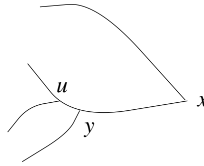

Fig. 18. Illustrating Lemma 4.9.

the inner vertex owned by s;[18] then the information stored is:

- The site p owning it, the coordinates of v if s is not a separating segment (2.18); if it is, a description of the separating ray. (If s is a separating segment, then the site p is a corner of the convex hull $H(P)$, and the index i of the leftmost entry in A covering s will be associated with p in the description of $H(P)$.)
- Indices to the (leftmost) records covering the leftmost and rightmost cusp descendants of v, from which the tube owned by s is easily determined.

DEFINITION 4.8. (See Figure 17.) The T*-pocket* associated with an internal node v is the set $R_v \backslash (R_u \cup R_w)$, where R_v is the tube associated with v and u and w are its two children in T.

LEMMA 4.9. *Pointer links defining the parent–child relations of T can be installed in the array A in constant time with n processors. Then the T-pocket associated with every node can be determined in constant time.*

PROOF. We assume one processor per entry in A. Since the range of leaf descendants is given with each node, it is trivial to test whether one node is an ancestor of another. Given a node u, let x be the inorder predecessor of its leftmost descendant, and let y be the inorder successor of its rightmost descendant. If x does not exist or is an ancestor of y, then u is a left child with parent y, otherwise u is a right child with parent x. See Figure 18. Thus with one processor assigned to each entry of A it can be decided quickly of every node whether it is a left or right child and which node is its parent. Then the parent–child pointer can be installed in the parent's record.

Since the tube associated with each node can be easily constructed, the T-pocket can then be determined. □

4.10. A pocket has at most twelve edges, including its infinite horizontal edges, it meets at most three cells of $\mathrm{Vor}(P)$ to the right of the beachline, and hence it meets at most three beachline segments. Hence, given a query point x, a single processor can decide easily whether x is in the pocket, and if so, whether x is to the right of the beachline, and if so, which cell of $\mathrm{Vor}(P)$ contains x. Assigning processors to all the entries in A, it is therefore easy to determine whether x is to the right of the (P, L)-beachline, and if so, which site in P is closest.

[18] Therefore $A[i-1]$ and $A[k+1]$ contain records for u and w, respectively.

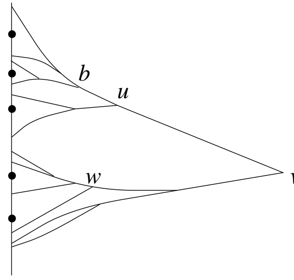

Fig. 19. Skeleton tree: b, u, v, w splitting nodes, b, w bottom. u has one child b in T', v has children u and w in T'.

4.11. An important sampling technique will allow $\sqrt{n}$ processors to solve location problems in bounded parallel time. The sampling involves taking a *skeleton tree* T' whose nodes are nodes of T and which have the same ancestor relation as T.

Let k be a proper divisor of n (for convenience we assume that n is a power of 2 (2.1)). A *k-sample* of T is the subsequence of leaves obtained by taking every kth element $A[i]$ indexing a feature f, and choosing either f if it is a cusp or its lower cusp if it is a segment. We call the leaves thus sampled the *marked* leaves of T.[19] The marked leaves form a subsequence such that between any two leaves in the sequence (in inorder) there are $O(k)$ nodes of T. The *span* of an internal node v is its set of marked leaf descendants: this is easily calculated from the indexes i and j of the leftmost and rightmost entries in A covering the descendants of v.

An internal node v is a *splitting node* if its two children both have nonempty span. It is a *bottom node* if it is but none of its descendants is a splitting node. Notice that every bottom node has exactly two marked descendants. See Figure 19. There is a binary tree T' whose nodes are the splitting nodes of T, and whose leaves are the bottom nodes of T. The parent of a node in T' is its closest splitting-node ancestor in T. A splitting node which is not a bottom node can have one or two children in T'.

If v is a splitting node, its T'-pocket is defined as $R_v \backslash (R_u \cup R_w)$ where u and w are its children in T', or $R_v \backslash R_u$ if v has just one child u in T'; if v is a bottom node, its T' pocket is simply the tube R_v. The highest splitting node r' in T'—which need not be the root of T—is the root of T': the complement of $R_{r'}$ defines one other kind of pocket, the *enclosing pocket*.

LEMMA 4.12.

(i) *T' is a binary tree with $O(n/k)$ nodes.*
(ii) *Each pocket contains $O(k)$ vertices of the (P, L)-fringe.*
(iii) *Using the covering array A, n processors can identify the nodes of T' and the parent–child relations in $O(1)$ parallel time.*

---

[19] The array A covers the beachline; a leaf can be marked because several of the sampled elements of A are associated with this leaf; if $A[i \cdots j]$ is the subinterval of A associated with the feature, then $A[i]$ is where the marking information should be written; this information can be calculated by the processor attached to $A[i]$, the others remaining idle.

(iv) *The vertices and cusps in the* (P, L)*-fringe can be mapped to the pockets containing them in* $O(1)$ *parallel time.*
(v) *When*[20] $k = \sqrt{n}$, *the location problem for any query point* q *can be solved by* k *processors in* $O(1)$ *parallel time.*

PROOF. (i) We show that it is a binary tree in the sense that every node in T' has at most two children in T'. The notion of "left" and "right" child is irrelevant where a node has just one child; otherwise, the children are independent nodes of T, and it is natural to order them the same way in T'.

Suppose that v has two different children u and w in T'. Then v is their closest ancestor in T which is a splitting node, so they are independent and v is their lowest common ancestor (LCA). It follows readily that v has no other children in T': If x were another splitting-node descendant of v, then say without loss of generality that both w and x were descendants of the right child of v in T: then their LCA y would also be a descendant of the right child, so it would differ from v, and hence x could not be independent of w (y is not splitting), so x would be a descendant of w.

Since every bottom node has two marked descendants in T, there are at most $\ell/2k$ bottom nodes, where $\ell < n$ is the number of beachline cusps. If T' were a full binary tree it would follow that it had fewer than n/k nodes. However, it might not be. Consider the full binary tree T'' obtained from T' as follows: if v is a splitting node, not a bottom node, with just one child u in T', then one of its children in T, say w, has exactly one marked descendant, and u is descended from the other child. Add w as a leaf of T'', so u and w are siblings in T''. Then in T'' every leaf has at least one marked descendant, so T'' has at most n/k leaves, and T'' is a full binary tree, so it, and therefore T', has at most $2n/k$ nodes.

(ii) Let v be a splitting node. If it has two children u and w in T', then let T'' be the subtree of T at v, with the subtrees at u and w deleted. Then the leaves of T'' come in three intervals along the beachline, each containing no marked cusp, so it has at most $3k - 3$ leaves, and hence, since it is (nearly) a complete binary tree, fewer than $6k$ nodes altogether. However, these are all the nodes in the T'-pocket at v.

If v has just one child u in T', let T'' be the subtree of T at v with the subtree at u deleted. Then T'' is (nearly) a complete binary tree, and its leaves are among two intervals of beachline cusps. These cusps contain exactly one marked cusp, so there are again at most $6k$ nodes in T'', hence in the pocket at v.

If v has no children in T', then it is a bottom node, and its leaf cusps form an interval containing exactly two marked cusps, and again the pocket at v, which equals the tube at v, contains at most $6k$ nodes of T.

Finally, if v is the highest splitting node, let T'' be T with the subtree at v deleted: then T'' is (nearly) a complete binary tree whose leaves form two intervals along the beachline and which has no marked leaf descendants, so it has at most $4k$ nodes. These are all the nodes of T in the enclosing pocket.

(iii) Let l_1 and l_2 be two adjacent marked leaves of T. This means that the range of

[20] Or $\sqrt{n/2}$, whichever is an integer.

entries of A covering either l_1 (resp. l_2) or else its inorder predecessor contains an index r (resp. $r + k$), where $r \equiv 1$ modulo k, and l_1 and l_2 are distinct.

Their LCA v in T must be a splitting node, since both its children possess marked descendants. Furthermore, if v is a splitting node, then taking as l_1 the rightmost marked leaf descendant of its left child, and as l_2 the leftmost marked leaf descendant of its right child, these are contiguous marked leaves and v is their LCA. Thus the splitting nodes are exactly the LCAs of adjacent marked leaves.

Suppose that $r \cdots s - 1$ is an interval of length k where $r \equiv s \equiv 1$ modulo k. There are leaves l_1 and l_2, possibly the same leaf, which are marked because r (resp. s) is in the interval covering them or covering the segment above them. If these are distinct, then they are adjacent marked beachline leaves, and the processors allocated to the interval can detect this, and since the LCA lies between these marked leaves in inorder, one of the processors can identify it and attach the information to the array entry covering l_1.

Note that if l_1 is a marked leaf (not the last), then a single processor can access the nearest marked leaf l_2 following it: it calculates the largest $r \equiv 1$ (modulo k) in the range covering l_1, and accesses l_2 by inspecting $A[r + k]$.

If u and v are the LCAs of adjacent pairs l_1, l_2 and l_2, l_3 of marked leaves, then either u is an ancestor of v in T, and v is its right child in T', or u is the left child of v in T'. This concludes (iii).

(iv) Each splitting node v is associated with a pair of adjacent marked leaves, hence with a block of at least k processors. Since its pocket contains $O(k)$ nodes, forming at most three intervals of contiguous nodes of T (in inorder), the nodes in v's pocket can be labelled (with the index of the leftmost entry in A covering the segment whose inner vertex is v) in bounded parallel time.

Each bottom node is associated with a unique pair of adjacent marked leaves, so it is easy to assign a block of k processors to label the interval of nodes in its pocket.

(v) Given the query point q, the processors are first distributed over the nodes of T', and q is located to the correct pocket of T'. The processors are then distributed to the nodes within the pocket, and q located in the correct pocket of T. □

5. Constructing the (P, Q)-Contour by Progressive Refinement.

We have seen that the fringe can be separated into pockets so that planar point location can be executed in constant time using $\sqrt{n}$ processors. Here we discuss the progressive subdivisions of the fringe obtained by sampling at progressively smaller intervals. By iterating the process $\log\log(n)$ times all the contour vertices are constructed. Here n is the number of sites in $P \cup Q$; the arrays covering the beachlines have size $O(n)$; for clarity we assume they have size n.

5.1. The interval lengths involved will progress through the sequence $k_0, k_1, \ldots$ where $k_0 = n$ (assumed a power of 2), and, for each i, k_{i+1} is either $\sqrt{k_i}$ or $\sqrt{k_i/2}$, whichever is an integer, until the value 1 is reached. Thus there are about $\log\log(n)$ refinement steps.

LEMMA 5.2. *No pocket edge crosses the contour more than once.*

PROOF. The horizontal pocket edges (those to the left of the beachline) do not cross the contour; the other pocket edges satisfy the hypothesis of Lemma 2.13. □

Any pocket has from two to six edges to the right of the beachline. Since the contour is to the right of the beachline, it cannot cross the other pocket edges. Hence the boundary of each pocket meets the contour at most six times and therefore the pocket meets the contour in at most three connected intervals which we call *contour fragments*.

The data-structure described in the previous section can be used to calculate the points where pocket edges cross the contour, but first the pocket edges which cross the contour are identified. In view of Lemma 5.2, for bounded edges it is enough to establish that their endpoints are on opposite sides of the contour. For the unbounded edges (the separating rays, 2.18), with only one endpoint, the following method is used.

LEMMA 5.3. *Let R be one of the separating rays associated with the beachline, so it is entirely within a cell of* $\mathrm{Vor}(P)$. *Then it can easily be determined whether R crosses the contour.*

PROOF. Let e be the side of $H(P)$ perpendicular to R; then R crosses the contour if and only if e is *not* an edge of $H(P \cup Q)$, which can be assumed to be already constructed (4.2). □

LEMMA 5.4. *The contour fragments (at the first stage of the construction) can be defined in constant parallel time.*

PROOF. We consider the first stage, where a $\sqrt{n}$-sampling is used. Assign $\sqrt{n}$ processors to each of the $O(\sqrt{n})$ pocket edges on the (Q, L)-fringe. We have seen how those unbounded edges which cross the contour can be identified; a bounded pocket edge crosses the contour if and only if its two ends lie on opposite sides of the contour, which can be decided in constant time (4.1; Lemma 4.12).

The problem reduces to using $\sqrt{n}$ processors to compute for a (possibly unbounded) line-segment X, known to cross the contour, and entirely contained in the cell of a site q in $\mathrm{Vor}(Q)$, the point c where it crosses the contour.

Assigning the processors to the (P, L)-fringe, the intervals of intersection (at most three) of X with each pocket are determined. Either X is unbounded and the last, unbounded, interval begins closer to q than P, or there is an interval with one end closer to q and the other to P—this is the interval containing c. It can then be computed by reassigning the processors to the fringe edges within the pocket and determining in the same way the cell of $\mathrm{Vor}(P)$ containing c, and hence calculating c.

Thus the endpoints of all the contour fragments on the (Q, L)-fringe are determined. Similarly for the (P, L)-fringe. □

We now consider the process of refining the samplings to compute ultimately all the contour vertices. Consider a refinement step. We have had (implicitly) a skeleton tree T' obtained by sampling every k_ith element of A. At the next level of refinement there is a skeleton tree T'' where every k_{i+1}th entry in A is sampled. Recall that the pockets are defined in terms of the "tubes" R_u associated with the nodes of the various skeleton trees. These tubes are fixed in relation to the original fringe tree T and have nothing to do with the samplings.

LEMMA 5.5. *Assuming that the tree T' is nonempty*:

(i) *Every marked cusp at one level will be marked at the next.*
(ii) *Every node in T' is a node in T''.*
(iii) *The T''-pockets subdivide the T'-pockets.*

PROOF. (i) Essentially trivial, since k_{i+1} divides k_i. (ii) Therefore every splitting node at the ith level is a splitting node at the $(i+1)$st.

(iii) Given any nodes x, y of T, x is a descendant of y if and only if $R_x \subseteq R_y$. The root node of T' is a descendant of that of T'', hence the enclosing pocket for T' contains that for T''. Let v'' be a node of T'', and let K denote the T''-pocket at v'', not the enclosing pocket if v'' is the root of T''. If no node in T' is an ancestor of v'' (including v''), then the root node of T' is a proper descendant of v'', hence a descendant of one of its (one or two) children in T'': hence K is contained in the enclosing pocket for T'. If v'' has some ancestor in T', let v' be the lowest such ancestor. Let u' be a child of v' in T': u' need not exist, but if it does, then it cannot be an ancestor of v'' in T' (being lower than v'). If u' and v'' are independent, then $R_{v''} \cap R_{u'} = \emptyset$, so $K \cap R_{u'} = \emptyset$. If u' and v'' are not independent, then u' is a proper descendant of v'' in T'', so the tube $R_{u'}$ is contained in the tube at some child of v'' in T'', and once again $K \cap R_{u'} = \emptyset$. Since $K \subseteq R_{v'}$ and K is disjoint from the tube at any child of v' in T', K is contained in the T'-pocket at v'. □

5.6. Let R be a pocket of the (P, L)-fringe at a fixed level in the refinement process. We want to compute where the contour intersects all the subpocket boundaries within R, where the subpockets are those of T''.

Note that the structure of T'' can be built in bounded time, and for any node of T'' the enclosing pocket in T' has been precomputed (Lemma 4.12).

There are $O(k_i)$ processors available to carry out this task for R. One deals separately with each of the (at most three) fragments, intervals of the contour intersecting R. The fragments can be assumed to be stored with the record for v, where v is the node of the (P, L)-fringe owning the pocket. The points where the contour crosses the pocket are stored sorted by y-coordinate. Because the contour is monotonic in the y-direction, the highest pair of endpoints, if they exist, define the highest fragment, the next pair define the middle fragment, and the lowest pair define the lowest fragment. The corresponding pockets (at the same refinement level) of the (Q, L)-fringe containing these endpoints can also be assumed stored with the endpoints, since locating the endpoints involved locating these pockets.

Call a fragment *short* if there is a Q-fragment which contains it—in particular, all of the fragment is in the same pocket, call it R', of the (Q, L)-fringe. Otherwise it is *long* (compare with Definition 3.2). By referring to the pockets of the (Q, L)-fringe containing the fragment endpoints, it is easy to locate the Q-fragments containing them, and hence determine whether a fragment in R is short or long.

Let F be a short fragment in R; let R' be the corresponding pocket of the (Q, L)-fringe containing F. R and R' each contain $O(k_{i+1})$ subpockets each containing $O(k_{i+1})$ vertices in its respective fringe. The points where F crosses subpocket boundaries in R and R' can be calculated in $O(1)$ time using the k_i processors assigned to R. So, in

bounded time, for each short fragment F relative to the subdivision T', the points where F crosses subpocket boundaries in the (P, L)- and (Q, L)-fringes can be calculated.

Interchanging the rôles of P and Q, subpocket crossing points can be calculated for all short fragments in the (Q, L)-fringe.

Let F be a long fragment in R. Its two endpoints are located in fragments F' and F'', say, of the (Q, L)-fringe. The processors assigned to R can compute the correct subdivision of $F \cap F'$ and $F \cap F''$ in $O(1)$ parallel time. Repeating this for all (up to three) long fragments in R, the subdivision is completed, because any subpocket crossing point on F is either in F' or in F'' or in a short fragment relative to the (Q, L)-beachline.

Thus the procedure can be iterated with bounded time per iteration, at the last iteration the pockets contain a bounded number of fringe edges, and we conclude

THEOREM 5.7. *Given arrays covering the (P, L)- and (Q, L)-fringes, the (P, Q)-contour vertices can be located in $O(\log\log n)$ parallel steps using n processors (CREW).*

5.8. Once a point where a fringe edge meets the contour has been calculated, information about the vertex can be stored with the fringe. Suppose that e is an edge of the (P, L)-fringe; let u be e's starting vertex; suppose that it is associated with fringe feature f. If u is a cusp, then $f = u$; otherwise u is the inner vertex associated with f. Let a be the point where e crosses the contour.

Then the following can be stored with f: the coordinates of a, and an index to the (Q, L)-fringe segment whose region contains a. Let s_1 and s_2 be the segments directly above and below the leftmost and rightmost descendant of u in the (P, L)-fringe; then a is the lowest point where A_{s_1} meets the contour and the highest where A_{s_2} meets the contour; so this information (i.e., an index to f) can be associated with s_1 and s_2 in the (P, L)-fringe. In other words,

LEMMA 5.9. *Edges of the (P, L)-fringe which meet the contour can be pointer linked.*

Since the corresponding fringe features follow the same sorted order as the corresponding contour vertices (Lemma 2.27(ii)), we can use broadcasting (Lemma 3.4) to cover the sorted sequence:

LEMMA 5.10. *Arrays covering*:

(i) *Those contour vertices with two adjacent sites in P.*
(ii) *All contour vertices, both in sorted order, can be constructed in time $O(\log\log(n))$.*

PROOF. (i) has been discussed; (ii) is obtained by constructing the array covering vertices with two adjacent sites in Q, and merging the two. □

6. Building Vor($P \cup Q$). In this section we address the problem of building the Voronoi diagram: up to now we have concentrated on computing contour vertices. Given

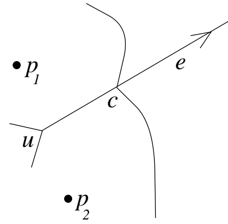

Fig. 20. Illustrating 6.3.

$S = P \cup Q$, we describe the representation of Vor(S), and show how to construct it from Vor(P) and Vor(Q), once the (P, Q)-contour has been constructed. This turns out to be easy.

We ignore the trivial cases in which P or Q have fewer than three sites; therefore Vor(P) and Vor(Q) each have at least one vertex.

6.1. The diagram Vor(S) is to be represented as a plane graph, organized as follows [17]: there are records for every site and vertex, and for each vertex there are three edge records. In this scheme, bounded edges are represented twice, and unbounded edges once, as directed edges.

Each vertex record contains the vertex coordinates. Suppose that a directed edge is oriented away from a vertex u: then we call u its "in-vertex." If the edge is bounded then we call its other end its "out-vertex."

Let e be an edge with in-vertex u. The edge record contains pointers to p_1 and p_2, where p_1 is the site on its left and p_2 the site on its right.

6.2. It is convenient to associate every Voronoi vertex with one of the adjacent sites, canonically, as follows.

Consider a Voronoi vertex v. Since none of the sites adjacent to v are covertical (2.1), none of the edges incident to v are horizontal; and either two edges extend downwards and one upwards or vice versa (Lemma 2.5). Therefore v is the highest or lowest point in one of the adjacent cells. It follows that there are at most $2|P|$ Voronoi vertices.[21] This gives a simple way to allocate space for Vor(P): to each site let there be associated two vertex records, one for the highest vertex in its cell (if the cell is bounded above), and one for the lowest (if bounded below). With each site let there be space allocated for six edge records (three for each vertex). Since the vertices and edges are stored together it is unnecessary to include pointer linkages from vertex u to the edges with in-vertex u, and so on.

6.3. In order to calculate the diagram, recall (4.7) that we assumed one more link between the beachline and the corresponding Voronoi diagram: to each cusp of the beachline is attached a pointer to an edge record. Specifically, if c is a cusp incident to an (oriented) fringe edge e, then there is a link from c to the record for the Voronoi edge e' containing e and oriented the same way as e. (If e is unbounded, then there is only one corresponding edge e', but it is oriented the same way as e: away from the beachline.) See Figure 20.

[21] If the sites are in general position, then there are $2|P| - k - 2$ Voronoi vertices, where k is the number of corners of the convex hull $H(P)$.

THEOREM 6.4. *Let A and B be arrays covering the (P, L)- and (Q, L)-fringes. Assuming the extra linkages from cusps to edges* (6.3) *have been stored with the (P, L)- and (Q, L)-fringes, and the (P, Q)-contour vertices have been located and stored in their respective fringes* (5.8), *then the graph structure for* Vor(S) *can be built in bounded parallel time.*

PROOF. (This realizes the idea of "stitching" along the contour.) Processors attached to the two beachlines combine the structures of Vor(P) and Vor(Q), allocating new contour edges and vertices, and discarding redundant parts of the diagrams as follows.

By Lemma 5.10 we can assume that the contour is sorted, so for every contour vertex its two adjacent vertices (if they exist) can be accessed. First, records are created in parallel for all contour vertices (they are stored in site records (6.2)).

Let v be a contour vertex stored in A or B; without loss of generality, two of the adjacent sites are from P and the other from Q. This means that v is where an edge a of the (P, L)-fringe meets the contour (we call a a "contour attachment"). Let e be the (directed) edge of Vor(P) containing a; let z be its in-vertex. If a meets the (P, L)-beachline, then e is associated with the cusp where it meets the beachline; otherwise, e is associated with the inner fringe vertex representing z.

A single processor can initialize records for the three new edges with in-vertex v (they are stored with the same site as v is), installing the sites adjacent to each of these edges. Two of these edges are along the contour, the third is opposite to e in direction.

For all contour "attachments" a in parallel, access the edge e containing a in Vor(P) (or Vor(Q)), mark its old inverse (if it exists) "deleted," and calculate its new inverse as the edge (v, z), where z is the in-vertex for e and v is the vertex where a crosses the contour.

Again access all these attachments in parallel; following the same notation, if e is not marked "deleted," install (v, z) as its inverse and make e the inverse for (v, z). Otherwise e and its inverse both cross the contour twice, at vertices v and v', say; the records for (v, v') and (v', v) are made mutually inverse. This finishes calculation of the inverses for all noncontour edges meeting the contour. For the contour edges the calculation is trivial, since the contour vertices are sorted vertically.

The structure of Vor(S) consists of the Voronoi vertices and edges not marked "deleted," together with the linkages described above; these vertices and edges are stored in the site records for S. □

7. Building the $(P \cup Q, M)$-Fringe. In Section 5 we used the (P, L)- and (Q, L)-fringes to construct the (P, Q)-contour. To allow $S = P \cup Q$ to adopt the role of P or Q at a higher level of recursion, we need to compute the (S, K)- and (S, M)-fringes, where K and M are vertical lines to the left and right of $S = P \cup Q$.

In this section we show how to construct an array G covering the (S, M)-fringe; an array covering the (S, K)-fringe is constructed symmetrically.

Recall what information is needed in the array G (4.7):

- Each entry $G[j]$ covering a feature f of the (S, M)-beachline needs the range $i \cdots k$ of entries in G covering f. For the other data, we assume $j = i$ (that is, the information need only be stored in the leftmost entry covering f).

- If f is a segment, the site p owning it is stored in $G[i]$.
- If f is a cusp, its coordinates are stored in $G[i]$.
- If it is a nonseparating segment, the coordinates of its inner vertex v are stored in $G[i]$, and the indices of the records of G covering the leftmost and rightmost cusp descendants of v.
- There is a representation built of Vor(S) as a planar graph; some linkages are needed between the (S, M)-fringe and Vor(S): to each beachline cusp, a pointer to an edge of Vor(S) which meets it (6.3).

7.1. Recall (2.15) that if s is a beachline segment owned by a site p, then A_s is that part of the Voronoi cell owned by p to the right of s. For this section only we make the notation more explicit: $A_s^{P,L}$ is that part of the cell of p in Vor(P) bounded by s and closer to L than to p.

Recall also that when the contour vertices were computed, certain data were stored with the (P, L)- and (Q, L)-fringes (5.8). Suppose that a contour vertex x is where a (P, L)-fringe edge e crosses the contour. Suppose v is the fringe node (inner vertex or cusp) at which e begins; let s_1 and s_2 be the segments adjacent to leftmost and rightmost descendants of v, let f be the feature (cusp or segment) of the (P, L)-fringe owning v, and let s be the segment of the (Q, L)-fringe whose region contains x. Then the coordinates of x, and a link to s, are stored with f; and there are links between f, s_1, and s_2 (x is the lowest contour vertex meeting $A_{s_1}^{P,L}$ and the highest meeting $A_{s_2}^{P,L}$ (5.8)).

The following arrays are available, or will be constructed:

- An array A covering the (P, L)-fringe, with all data as required in 4.7 and 6.3, and also information related to contour vertices as described above.
- An array B covering the (Q, L)-fringe, similar to A.
- An array C covering the (P, M)-fringe will be constructed from A.
- An array D covering the (Q, M)-fringe.
- An array E covering the (Q, L)-fringe edges which cross the contour, sorted according to the vertical order of the associated contour vertices. This is derived from B (Lemma 5.10).
- A ruling F for the (S, M)-beachline. Recall that this is a sorted array of $O(|S|/\log(n))$ horizontal lines, such that between any adjacent lines there are $O(\log(n))$ beachline cusps. Here n refers to the size of the "global" problem, different from $|S|$. In Section 8 it will be shown how all necessary rulings can be precomputed.
- From C and D an array covering the (S, M)-beachline cusps will be constructed, and then using F an array G covering the (S, M)-beachline will be constructed. The last step is to ensure that G has size $O(|S|)$ uniformly for all sets S of sites processed. The necessary information (4.7, 6.3) will be installed in G.

REMARK 7.2. The problem of calculating the combined beachline may be dismissed as a matter of "merging" two beachlines together. Of course, merging, in the conventional sense, is heavily used in our algorithm, but since a single segment of the (P, M)-beachline could be split into many segments of the (S, M)-beachline, there are difficulties in allocating array space. Indeed, this is the main difficulty of our paper. Without careful treatment, it would lead to processor allocation problems.

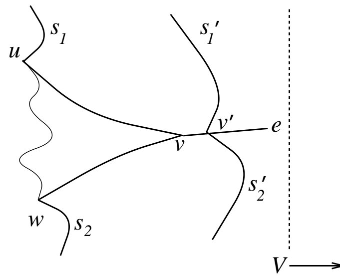

Fig. 21. Illustrating Lemma 7.3.

LEMMA 7.3.

(i) *Given an array A covering the (P, L)-fringe, an array C covering the (P, M)-fringe can be constructed in parallel time $O(\log\log(n))$. For every segment s' of the (P, M)-beachline, there is a unique corresponding segment s of the (P, L)-beachline.*
(ii) *s and s' will be mutually accessible.*

PROOF. First we explain what "corresponding segment" means. Every segment s' of the (P, M)-beachline is contained in a unique region $A_s^{P,L}$ (2.15) of the (P, L)-beachline; then s is the corresponding segment. See Figure 21. Imagine a vertical line V sweeping from L to M, and consider how the (P, V)-beachline changes. The only event changing the beachline qualitatively is where a beachline segment disappears, being occluded by the two adjacent segments (the point at which it disappears is a vertex of $\mathrm{Vor}(P)$).[22] As V moves, the segments sweep through beachline regions and are occasionally extinguished.

We say that a segment s of the (P, L)-beachline *persists* when its region $A_s^{P,L}$ (2.15) intersects the (P, M)-beachline. The segment s' where it intersects is to the right of s in the same cell of $\mathrm{Vor}(P)$. The order of beachline segments is preserved: if s_1 is above s_2 in the (P, L)-beachline, then s_1' is above s_2' in the (P, M)-beachline. Therefore, if e' is an edge of the (P, M)-fringe, and e is the edge of the (P, L)-fringe which contains it, then (Lemma 2.23) they both have the same orientation. An edge of the (P, L)-fringe persists if all or part of it is to the right of the (P, M)-beachline.

The array C is initially just a copy of A, and processors are allocated to corresponding entries in both arrays. Then:

(a) The persistent edges and segments are identified.
(b) Pointer links are installed between adjacent persisting segments; in this way, using Lemma 3.4, all the entries in C are redirected to cover persistent segments.
(c) The appropriate revised information (4.7) is stored in C.

(a) With one processor allocated to each (active) record in C, if the record covers a cusp (of the (P, L)-beachline), then that cusp is marked "absent"; if the record covers a nonseparating segment with internal vertex v, then it is first determined if v is on the

[22] We omit a proof of this rather plausible fact. See [11].

(P, M)-fringe by calculating its distance from M. Then it is marked "absent" if closer to P than to M, otherwise "present."

Next the processors determine which edges of the (P, L)-fringe intersect the (P, M)-beachline. We can assume that the parent–child relation on the (P, L)-fringe is stored in C (Lemma 4.9). An unbounded edge whose endpoint is marked absent must intersect the (P, M)-beachline. If a bounded edge (v, x) intersects the (P, M)-beachline, where x is the parent of v, then v must be absent and x present, since the orientation of the edge must be away from the beachline in *both* fringes.

Thus if a (P, L)-beachline edge oriented away from a vertex v intersects the (P, M)-beachline, the intersection point v' can be calculated by a processor assigned to v, and stored in the same record of C as v (v can be a cusp or an internal vertex). At this time also, the link required by (6.3), from v' to the edge of $\mathrm{Vor}(P)$ containing it, can be installed, since that edge is accessible through v. Thus the cusps of the (P, M)-beachline are calculated and stored.

(b) With v and v' as above, let u and w be the extremal descendant cusps of v in the (P, L)-fringe, bounding beachline segments s_1 from below and s_2 from above, say. Then all of the (P, L)-beachline between u and w is "occluded," and the regions $A_{s_1}^{P,L}$ and $A_{s_2}^{P,L}$ (2.15) intersect the new beachline in segments s_1' and s_2' with the cusp v' in common. In this way s_i' are recognized as segments which persist in the (P, M)-beachline. A processor assigned to v can attach this information to s_1 and s_2, and provide pointer links between them. See Figure 21. Applying Lemma 3.4, all the occluded parts of the beachline can be mapped to the closest nonoccluded parts in time $O(\log\log(n))$. Suppose that the occluded part of the beachline between segments s_1 and s_2 is covered by the subinterval $i \cdots k$ of C. This part of C should now cover just the single cusp v'.

(c) To set up information as described in 4.7, the processors attached to the interval $i \cdots k$ write the range $i \cdots k$ in each array entry, and the coordinates of v' into the leftmost entry. When this is done, the effect on those entries of C which covered s_i is that they now cover s_i' ($i = 1, 2$).

In general, if s is a persistent segment and s' the corresponding segment of the (P, M)-beachline, they are covered by exactly the same records of C. Therefore the connection between corresponding segments is trivial: they are covered by corresponding records in A and C, proving (ii).

Also, because edge orientations are the same in the (P, L)- and (P, M)-fringes (2.23), corresponding segments have the same inner vertex (if any).

Suppose that v is this inner vertex. The only data which need to be changed are the leftmost and rightmost cusp descendants of v. However, this is easily done: suppose that u is the leftmost cusp descendant of v in the (P, L)-fringe. If j is the leftmost index covering u in A, $C[j]$ now contains the interval $i' \cdots j'$ of records covering the cusp u' which replaces u. Similarly for the rightmost descendant. This finishes the proof of (i). □

Now we have arrays covering the (P, M)- and (Q, M)-fringes.

REMARK. The rest of this section is mostly concerned with finding short access paths to various features of the (S, M)-fringe. Sometimes these are not described in full, or are left implicit: for example, if two arrays are merged together into one, it is assumed

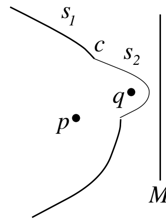

Fig. 22. Illustrating Lemma 7.4.

that from any record of the source arrays or target array the corresponding element in the other can be accessed directly. Again, some access paths can be inverted efficiently. For example, there are links from both the (Q, L)- and (Q, M)-fringes into Vor(Q) (6.3): since every edge in Vor(Q) intersects at most two edges in either fringe, by inverting the access relations in parallel we can assume that from any fringe edge the others can be quickly accessed.

LEMMA 7.4. *Suppose that s_1 and s_2 are adjacent segments on the (S, M)-beachline, with s_1 above and s_2 below their common cusp c, the segments being owned by sites p and q, respectively. If p is to the left of q, then c is the highest cusp for q, otherwise it is the lowest cusp for p.*

PROOF. Imagine that p and q are the only sites. If p is left of q, then q is closer to M and the beachline consists of three segments, the outer two owned by p and the bounded segment owned by q. Let R be the parabola (with focus q) containing the bounded segment. Any point above c on R is closer to p than to q, hence cannot be on the beachline, whether or not p and q are the only sites; hence c is the highest beachline cusp owned by q. Similarly if p is closer to M. (See Figure 22.) □

COROLLARY 7.5. *Let g be a site which owns a segment s on the (S, M)-beachline.*

(i) *If $g \in P$ (resp. Q), then this segment is contained in a segment r of the (P, M)- (resp. (Q, M)-) beachline.*
(ii) *If $g \in Q$, then r contains no other segment of the (S, M)-beachline.*[23]

PROOF. (i) Without loss of generality, $g \in P$. Let U be the set of all points on the (P, M)-beachline as close to g as to any other site in P: U is the disjoint union of all segments owned by g on the (P, M)-beachline. Any point v on s is equidistant from g and M, and as close to g as to any other site in S, hence as close as to any other site in P. Thus $s \subseteq U$. Since s is connected, it is contained in one of the connected components of U, which is a (P, M)-beachline segment r.

(ii) Suppose that $g \in Q$. The upper end of s is either that of r, or it is a point on the (P, Q)-contour, and by the above lemma it is the highest beachline cusp owned by g: thus r contains no segment above s on the combined beachline. Similarly for the lower end. □

[23] In contrast, a segment of the (P, M)-beachline can be split into many segments of the (S, M)-beachline.

Thus every segment of the (S, M)-beachline, if it is owned by a site from Q, is contained in a unique segment of the (Q, M)-beachline. The same cannot be said for sites from P, and we use another method to identify such segments.

LEMMA 7.6. *From arrays C and D covering the (P, M)- and (Q, M)-beachlines, the cusps of the (S, M)-beachline can be calculated in $O(\log\log(n))$ parallel time, and the contour cusps stored in a modified copy of D.*

PROOF. First locate each (P, M)-beachline cusp in the (Q, M)-beachline segment level with that cusp. This can be done, say, by forming copies of C and D, redirecting entries covering segments to cover adjacent cusps, and merging the two arrays together. Once this is done, a cusp u on one beachline can be marked "present" or "absent" according to whether it is to the right or left of the segment s of the other beachline whose endpoints span u.

The remaining (S, M)-beachline cusps can be calculated in $O(1)$ parallel time as follows. Let c be a cusp (yet to be computed) where a segment s_1 of the (P, M)-beachline intersects a segment s_2 of the (Q, M)-beachline. Without loss of generality, above c, s_1 is to the right of s_2, and below c, and sufficiently close to c, the converse holds. This implies that near c the (S, M)-beachline is contained in s_1 above c and in s_2 below c, and c is a cusp. See Figure 22.

Now c will be the highest cusp owned by the site q owning s_2 in the combined beachline: therefore the upper endpoint of s_2 will have been marked "absent."

A processor assigned to s_2 or s_1 can calculate the one or two points where s_2 can intersect s_1. Figure 22 illustrates the idea: calculate the intersection points of the two parabolas containing s_1 and s_2; if an intersection point is within the vertical span of s_1 and s_2, then it is a contour cusp (for example, in Figure 22, c is a contour cusp, because directly above c the beachline is owned by $p \in P$ and directly below c by $q \in Q$). The new contour cusp or cusps, if they exist, can be stored as the cusps bounding the "new" version of s_2, i.e., that part of s_2 remaining in the (S, M)-beachline. □

The beachline description is next written to an array G of size $O(|S|)$. It is here that we use a precomputed "ruling" array for the (S, M)-beachline. This is in order to compress the array G to a manageable size. Otherwise, the crude way to construct G would be to double the size of D before merging, then G would have enough room to store both cusps and segments. The effect of this, over several levels of recursion, is to increase the size requirement, and hence increase the processor requirement, to $\Omega(n^{\log_2 3})$.[24] As noted previously, this was one of the main difficulties of our paper. In the earlier version [9], not just a ruling for the beachline but the full beachline was precomputed, with $n\log(n)$ processors overall (CREW).

LEMMA 7.7. *Let C and D be arrays covering the (P, M)- and (Q, M)-beachlines, and let F be a ruling for the (S, M)-beachline. Having identified the contour cusps, an*

[24] Solution to the recurrence $F(2k) = 3F(k)$.

array G covering the (S, M)-beachline can be calculated in parallel time $O(\log\log(n))$ (CRCW).

PROOF. A cusp of the (S, M)-beachline is either a contour cusp or a cusp of the (P, M)- or (Q, M)-beachline marked "present" as described in the previous lemma.

We suppose that a modified copy D' of D has been created, covering the contour cusps and cusps of the (Q, M)-beachline. In D' the records storing the closest adjacent contour cusps can be identified in $O(\log\log(n))$ parallel time (CRCW) by forward chaining (Lemma 3.5). By this means, records covering an "absent" cusp of the (Q, M)-beachline can be changed to cover an adjacent contour cusp, so D' now covers only cusps of the (S, M)-beachline. By merging the cusps in D' with the cusps in C, for every cusp c of the (S, M)-beachline, the nearest cusp c' above c in the (P, M)-beachline, and the nearest cusp d' above c in D' (if they exist), can be located. If c' exists, is below d', and marked "present," then c' is the closest (S, M)-beachline cusp above c; otherwise, d' is, if it exists.

Thus all the (S, M)-beachline cusps can be linked together in a single-linked list. Now, by merging the cusps in C and D' with the ruling array F, this list is broken into shorter lists; if a cusp c and the closest cusp d above c are separated by a line of the ruling, then break the link connecting c to d. Since between two adjacent lines of the ruling there are $O(\log(n))$ (S, M)-beachline cusps, the result is that the (S, M)-beachline is broken into linked lists of length $O(\log(n))$. For each cusp c, its rank (distance from the end of the list) in the list containing it can be calculated by parallel list-ranking. This takes $O(\log\log(n))$ parallel time (3.7).

Let an array G of size $O(|S|)$ be prepared, subdivided into blocks of size $O(\log(n))$; each block sufficiently large to cover all of the beachline between two adjacent lines of the ruling. Let the beachline cusp of rank i in its strip be copied into location $2i$ of the corresponding block; so the cusp records will be in alternate locations of G. There is room to store the segment records between them; and unused records at the end of a block can be redirected to cover the last segment in the block. □

7.8. Next we install in G information about the (S, M)-fringe, as described in 4.7 and 6.3. The first required data, namely, for every index j the interval $i \cdots k$ of G covering the feature covered by $G[j]$, is trivial to compute and install. The coordinates of cusps will have been installed in G, and also, of course, the sites owning the various segments in the beachline.

LEMMA 7.9. *The segments in the (S, M)-beachline can be classified as "separating" (2.18) or "nonseparating" in $O(\log\log(n))$ time. Links from $H(S)$ to the separating segments can be determined also.*

PROOF. The separating rays can be calculated from $H(S)$ and M in bounded parallel time, and listed in sorted order (the sorted order derived from $H(S)$). By merging with G, the separating segments can be determined. Obviously in determining these separating segments, the appropriate links between $H(S)$ and the beachline are determined. □

7.10. Recall (Corollary 2.23) that given a point v on an edge of the (S, M)-fringe, with

adjacent sites p_1 and p_3, the edge is oriented so that the beachline segment owned by the site on the left (p_1, say), is above the segment owned by the other site. If v is a Voronoi (fringe) vertex, then the edge orientations at v are determined by the order of the three beachline segments whose regions meet at v. We assume that the three sites p_i are ordered so the segments are in descending order along the beachline, that owned by p_1 highest and p_3 lowest; then v is the inner vertex owned by the middle segment. The sites p_1, p_2, p_3 will be implicitly associated with v, with an implicit ordering, for the rest of this section.

We classify the (S, M)-fringe vertices as PPP, PPQ, $\ldots$, QQQ, according to which sets contain the corresponding sites; so, for example, if $p_1, p_2 \in P$ and $p_3 \in Q$, then the vertex is of type PPQ. Vertices of type PPP are to the left of the contour, those of type QQQ are to the right of the contour, and the others are on the contour.

We next show how to match every nonseparating segment with its inner vertex. There are eight types of vertex to be considered; five are relatively straightforward and are considered first.

LEMMA 7.11. *Let v be an (S, M)-fringe vertex owned by a segment s_2. If v is of type PPP, PQP (actually impossible), QQQ, PPQ, or QPP, then it can be associated with s_2 through linkages already present.*

PROOF. Let s_1, s_2, s_3 be the (S, M)-beachline segments whose regions meet at v, given in descending order, so s_i is owned by p_i. We deal with the five possible cases, based on the type of v.

Case 1: *v of type PPP.* The line-segments vp_1 and vp_3 meet the (S, M)-beachline in s_1 and s_3, respectively; from Lemmas 2.13 and 2.22, all of the (S, M)-fringe between the beachline and these line-segments is to the left of the contour, so all of the (S, M)-beachline between s_1 and s_3 is owned by sites from P, so v is associated with s_2 through the (P, M)-fringe (s_2 is a segment of the (P, M)-beachline).

Case 2: *v of type PQP.* By the reasoning given for Case 1, this is impossible (s_2 would be to the left of the contour).

Case 3: *v of type QQQ.* The segment s_2 is contained in a unique segment of the (Q, M)-beachline (Corollary 7.5), and v is its inner vertex in the (Q, M)-fringe. Hence v can be taken from the (Q, M)-fringe.

Case 4: *v of type PPQ.* By the same reasoning as for Case 1, the beachline between s_1 and s_2 is owned by sites from P, and hence the cusp bounding s_2 from above is on the (P, M)-fringe. Hence s_2 can be accessed through the (P, M)-fringe. Let s be the corresponding segment of the (P, L)-fringe. Then v is the highest contour vertex meeting the region $A_{s_2}^{P,L}$ (Lemma 2.27(ii)); this was associated with s_2 when calculating the contour (5.8), and hence can be accessed through s_2.

Case 5: *v of type QPP.* This is a symmetric variant of Case 4. □

7.12. We turn to vertices of type QPQ, PQQ, and QQP. Recall that we are given an array E covering all (Q, L)-fringe edges which meet the contour (at contour vertices of one of these three types, of course), in the order in which the vertices occur on the contour.

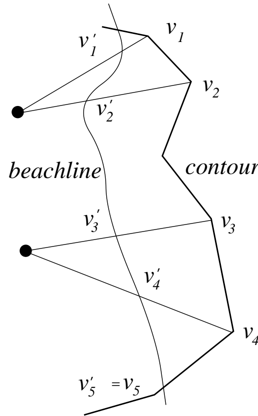

Fig. 23. Illustrating sample points on the (S, M)-beachline (7.12).

Let v be one of these vertices; define a point v' as follows; if v is to the left of the (S, M)-beachline, i.e., if it is more distant from M than to the three adjacent sites, then $v' = v$. Otherwise, let p be the unique site from P adjacent to v, and let v' be the point where the line-segment pv intersects the (S, M)-beachline. See Figure 23.

LEMMA 7.13. *The points v' are in sorted order.*

PROOF. Let J be an interval of the contour to the right of the (S, M)-beachline. By definition of the contour, the region enclosed between J and the beachline is entirely owned by sites from P. Thus, between the endpoints of J, the (S, M)-beachline coincides with the (P, M)-beachline. See Figure 23: the illustrated interval to the right of the beachline includes vertices v_1 to v_4.

Consider two contour vertices on J, of type QPQ, PQQ, or QQP, and consecutive in vertical order (though perhaps separated by contour vertices of another type).

If there are no intermediate contour vertices, for example, with v_1 and v_2 in the figure, then the vertices are both closest to the same site p in P. The points v_1' and v_2' are both on the parabola with focus p and directrix M, and, because the direction from v_1 to v_2 is clockwise around p, the direction from v_1' to v_2' is also clockwise around p, and therefore v_1' is above v_2'.

By a nearly identical argument, the point v_1' is below the upper endpoint of J, and v_4' is above the lower endpoint of J.

If there are intermediate contour vertices, for example, with v_2 and v_3 in the figure, we suppose for simplicity that there is exactly one, as with v_2 and v_3. By Lemma 2.27(iii), the segment containing v_2' is above that containing v_3', so v_2' is above v_3'.

In the more general case, where there are several attachments between v_2 and v_3, say, the beachline features owning them are in sorted order along the beachline (Lemma 2.27(ii)) and again using part (iii) of the lemma it can be argued that v_2' is above all these features and v_3' below them.

Therefore our argument applies to the whole interval J, and the endpoints of J

bracket the intermediate points v', and the argument is trivial for contour vertices left of the beachline; combining these observations, our proof is complete. □

Case 6 (QPQ vertices) is now dealt with:

COROLLARY 7.14. *Every vertex of type QPQ can be associated with the segment owning it in $O(\log\log(n))$ parallel time.*

PROOF. Merge the points v' (according to their vertical ordering) with the (S, M)-beachline; for every vertex v of type QPQ, if v is to the right of the beachline, then v' is on the beachline segment s whose inner vertex is v. □

Recall also that the (Q, L)- and (Q, M)-fringe edges are linked to the graph representation of $\mathrm{Vor}(Q)$ (6.3). Each edge of $\mathrm{Vor}(Q)$ contains at most two edges on each of these fringes; it is easy to invert these linkages in parallel, so we can assume that from any edge of the (Q, M)-fringe all of the other fringe edges (at most three) contained in the same edge of $\mathrm{Vor}(Q)$ can be accessed.

It remains to handle the PQQ and QQP vertices (Cases 7 and 8).

LEMMA 7.15. *Let v be a vertex of type PQQ or QQP. It can be associated with the segment owning it in bounded time.*

PROOF. Without loss of generality v is of type PQQ. A processor, accessing v through the (Q, L)-beachline, can access the (Q, M)-fringe edge e containing v. The segment s owning v is the unique (Lemma 7.5) segment contained in the segment s' of the (Q, M)-beachline such that $A_{s'}^{Q,M}$ meets e on the left. □

Thus each (S, M)-beachline segment has either been classified as "separating" or supplied with its inner vertex. It remains to supply for each inner vertex v the indices to its leftmost and rightmost cusp descendants, and then the links between the (S, M)-fringe and $\mathrm{Vor}(S)$ (6.3).

The following simple result is used in Lemma 7.17. It says, intuitively, that if two convex curves bend in opposite directions, then they intersect at most twice.

LEMMA 7.16. *Let L and R be two unbounded curves, monotonic in the y-direction, such that the sets A and B of points left of L (resp. right of R) are convex.* (*See Figure* 24.) *Suppose also that one of the lines, L, say, is strictly convex, so no three points on L are collinear. Then L and R intersect at most twice.*

PROOF. Let i and j be two points of intersection of L and R, with i higher than j, let H be the line through i and j, and let a be a point above i and to the right of the line H. Claim that a does not belong to A: for otherwise, take another point b left of H at the same height as a, so b belongs to the set A by definition; therefore the triangle abj, whose corners belong to A, is contained in A and therefore i, being interior to this triangle, is interior to A, which it is not.

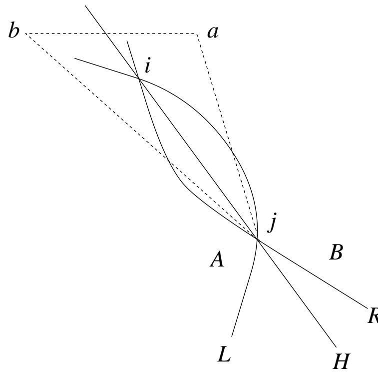

Fig. 24. Illustrating Lemma 7.16.

In other words, A contains no points above i and to the right of the line H; similarly, B contains no points above i and to the left of the line H; so above i, the intersection $A \cap B$ is contained in H. Therefore, above i, the intersection $L \cap R$ is contained in H, and therefore empty, since otherwise $L \cap R$ would contain three collinear points. By the same argument, $L \cap R$ contains no points below j. The proof is nearly finished: if the intersection contained a point x between i and j (in vertical order), then by considering i and x rather than i and j, j could not belong to the intersection. □

LEMMA 7.17. *For each nonseparating segment s_2 with inner vertex v, indices to the leftmost and rightmost cusp descendants of v can be calculated in $O(1)$ parallel time, using the linkages established in the previous lemmas.*

PROOF. We treat the seven possible types of v as before.

Case 1: *type PPP.* The cusp descendants are on the (P, M)-beachline and can be accessed from there.

Case 3: *type QQQ.* Let c be the leftmost cusp descendant of v in the (Q, M)-beachline; it is accessible through the (Q, M)-beachline, and it bounds (from above) a segment s'_1 which contains a unique segment of the (S, M)-beachline; this is s_1, it is accessible through s'_1, and its lower endpoint (which is c or is a contour cusp) is the leftmost descendant of v in the (S, M)-beachline. Similarly for the rightmost descendant.

Case 4: *type PPQ.* The leftmost descendant is as in Case 1. For the rightmost descendant, first locate the segment t of the (Q, L)-beachline (L, not M) such that $v \in A_t^{Q,L}$. The segment t is easy to access (see 5.8); but we are looking for the segment s'_3 of the (Q, M)-beachline whose region contains v. See Figure 25.

By Lemma 2.27, the contour intersects $A_t^{Q,L}$ in a connected interval I; let R' be the monotonic convex curve obtained by extending I to infinity above and below. Let L' be the parabola with focus q and directrix M, where q is the site whose cell contains $A_t^{Q,L}$. Then R' and L' satisfy the hypotheses for R and L in Lemma 7.16 and therefore intersect at most twice. Therefore I intersects the (Q, M)-beachline at most twice.

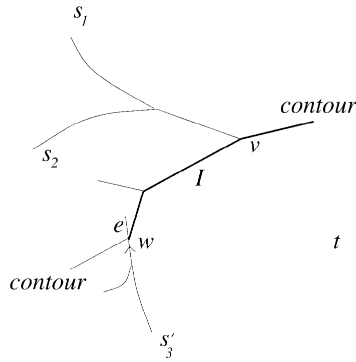

Fig. 25. Illustrating Lemma 7.17, Case 4.

Therefore either I intersects the (Q, M)-beachline at a contour cusp c below s_2, or it has a lower endpoint w to the right of the (Q, M)-beachline. In the first case, c is the rightmost descendant of v, and it is the closest contour cusp below s_2, hence accessible; otherwise, w exists and is a vertex of type QPQ. Let e be the (Q, M)-fringe edge meeting w; it is accessible through $\text{Vor}(Q)$ and the (Q, L)-fringe. Then s_3' is the segment whose region meets e from the right, and hence is accessible.

Case 5. This case is similar.

Case 6: *type* QPQ. Let e be the (Q, M)-fringe edge containing v; it can be accessed through the (Q, L)-fringe. Let s_1' and s_2' be the (Q, M)-beachline segments owning the regions on either side of e; they contain unique segments s_i of the (S, M)-fringe, and the cusp descendants sought are endpoints of these latter segments.

Case 7: *type* PQQ. The rightmost descendant can be accessed as in Case 3. For the leftmost descendant, the (S, M)-beachline segment s_1 meeting it can be accessed through E using a sample point on s_1, just as when calculating inner vertices, Case 6 (Corollary 7.14).

In other words, suppose that v is the vertex of type PQQ, and p is the site from P closest to v. Then the line-segment pv intersects the (P, M) beachline in a unique point v'; all these "sample points" v' were already calculated for the purposes of Corollary 7.14; by merging with the (S, M)-beachline, the segment s_1 containing v' can be identified.

Case 8. This case is similar. □

The last item we need consider is

LEMMA 7.18. *In* $O(\log\log(n))$ *parallel time, links can be installed from the cusps of the* (S, M)*-beachline to the edges of* $\text{Vor}(S)$ *containing them.*

PROOF. Let c be a cusp, and suppose e is the edge of $\text{Vor}(S)$ crossing c (6.3). If e is not on the (P, Q)-contour, then the existing link from the cusp as a cusp of the (P, M)- or (Q, M)-beachline can be copied. To install links from the contour cusps, merge the sorted list of contour vertices with the (S, M)-beachline cusps, both sorted in vertical

order. Thus every contour cusp is located between the two contour vertices closest in vertical order; hence the edge of $\text{Vor}(P)$ crossing the cusp can be accessed. □

8. Building a Ruling for the (P, L)-Beachline.

In this section we see how to do the essential preprocessing step of producing rulings for all the beachlines in advance. Recall, once again, that a ruling D for the (P, L)-beachline is an array of size $O(|P|/\log(n))$ containing a sorted list of horizontal lines, such that between two adjacent lines there are always $O(\log(n))$ cusps of the (P, L)-beachline. Here n is the number of sites in the global problem, and in general the number of sites in P will be less than n.

8.1. We can assume that P is available in both horizontal and vertical sorted order. The first is available since P is just a block of contiguous sites from the horizontally sorted array of sites, the second can be constructed in $O(\log(n) \log\log(n))$ time, say, for all blocks P of contiguous sites involved in the recursive construction of the diagram, by recursive application of Valiant's merging procedure [27] to sort the global set of sites in vertical order.

Let $m = |P|/\log(n)$. There are m processors, and the plane is divided into m horizontal strips, where each strip contains $\log(n)$ sites from P. The ruling for the beachline will be calculated using a vertical recursive partition of P under the control of a balanced interval tree I whose leaves cover the strips. The tree can have its nodes indexed from 1 to $2m - 1$, say, following the classical indexing of a heap. Thus each slab (i.e., union of contiguous strips) in the recursive partitioning corresponds to a node of this tree and a number in the range 1 to $2m - 1$. See Figure 26.

DEFINITION 8.2. Let U be a subset of P bounded by horizontal lines A above and B below. The *ends* of the (U, L)-beachline are those pieces above A and below B, respectively. See Figure 27.

8.3. We consider a single strip bounded between two lines A and B. Let W be the sites in the strip, and W_a and W_b the sites above and below the strip, respectively. That part of the (P, L)-beachline between A and B is formed from the beachline for W, the lower end for W_a and the upper end for W_b. Our goal in this section is to provide partial information about the ends of W_a and W_b, from which the ruling can be calculated. We do not compute the ends exactly, since that would be difficult to perform efficiently for all m strips.

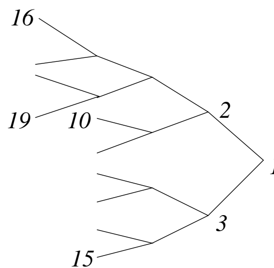

Fig. 26. Interval tree, with node indexing.

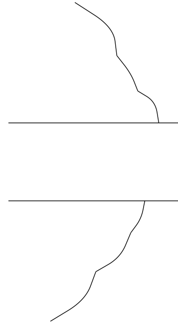

Fig. 27. Ends of a partial beachline.

LEMMA 8.4.

(i) *The parent of a node indexed* $k \neq 1$ *in* I *is indexed* $\lfloor k/2 \rfloor$, *and the node is a left* (*right*) *sibling iff* k *is even* (*odd*).
(ii) *Given a strip* W, *the set* W_a *of sites in* P *above* W *is the union of those slabs which are the left siblings of all ancestors of* W *which are right siblings*.
(iii) *The set* W_b *of all sites below* W *is the union of those slabs which are the right siblings of all ancestors of* W *which are left siblings*.

PROOF. (i) is part of the definition of I and (ii), (iii) are straightforward. □

Consider a fixed strip W. To assess the contribution of the lower end of W_a to the beachline within W, we compute for every left-sibling slab U the restriction of its lower end to the vertical interval covered by its right sibling V. Having done this, we shall be able to calculate all cusps contributed by left-sibling slabs to the vertical interval spanned by the strip W. This provides an estimate of the number of cusps contributed by W_a to the strip; similarly for the upper end of W_b.

The remainder of this section proceeds as follows:

(i) After some simple observations about the shape of the upper and lower ends, it gives an optimal parallel algorithm constructing the ends of a *horizontally sorted* set U of sites. Then the construction:
(ii) Generates a sequence of pairs (J, p), where J indexes a right-sibling slab V, and p is a site in its left sibling U, which possibly owns[25] a segment of the lower end of U in the vertical interval spanned by V: this sequence is sorted according to the horizontal ordering of sites p.
(iii) Reorders this sequence by stable integer sorting, so for each right-sibling slab indexed J there is a horizontally sorted sequence of the pairs (J, p).
(iv) Thus computes for each left-sibling slab the portion of its lower end in the vertical span of its right sibling V.
(v) Reorganizes this information and calculates the ruling.

[25] The criterion for generating the pair (J, p) is based on the horizontal position of p in the slab U.

The task would be much simpler if all the slabs were available in horizontally sorted order. However, the slabs are not disjoint, and their total size is about $m \log(m) \log(n) = |P| \log(m)$. Thus we do not attempt to produce them all in sorted order with just m processors. Step (ii) is intended to reduce the data to a manageable quantity $(O(|P|))$.

LEMMA 8.5. *Let U be a subset of P bounded by horizontal lines A from above and B from below. Then*:

(i) *Each site in U owns at most one segment in each end.*
(ii) *The vertical ordering of cusps along each end matches the horizontal ordering of sites in U.*

PROOF. Following the reasoning of Lemma 7.4, if s and s' are segments in the upper end, owned by sites p and p', respectively, meeting at a common cusp c bounding s from below and s' from above, then p is to the left of p' and c is the uppermost cusp owned by p'. Furthermore, if one imagines the parabolic segments s and s' continued below c, then they do not intersect again above p', so c is the lowest cusp owned by p in the upper end. Thus s is the lowest segment owned by p and s' is the highest owned by p', in the upper end: so each site owns at most one segment.

Left-to-right order of sites in U corresponds to downward order for segments on the upper end and upward order on the lower end. □

LEMMA 8.6. *If U is a horizontally sorted array of sites between horizontal lines A and B, then its ends can be constructed*:

(i) *In linear time by one processor.*
(ii) *In time $O(\log(n))$ with $|U|/\log(n)$ processors.*

PROOF. See Figure 28. (i) Divide-and-conquer based on the horizontal subdivision is applied. We consider only the lower end for U. This is built by a recursive combination procedure, whose recursive step involves taking subsets U_1 and U_2 (vertical sections

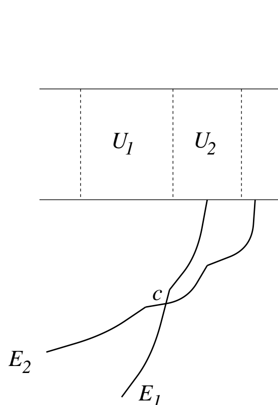

Fig. 28. Illustrating Lemma 8.6.

from U contiguous in horizontal order, where U_1, say, is to the left of U_2) with lower ends E_1 and E_2, respectively, finding the unique point c below B where these ends cross (Lemma 8.5), and catenating the truncated versions of E_1 and E_2. The point c is unique if it exists, and it exists if and only if E_1 is to the left of E_2 at B. If the ends are stored in balanced trees, locating the point c can be done, say, in time $O(\log^2(|U_1 \cup U_2|))$: find whether the median cusp of E_1 is above or below c by locating it in E_2 in time $O(\log(|U_2|))$, and repeat this process $O(\log(|U_1|))$ times. To catenate the partial ends in time $O(\log(|U_1 \cup U_2|))$ is a straightforward split-and-join operation [26].

The recursion implicit in this is: with $k = |U|$, $T(k) = 2T(k/2) + \log^2(k)$, whose solution is $O(k)$.

(ii) (a) We first show how to do this with $|U|$ processors in the stated time. As in (i), the recursive step involves subsets U_1 and U_2 of U, where U_1 is left of U_2; their lower ends are E_1 and E_2, respectively. With $|U_1 \cup U_2|$ processors the point c where the ends cross can be calculated in $O(1)$ parallel time. This time the ends E_1 and E_2 are just stored in arrays. For simplicity suppose $|E_1| = |E_2| = r$.

First check whether c exists by verifying that E_1 is to the left of E_2 at B. Suppose that y is a cusp of E_1. With $\sqrt{r}$ processors the two adjacent cusps y_1 and y_2 of E_2 bracketing y can be located in $O(1)$ time, by the usual method of first sampling every $\sqrt{r}$th cusp to reduce the interval bracketing y to size $\sqrt{r}$, and then inspecting every cusp in the interval. Then it can be checked whether E_1 is to the left or the right of E_2 at y, in which case y is above (resp. below) c.

Thus with r processors a subinterval of E_1 containing c, with $\sqrt{r}$ cusps, can be identified. Applying the processors to all cusps in the subinterval, two adjacent cusps of E_1 bracketing c can be isolated. Then two adjacent cusps of E_2 bracketing c can be isolated, and c can then be calculated immediately.

This is just another application of $\sqrt{r}$ divide-and-conquer to a location problem, as in Section 4.

(b) To obtain an optimal parallel algorithm we apply a mixed strategy which imitates [28]. Again the recursive problem is to calculate the point c where two ends E_1 and E_2, from sets U_1 and U_2 of sites, cross.

The algorithm begins by separating U into vertical slabs of size $\log(n)$, assigning one processor to each slab, and calculating its end in $O(\log(n))$ serial time by (i). When each processor has calculated the lower end of the slab assigned to it, it stores the result in sorted order in a balanced tree.

These ends are recursively combined using two strategies, an "early" and a "late" strategy. The late strategy is adopted once the subsets U_1 and U_2 reach a threshold size u. We calculate u later.

In the early stages, the ends are stored in balanced trees, as in (i). Given the ends for two adjacent slabs stored in balanced trees, a single processor calculates the end for the combined slab in $O(\log^2(u))$ serial time as in (i). Since the early stage lasts until the threshold size u is reached, the overall parallel time taken by the early stages is $O(\log^3(u))$.

These balanced trees can have stored at each node the number of descendants that node has; with this information a single processor can access a cusp of given rank (in the end E represented in the tree) in $O(\log(u))$ time. In the late stages, the ends are reorganized into an array representation. We discuss later how to convert from the early to the late representation.

Let U_1 and U_2 be as before, with corresponding ends E_i; we want to calculate the point c (if any) where the lower ends cross. The lower end E_1 (and similarly E_2) is stored in a two-tier array. Let $m_1 = |U_1|/\log(n)$; E_1 is divided into at most $2m_1 - 1$ groups each of size at most $\log(n)$. The threshold size u will be chosen to ensure that $m_1 \geq \log(n)$. The cusps bounding the groups are stored in one array; the cusps within the groups are stored in m_1 arrays each of size at most $\log(n)$.

To calculate the lower end of $U_1 \cup U_2$, using, say, $2m_1$ processors, first allocate $\sqrt{2m_1}$ processors to each $\sqrt{2m_1}$th cusp y of the $< 2m_1$ cusps bounding a group of cusps of E_1. Taking a cusp bounding every $\sqrt{2m_1}$th group of E_2, y is bracketed to within $\sqrt{2m_1}$ groups of cusps; repeating the process, y is bracketed to a group; repeating this at most twice more (since $m_1 \geq \log(n)$), y is bracketed to a segment of E_2, and hence it is determined whether y is above or below c. Repeating this process three more times c is bracketed to a segment of E_1; c is similarly bracketed to a segment of E_2. Then c can be calculated immediately.

The lower end E of $U_1 \cup U_2$ is composed of all cusps in E_1 below c, then c, then all cusps in E_2 above c (assuming U_1 is to the left of U_2). This is a sequence of (at most $2m_1 - 1$) groups from E_1, followed by a group containing c alone, followed by a sequence of groups from E_2. This results in at most $4m_1 - 1$ groups, as desired.

Each stage of the late strategy takes $O(1)$ time, and there are $\log(|U|/u)$ of them; hence the late stages take $O(\log(n))$ time. The early stages take $O(\log^3(u))$ time. The only requirement is that $2m_1 \geq \log(n)$, as mentioned above; this means $2u/\log(n) \geq \log(n)$, i.e., $u = \log^2(n)/2$. Then $\log^3(u)$ is $O(\log(n))$.

It remains to see how to convert from a balanced tree of size u to a two-tier array representation, using $u/\log(n)$ processors. Using the descendant count information stored at each node, for $1 \leq i \leq u/\log(n)$, an assigned processor can locate the cusp of rank $i \log(n)$ in the tree. These cusps form the group boundaries and can be written directly to an array using all available processors. Each processor can then traverse a group of $\log(n)$ adjacent cusps stored in the tree; to traverse a group of k adjacent cusps, and write them into an array, can be accomplished in $O(k + h)$ sequential time, where h is the tree height. With $k = \log(n)$ the time is $O(\log(n))$. □

8.7. We next see how to construct a list of pairs (J, p), where J indexes a right-sibling slab V, with left sibling U directly above it, and p is a site in U possibly contributing a segment to the lower end within the vertical range of V. In other words, either p owns no segment on the lower end of U, or it has and it intersects the vertical range of V.

LEMMA 8.8. *Such a list can be created and stored in an array of size $O(|P|)$, horizontally sorted on the second key p, in time $O(\log(n) \log\log(n))$ with m processors.*

PROOF. There is one processor attached to each strip s of P. For each ancestor U of s which is a left sibling, say its right sibling V has index J; the processor creates a pair (J, p) for each site p possibly contributing a lower-end segment to V.

The processor allocated to strip s first sorts all sites in s in horizontal order, in serial time $O(\log(n) \log\log(n))$. (This is the bottleneck—the rest is $O(\log(n))$ parallel time.)

The processors next construct the upper and lower ends for the strips. A processor assigned to strip s inspects all slabs corresponding to its ancestors in the interval-tree I, traversing the tree from bottom to top. It generates a list of all slabs V which are right siblings of its ancestors in I, in bottom-up order: therefore the slabs V are visited in descending vertical order. By essentially a merging process, it locates the cusps of the lower end of s in these slabs in linear time $(O(\log(m) + \log(n)) = O(\log(n)))$, and for each site p and slab V where p owns a segment intersecting V, the processor creates a pair (J, p), where J is the index (1 to $2m - 1$) of V in I, and attaches it to a list. Since the processor takes $O(\log(n))$ time, it creates $O(\log(n))$ pairs (J, p). Therefore there are $O(m \log(n)) = O(|P|)$ such pairs created altogether.

For each site p visited it counts the number $c(p)$ of such pairs (J, p), and writes it in a record attached to p.

With a single parallel prefix computation, the partial sums of the $c(p)$, added according to horizontal order of sites p, can be calculated in $O(\log(n))$ time with m processors (3.7).

As noted above, the sum $\sum c(p)$ is $O(|P|)$. An array C of this size can be created to hold the pairs (J, p), and each site p allotted an interval $[i \cdots j]$ of length $c(p)$ in this array, based on the prefix-sum calculation.

The list of pairs (J, p) can be written into the correct interval $C[i \cdots j]$ by the processor which created the $c(p)$ pairs. The array C contains the pairs in the order desired. □

The array C can be sorted on the first index J by stable integer sorting in time $O(\log(n) \log\log(n))$, using m processors (CRCW) (Proposition 3.6).

LEMMA 8.9. *For each right-sibling slab V let U' be the set of sites in its left sibling U which can own lower-end segments within its vertical span. The lower ends of all such sets U' can be calculated in parallel time $O(\log(n))$ with m processors.*

PROOF. Let J be the index of a typical slab V. The subsequence U' is represented by a horizontally sorted block of pairs (J, p) in C, so the calculation of Lemma 8.6 can be executed. □

Of course, all the constructions described so far apply to calculating the relevant upper ends as well.

LEMMA 8.10. *Given the relevant parts of the upper and lower ends for all the slabs of the vertical partition of P, and using m processors, it is possible to calculate in $O(\log(n) \log\log(n))$ time the merged list of all cusps of all these upper and lower ends.*

PROOF. Consider a pair U and V of sibling slabs, U the left sibling, so it is directly above V. There were $|U'|/\log(n)$ processors assigned to calculate that part of the lower end of U within the span of V, where U' was the subset of sites in the slab which could have owned segments of the lower end within this span. Also, the lower end E contains $O(|U'|)$ segments and cusps.

One can arrange to merge the cusps of E with the $v = 1 + |V|/\log(n)$ strip-boundaries in V using $(|U'| + v)/\log(n)$ processors for all such pairs of slabs simultaneously. The

time taken is $O(\log(n) \log\log(n))$ using Valiant's merge algorithm [27], slowed down by a factor of $\log(n)$ to reuse the processors available.

This associates with each strip s in V the sorted list of cusps contributed to s by U; also, the number $c(U, s)$ of cusps in this list. Similarly, with each strip s in U the list of cusps contributed to s by the upper end of V can be calculated; also, the number $c(V, s)$ of cusps in the list. These numbers can be written into an array of size $2\log(m) \times m$, and with one processor per strip s the total number $c(s)$ of cusps contributed to the strip s from $O(\log(m))$ slabs above and below it can be calculated.

The sum of all these numbers $c(s)$ is $O(|P|)$ (linear in the total number of pairs (J, p) described above). By applying parallel prefix, for each strip s a block of size $2c(s)$ can be reserved in an array, and the various lists contributing to the strip can be stored together in the first half of the block. The second half will be used as temporary storage for merging.

For each strip s, we then want to merge together about $2\log(m)$ lists: we iterate Valiant's algorithm, as follows. For each strip s, $c(s)/\log(n)$ processors can be assigned to merge together all the $2\log(m)$ lists of cusps contributing to the strip s. A single merge step can be executed in time $O(\log\log(c(s))$ using $c(s)/\log\log(c(s))$ processors by Valiant's merge procedure, with optimized processor usage [18]; hence in time $O(\log(n))$ with the available number of processors.[26] Iterating over the $O(\log\log(n))$ phases the overall runtime is $O(\log(n) \log\log(n))$. Ultimately we obtain the sorted list of cusps for each strip. □

DEFINITION 8.11. The ruling D of the beachline consists of the m strip boundaries, together with, for each strip containing more than $\log(n)$ cusps, a horizontal line through every $\log(n)$th cusp.

THEOREM 8.12. *There are $O(m)$ lines in the ruling and between any two adjacent lines there are $O(\log(n))$ cusps of the (P, L)-beachline.*

PROOF. Let t be the strip between two adjacent lines in the ruling. It is entirely contained in a strip s containing $\log(n)$ adjacent sites of P. The sites above s, U_a, say, have been extracted from about $\log(m)$ disjoint slabs, and the total number of beachline cusps contributed to t by these slabs is $O(\log(n))$.

Therefore the total number of beachline segments contributed from above is $O(\log(n))$. Now, because the interaction between different slabs has been lost, not all the cusps counted in t need figure in the (P, L)-beachline, and there may be other beachline cusps within t. However, the number of *sites* in U_a contributing segments to the strip t is likewise $O(\log(n))$. Likewise, the number of sites below s contributing beachline segments to t is $O(\log(n))$.

Hence the beachline between two adjacent lines of the ruling comes from $O(\log(n))$ sites and has size $O(\log(n))$. □

[26] Valiant's algorithm is not essential here; any logarithmic time, linear work algorithm will suffice.

9. Comments and Open Problems. At time of writing we believe this to be the best deterministic parallel algorithm. Obviously, one aspires to an $O(\log(n))$ time, n-processor algorithm. The methods of this paper do not seem capable of such an improvement. Fractional cascading [3] is the most obvious direction to seek such an improvement; but fractional cascading is generally about pipelined merging with successively larger samples. As remarked in paragraph 7.2, the business of combining fringes is related to merging but much more complicated; we thus find difficulties with fractional cascading.

Sampling is indeed exploited in this paper; Section 8 is about constructing a sample (a "ruling" for the beachline); but it is used just once rather than pipelined, and it is constructed by CRCW methods unconnected with fractional cascading.

A lesser goal is to make the overall work optimal, using, say, $n/\log\log(n)$ processors. The last section in this paper, in which we show how to construct the beachline rulings, uses some processor-optimized parallel techniques, so it is not obvious how to apply these techniques with $n/\log\log(n)$ processors.

Perhaps the methods of the last section could be sharpened to produce all beachlines in advance, with the same time and processor bounds. This might streamline other parts of the paper.

A significant note also is the essential way we exploit the CRCW architecture in the construction of the rulings for which "forward chaining" and integer sorting are employed [6]. These are $\Omega(\log(n))$ operations in the CREW architecture [12], [13].

The only other place where the CRCW property is required is in Lemma 7.7, where we used forward chaining to solve the following problem. Given two sorted arrays X and Y, suppose that some entries in these arrays are marked "absent," some ''present"; we want to merge the "present" entries from each array together, without compressing the arrays. The trick we use is aliasing (broadcasting), identifying the "absent" entries with nearby "present" entries; forward chaining enables this aliasing. Perhaps further geometric analysis would lead to an alternative aliasing scheme, and the CRCW requirement would be confined to Section 8.

It remains to be seen whether, and how, these time and processor bounds can be achieved in the CREW architecture.

One last point is that many of the results about the fringe data-structure derive from the Jordan Curve theorem [16], [21], and perhaps this could be generalized, for example, to the Voronoi diagram for line-segments [15].

Acknowledgements. Chee Yap made valuable contributions about the geometric problems, and Torben Hagerup supplied essential information on integer sorting. I am also indebted to two anonymous referees whose careful reviews corrected many errors and substantially improved this complicated paper.

Some of the diagrams in this paper were generated by a "beachline-driven" version of Fortune's Voronoi diagram algorithm [14], [11] suggested by the first author and implemented in Dublin by Keith Brady, Andrew Farrell, the third author, and Colman Reilly.

References

[1] A. Aggarwal, B. Chazelle, L. Guibas, C. Ó Dúnlaing, and C. Yap (1988). Parallel computational geometry. *Algorithmica*, **3**(3), 293–328.

[2] M. J. Atallah (1985). Some dynamic computational geometry problems. *Computers and Mathematics with Applications*, **11**, 1171–1181.

[3] M. J. Atallah, R. Cole, and M. T. Goodrich (1989). Cascading divide-and-conquer: a technique for designing parallel algorithms. *SIAM Journal on Computing*, **18**(3), 499–532.

[4] M. J. Atallah and M. T. Goodrich (1988). Parallel algorithms for some functions of two convex polygons. *Algorithmica*, **3**, 535–548.

[5] F. Aurenhammer (1990). Voronoi Diagrams—a Survey of a Fundamental Geometric Data Structure. Technical report, FB Mathematik Serie B, Freie Universität Berlin.

[6] P. Bhatt, K. Diks, T. Hagerup, V. Prasad, T. Radzik, and S. Saxena (1991). Improved deterministic parallel integer sorting. *Information and Computation*, **94**, 29–47.

[7] L. Boxer and R. Miller (1989). Parallel dynamic computational geometry. *Journal of New Generation Computer Systems*, **2**(3), 227–246.

[8] A. Chow (1980). Parallel Algorithms for Geometric Problems. Ph.D. thesis, Computer Science Department, University of Illinois.

[9] R. Cole, M. Goodrich, and C. Ó Dúnlaing (1990). Merging free trees in parallel for efficient Voronoi diagram construction. *Proc*. 17*th ICALP*. LNCS, vol. 443. Springer-Verlag, Berlin, pp. 432–445.

[10] D. Evans and I. Stojmenović (1989). On parallel computation of Voronoi diagrams. *Parallel Computing*, **12**, 121–125.

[11] A. Farrell (1994). Fortune's Voronoi sweepline algorithm for convex sites. M.Sc. dissertation, Department of Mathematics, Trinity College, Dublin.

[12] F. Fich (1993). The complexity of computation on the parallel random access machine. In *Synthesis of Parallel Algorithms*, ed. J. Reif. Morgan Kaufmann, Los Altos, CA.

[13] F. Fich and V. Ramachandran (1990). Lower bounds for parallel computation on linked structures. *Proc*. *Annual ACM Symp*. *on Parallel Algorithms and Architectures*, Crete, pp. 109–116.

[14] S. Fortune (1987). A sweep-line algorithm for Voronoi diagrams. *Algorithmica*, **2**(2), 153–174.

[15] M. T. Goodrich, C. Ó Dúnlaing, and C. Yap (1993). Constructing the Voronoi diagram of a set of line segments in parallel. *Algorithmica*, **9**, 128–141.

[16] M. Greenberg and J. Harper (1981). *Algebraic Topology—A First Course*. Benjamin/Cummings, Menlo Park, CA.

[17] L. Guibas and J. Stolfi (1985). Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Transactions on Graphics*, **4**, 74–123.

[18] C. Kruskal (1983). Searching, merging, and sorting in parallel computation. *IEEE Transactions on Computers*, **32**(10), 942–946.

[19] C. P. Kruskal, L. Rudolph, and M. Snir (1985). The power of parallel prefix. 1985 *Internat*. *Conf*. *on Parallel Processing*, pp. 180–185.

[20] R. E. Ladner and M. J. Fischer (1980). Parallel prefix computation. *Journal of the Association for Computing Machinery*, **27**, 831–838.

[21] E. Moise (1977). *Geometric Topology in Dimensions* 2 *and* 3. Graduate Texts in Mathematics, No. 47. Springer-Verlag, New York.

[22] C. Ó Dúnlaing (1993). Parallel computational geometry. In *Lectures on Parallel Computation*, ed. A. Gibbons and P. Spirakis. Cambridge International Series on Parallel Computation, Vol. 4. Cambridge University Press, Cambridge, pp. 77–108.

[23] I. Parberry (1987). On the time required to sum *n* semigroup elements on a parallel machine with simultaneous writes. *Theoretical Computer Science*, **51**, 239–247.

[24] J. H. Reif and S. Sen (1992). Optimal parallel algorithms for 3-dimensional convex hulls and related problems. *SIAM Journal on Computing*, **21**(3), 466–485.

[25] M. I. Shamos and D. Hoey (1975). Closest-point problems. *Proc*. 15*th IEEE Symp*. *on Foundations of Computer Science*, pp. 151–162.

[26] R. Tarjan (1983). *Data Structures and Network Algorithms*. CBMS–NSF Regional Conference Series in Applied Mathematics, No. 44. SIAM, Philadelphia, PA.

[27] L. Valiant (1975). Parallelism in comparison problems. *SIAM Journal on Computing*, **4**(3), 348–355.

[28] H. Wagener (1985). Optimally Parallel Algorithms for Convex Hull Determination. Manuscript, Technical University of Berlin.