Discrete Comput Geom 18:125–134 (1997)

Discrete & Computational
Geometry

A Near-Linear Algorithm for the Planar 2-Center Problem*

M. Sharir

School of Mathematical Sciences, Tel Aviv University,
Tel Aviv 69978, Israel
sharir@math.tau.ac.il
and
Courant Institute of Mathematical Sciences, New York University,
New York, NY 10012, USA

Abstract. We present an $O(n \log^9 n)$-time algorithm for computing the 2-center of a set S of n points in the plane (that is, a pair of congruent disks of smallest radius whose union covers S), improving the previous $O(n^2 \log n)$-time algorithm of [10].

The 2-Center Problem

Let S be a set of n points in the plane. The 2-*center* problem for S is to cover S by (the union of) two congruent closed disks whose radius is as small as possible. This is a special case of the general p-center problem, where we wish to cover S by p congruent disks whose radius is as small as possible. When p is part of the input, the problem is known to be NP-complete [15], so the complexity of algorithms for solving the p-center problem, for any fixed p, is expected to increase exponentially with p. A recent improved result in this direction, given in [9], is an $n^{O(\sqrt{p})}$-algorithm for the p-center problem. At the other extreme end, the 1-center problem, also known as the *smallest enclosing disk* problem, can be solved in $O(n)$ time [14]. The 2-center problem is the next problem down the list, and is of some practical interest, e.g., in the context of efficient transportation [4]. This problem has been studied in several recent papers [1], [5], [10], [11], and the currently best algorithm for its solution runs in time $O(n^2 \log n)$ [10].

In this paper we present a new algorithm for solving the 2-center problem. The

* Work on this paper has been supported by NSF Grants CCR-91-22103 and CCR-93-11127, by a Max-Planck Research Award, and by grants from the U.S.–Israeli Binational Science Foundation, the Israel Science Fund administered by the Israeli Academy of Sciences, and the G.I.F., the German–Israeli Foundation for Scientific Research and Development.

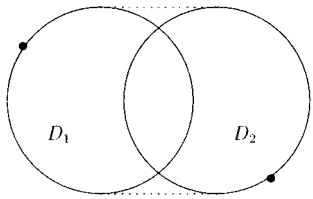

Fig. 1. C_1 and C_2 must pass through points lying on the boundary of $conv(D_1 \cup D_2)$.

algorithm runs in $O(n \log^9 n)$ time, thus providing the first subquadratic solution, and improving substantially the previous solutions. Our solution uses a mixture of techniques, including parametric searching, searching in monotone matrices, dynamic maintenance of planar configurations, and techniques similar to those used to handle "fat" objects (see [17]).

As in the previous solutions, a major component of the algorithm is a procedure for solving the fixed-size problem: Given a radius r, we want to determine whether S can be covered by two closed disks of radius r. We then combine this procedure with the parametric searching technique of [13], to obtain the complete algorithm (see below for details). We refer to this problem as the 2DC (2-disk cover) problem. The best previous solution of the 2DC problem runs in $O(n^2)$ time [6] (see also [7]). Our strategy is to assume that such a pair of disks exist, call them D_1, D_2, and to conduct a search for their centers. Let c_i denote the center of D_i, and let C_i denote the circle bounding D_i, for $i = 1, 2$. We may assume, with no loss of generality, that $|c_1c_2|$ is as small as possible. In this case it is clear that, for $i = 1, 2$, the circle C_i passes through at least one point of S that lies on the portion of C_i that appears on the boundary of the convex hull of $D_1 \cup D_2$; see Fig. 1.

Dynamic Maintenance of the Intersection of Congruent Disks

Before describing the main algorithm, we first describe in detail a procedure, which the algorithm will use repeatedly, for solving the following problem. We want to maintain dynamically a set P of points in the plane, under insertions and deletions of points. After each update, we wish to determine whether the intersection $K(P) = \bigcap_{p \in P} B_r(p)$ is nonempty, where $B_r(p)$ is the closed disk of radius r centered at p. This condition is equivalent to the condition that P can be covered by a disk of radius r. Such a procedure is also used in the preceding algorithms of [6], [7]. We give here a slightly inferior implementation of this procedure. This is done because it is easier to describe, and, more importantly, it is easier to parallelize, which is required by the parametric searching technique.

To keep track of $K(P)$ as P is being updated, we maintain separately the intersections $K^+(P) = \bigcap_{p \in P} B_r^+(p)$ and $K^-(P) = \bigcap_{p \in P} B_r^-(p)$, where $B_r^+(p)$ (resp. $B_r^-(p)$) is the region consisting of all points that lie in or above (resp. in or below) $B_r(p)$. The boundaries of these regions are (weakly) x-monotone, one of them is a convex curve and the other is concave, so it is fairly easy to determine, by a binary search through

the vertices of both regions, whether their intersection is nonempty; see below for more details.

Consider the problem of maintaining $K^+(P)$; the maintenance of $K^-(P)$ is fully symmetric. Let $\gamma(p)$ denote the boundary of $B_r^+(p)$. Note that the set $\{\gamma(p) \mid p \in P\}$ is a collection of "weak pseudolines" in the plane, meaning that any pair $\gamma(p)$, $\gamma(p')$ of curves intersect in at most one point. Moreover, $\gamma(p)$ and $\gamma(p')$ intersect if and only if their x-projections overlap (that is, the difference between the x-coordinates of p and p' is $\leq 2r$), and then $\gamma(p)$ appears to the left of $\gamma(p')$ on $\partial(B_r^+(p) \cap B_r^+(p'))$ if and only if p lies to the right of p'.

All the sets P for which we want to maintain $K^+(P)$ will be subsets of the given set S. This allows us to use the following variant of the dynamic data structure of Overmars and van Leeuwen [16]. We sort the points of S by their x-coordinates, and store them in this order at the leaves of a minimum-height binary tree T. Each node v of T maintains the intersection $K^+(P_v)$, where P_v is the subset of the current set P whose points are stored at the leaves of the subtree of T rooted at v. Each leaf of T stores a flag that indicates whether the point p of S associated with it belongs to the current set P. (Actually, to conform with the structure of internal nodes, we store the x-range of $B_r^+(p)$ at the leaf, if p belongs to the current set P, and store the full x-axis otherwise.) If v is an internal node, with a left child w_l and a right child w_r, then:

(a) v stores the x-range of $K^+(P_v)$, which is simply the intersection of the x-ranges of $K^+(P_{w_l})$ and $K^+(P_{w_r})$.
(b) If the x-range of $K^+(P_v)$ is nonempty, then the pseudoline property of the curves $\gamma(p)$, and the fact that the points of S are stored in T in increasing x-order, are easily seen to imply that $\partial K^+(P_{w_l})$ and $\partial K^+(P_{w_r})$ intersect in exactly one point q, and we also store q at v (with pointers to the pair of curves that intersect at q).

We construct, search, and update this structure as in [16]. We first describe the searching procedure. We are given a query point z and wish to determine whether z lies in $K^+(P)$. To do so, we examine the root v of T. If the x-range of $K^+(P_v)$ is empty, we report that z lies outside $K^+(P_v)$. Similarly, if the x-coordinate of z falls outside the x-range of $K^+(P_v)$, we also report that z lies outside $K^+(P_v)$. Otherwise, let q be the point stored at v. If $x(q) < x(z)$, then we continue the search recursively at the left child of v. If $x(q) > x(z)$, we continue the search at the right child of v, and if $x(q) = x(z)$, we simply test whether z lies above or below q, to obtain the answer to the query. (Note that, once we have decided that z falls in the x-range stored at the root, there is no need to repeat this test at other nodes along the search path, because the answer will always be positive.) When we reach a leaf of T, we test explicitly whether z lies in the corresponding set $B_r^+(p)$, and thereby obtain the answer to the query. The cost of the query is thus $O(\log n)$.

Consider next the updating of T, when a point p is inserted into or deleted from P. We find the path π in T leading to p, and update the data stored at the nodes of π, proceeding along π in a bottom-up fashion, and leaving all other nodes of T intact. We update the x-range stored at the leaf of p, as appropriate. To update an internal node v, with a left child w_l and a right child w_r, we first compute the intersection of the x-ranges of $K^+(P_{w_l})$ and $K^+(P_{w_r})$, and store it at v. If it is empty, no further updating at v is needed. Otherwise, we next compute the unique intersection point q_v of $\partial K^+(P_{w_l})$ and

$\partial K^+(P_{w_r})$. This is done in a manner similar to the technique of [16]. That is, let q_{w_l}, q_{w_r} be the intersection points stored at w_l, w_r, respectively. We take an arc γ_l adjacent to q_{w_l} and an arc γ_r adjacent to q_{w_r}, and apply a case analysis to decide which subtree of w_r or of w_l can be discarded in the further searching. We refer the reader to [16] for additional details concerning the analogous decision step that they use, and conclude that the updating at v can be done in $O(\log n)$ time, so the total cost of an update is $O(\log^2 n)$.

Finally, the initial construction of T, for the initial value of P (and with knowledge of the full set S), can be done in a similar manner: We sort the points of S by x-coordinate, and construct a minimum-height binary tree T over them (with the points stored at the leaves). We then traverse T in a bottom-up fashion, computing the data to be stored at each node v of T from the data already stored at the children of v, exactly as above. Since T has $O(n)$ nodes, its construction takes $O(n \log n)$ time.

We maintain a symmetric tree structure for the set $K^-(P)$, in which searches and updates are performed in a symmetric fashion. After each update, we can determine, in a final step, whether $K^+(P)$ and $K^-(P)$ intersect, by conducting a binary search through the "breakpoints" of $\partial K^+(P)$ and $\partial K^-(P)$. This search also takes $O(\log^2 n)$ time, as is easy to check.

Solving the 2DC Problem: the Case Where the Centers Are Well-Separated

Suppose first that $|c_1c_2| > r$. Let $\delta > 0$ be some sufficiently small constant angle, say 1°. We rotate the coordinate axes by $j\delta$, for $j = 0, 1, \ldots, \lfloor 2\pi/\delta \rfloor$. In one of these orientations, c_1 and c_2 will be "almost cohorizontal" (meaning that the orientation of c_1c_2 has absolute value $<\delta$), with c_1 lying to the left of c_2. In this case we have $x(c_2) - x(c_1) > r\cos\delta > 0.99r$.

Assume further that $|c_1c_2| > 3r$, say. Then a vertical line separating D_1 and D_2 exists. To detect whether this case arises, we sort the points of S by their x-coordinates, and scan them from left to right. Let S_L denote the set of points to the left of the currently scanned point q, including q, and let S_R denote the complementary set. We maintain the sets S_L and S_R dynamically, repeatedly moving each scanned point from S_R to S_L, and checking, after each update, whether $K(S_L)$ and $K(S_R)$ are nonempty. If both are nonempty, we have found two disks of radius r whose union covers S. If the currently assumed configuration does exist and we are at the correct orientation, then, in exactly one of these steps, both intersections must be nonempty, so the above procedure will detect the existence of a 2DC of this kind. Using the dynamic procedure described above, the cost of handling this case is $O(n \log^2 n)$.

Remark. This step can also be implemented using a simpler approach, which performs a binary search over the sorted sequence of points of S, to locate the line separating S_L from S_R. Each binary search step computes the smallest enclosing circles of the current S_L and S_R (in linear time, using the algorithm of [14]). If the radii of both circles are $\le r$, then we have found a solution to the 2DC problem. If both are $> r$, this subcase cannot arise. If the radius of the circle enclosing S_L (resp. of S_R) is $> r$ and the other is $\le r$, the binary search has to continue to the left (resp. to the right). This procedure takes only

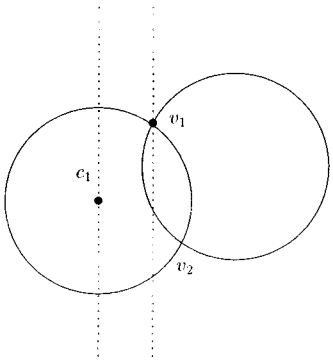

Fig. 2. The case $r < |c_1c_2| \leq 3r$

$O(n \log n)$ time, but this will be subsumed in the cost of the subsequent steps of the algorithm.

Next assume that $r < |c_1c_2| \leq 3r$. Let v_1 and v_2 denote the points of intersection of C_1 and C_2, with v_1 lying to the left of v_2. See Fig. 2. If D_1 and D_2 are disjoint, we define v_1 to be the leftmost point of D_2 and define v_2 to be the rightmost point of D_1; if v_1 lies to the right of v_2, we can proceed as in the previous case, because D_1 and D_2 are then separated by a vertical line; so we still assume that v_1 lies to the left of v_2. Since we assume that the orientation of c_1c_2 is at most δ in absolute value and that $x(c_2) - x(c_1) > 0.99r$, it is easily seen that $x(v_1) - x(c_1) > 0.4r$. Note that the left semicircle of C_1 must pass through at least one point q of S (or else we could have brought D_1 and D_2 closer together, by moving D_1 to the right). Let λ be any vertical line separating c_1 from v_1, and let S_L denote the subset of points of S lying to the left of λ. Then S_L contains q and is fully contained in D_1. Note that the difference between the largest and smallest x-coordinates of points of S is at most $5r$, so we can draw a constant number of vertical lines λ, say with horizontal separation $0.3r$ between adjacent lines, so that at least one of them will separate c_1 and v_1. Assume that λ is the correct line. We then have the set S_L available, and we compute the region $K(S_L) = \bigcap_{p \in S_L} B_r(p)$, in $O(n \log n)$ time. The above arguments imply that c_1 must lie on the (right) boundary of $K(S_L)$. For each $p \in S_R = S \backslash S_L$, we intersect $\partial B_r(p)$ with $\partial K(S_L)$. As is well known (see, e.g., [7]), each such intersection consists of at most two points. We obtain all these points, and sort them, including the vertices of $K(S_L)$, along $\partial K(S_L)$, into a list Γ. This can easily be done in $O(n \log n)$ time.

We now iterate over each point v in Γ, place the center c_1 of D_1 at v, or on the subarc of $\partial K(S_L)$ between v and the next point in Γ, and update the set $S'(c_1)$ of points of S *not covered* by D_1. We note that when c_1 moves from a point in Γ to an adjacent subarc, or from a subarc to an adjacent point, either a single point is added to $S'(c_1)$, or a single point is removed from that set, or the set remains unchanged. At each point c_1 that we visit, we test whether $S'(c_1)$ can be covered by a disk of radius r, and stop as soon as this happens, for we have obtained an affirmative solution to the 2DC problem (with radius r). Otherwise, we continue until Γ is exhausted, and conclude that λ cannot be

the desired line. If this procedure fails for all of the $O(1)$ lines λ, and for all the $O(1/\delta)$ orientations, we conclude that there is no solution to the 2DC problem (with radius r) with the currently assumed configuration. Using the dynamic procedure described earlier, the cost of handling this case is $O(n \log^2 n)$.

Solving the 2DC Problem: the Case Where the Centers Are Close to Each Other

Finally, assume that $|c_1c_2| < r$. In this case the area of $D_1 \cap D_2$ is at least $r^2\sqrt{3}/2 \approx 0.866r^2$, whereas the entire S can be covered by, say, an axis-parallel square R of size $3r$, which we can easily compute in $O(n)$ time. It follows that we can construct $O(1)$ points within R, so that at least one of them will lie in $D_1 \cap D_2$ (and fairly close to both c_1 and c_2). Let z be such a point. We sort the points of S in angular order about z, and partition the sorted list into two sublists, Q^+, Q^-, by the horizontal line passing through z. Assume that Q^+ is sorted in the clockwise direction about z and that Q^- is sorted in the counterclockwise direction. See Fig. 3 for an illustration.

Lemma 1. *Prefixes S_{L}^+ of Q^+ and S_{L}^- of Q^- exist such that $S_{\mathrm{L}}^+ \cup S_{\mathrm{L}}^- \subseteq D_1$ and $S \setminus (S_{\mathrm{L}}^+ \cup S_{\mathrm{L}}^-) \subseteq D_2$.*

Proof. Note that $A = D_1 \cup D_2$ is star-shaped with respect to z. Let ρ^+, ρ^- denote the rays emanating from z and passing through the two points of intersection of C_1 and C_2, where ρ^+ passes through the top intersection point. Let S_{L}^+ be the prefix of Q^+ consisting of points that lie counterclockwise to ρ^+ (with respect to z), and let S_{L}^- be the prefix of Q^- consisting of points that lie clockwise to ρ^-. It is easily seen that S_{L}^+ and S_{L}^- satisfy the desired properties. See Fig. 3; it is interesting to note that we only need here the fact that the point z lies in $D_1 \cap D_2$. □

We now apply a technique that resembles standard searching in monotone matrices. Let M be the matrix whose rows correspond to points in Q^+ (in clockwise angular order), and whose columns correspond to points in Q^- (in counterclockwise order). For $a \in Q^+$, $b \in Q^-$, we define M_{ab} as follows. Let ρ^+ be a ray emanating from z and

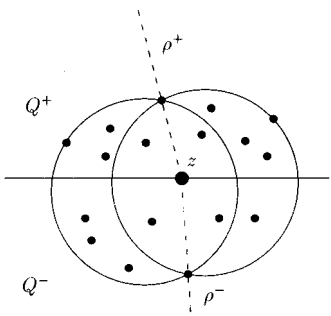

Fig. 3. The case where $|c_1c_2| < r$.

passing between a and the next point of Q^+, and let ρ^- be a ray emanating from z and passing between b and the next point of Q^-. Let S_{L}^+ be the prefix of Q^+ consisting of points that lie counterclockwise to ρ^+, and let S_{L}^- be the prefix of Q^- consisting of points that lie clockwise to ρ^-. Let $S_{\mathrm{L}} = S_{\mathrm{L}}^+ \cup S_{\mathrm{L}}^-$ and let $S_{\mathrm{R}} = S \backslash S_{\mathrm{L}}$. Then

$$M_{ab} = \begin{cases} \text{“YY”} & \text{if both } S_{\mathrm{L}} \text{ and } S_{\mathrm{R}} \text{ can be covered by disks of radius } r, \\ \text{“YN”} & \text{if } S_{\mathrm{L}} \text{ can be covered by a disk of radius } r \text{ but } S_{\mathrm{R}} \text{ cannot,} \\ \text{“NY”} & \text{if } S_{\mathrm{R}} \text{ can be covered by a disk of radius } r \text{ but } S_{\mathrm{L}} \text{ cannot,} \\ \text{“NN”} & \text{if neither } S_{\mathrm{L}} \text{ nor } S_{\mathrm{R}} \text{ can be covered by a disk of radius } r. \end{cases}$$

(Note that the number of rows plus the number of columns of M is n.) Our goal is thus to determine whether M has an entry “YY.” We denote by $M^{(\mathrm{L})}$ (resp. $M^{(\mathrm{R})}$) the matrix containing the left (resp. right) characters of the entries of M. The matrices $M^{(\mathrm{L})}$, $M^{(\mathrm{R})}$ have the following monotonicity properties, whose proof is obvious:

(a) If $M_{ab}^{(\mathrm{L})} =$ “N,” then $M_{a'b'}^{(\mathrm{L})} =$ “N” for any $a' \geq a$, $b' \geq b$.
(b) If $M_{ab}^{(\mathrm{R})} =$ “N,” then $M_{a'b'}^{(\mathrm{R})} =$ “N” for any $a' \leq a$, $b' \leq b$.

Moreover, if Γ is any sequence of entries of M, so that each element of Γ is adjacent to the preceding one in some row or column of M, then the values of all entries in Γ can be computed in time $O(n \log n + |\Gamma| \log^2 n)$. This is immediate from the dynamic scheme for maintaining intersections of disks, and from the observation that each of the sets S_{L}, S_{R} is updated by the insertion or deletion of a single point as we move from one entry in Γ to an adjacent entry.

We first compute all entries in the middle column of M. As just noted, this can be done in $O(n \log^2 n)$ time. If an entry “YY” has been detected, then we are done. Suppose we have found an entry $M_{ab} =$ “NN.” Then properties (a) and (b) imply that the top-left submatrix $\{M_{a'b'}\}_{\substack{a' \leq a \\ b' \leq b}}$ and the bottom-right submatrix $\{M_{a'b'}\}_{\substack{a' \geq a \\ b' \geq b}}$ of M can be discarded from further analysis, because they cannot contain a “YY” entry. We thus recurse with the remaining bottom-left and the top-right submatrices of M. If no “NN” entry is detected, then either all entries in the middle column are “YN,” or all are “NY,” or there is a single transition from a “YN” entry to a following “NY” entry. In the first case we can discard the left submatrix of M, and in the second case we can discard the right submatrix of M. In the third case we can discard, as above, the top-left and the bottom-right submatrices of M. (The difference from the previous case is that if $M_{ab} =$ “YN” and $M_{a+1,b} =$ “NY,” then now we discard $\{M_{a'b'}\}_{\substack{a' \leq a \\ b' \leq b}}$ and $\{M_{a'b'}\}_{\substack{a' \geq a+1 \\ b' \geq b}}$.) In each case we thus recurse on subproblems whose total size is half the size of the original matrix, so the procedure will terminate after logarithmically many steps. The terminal stage is when the current submatrix has only a single column. We then scan this column, as above; if a “YY” entry has been found, we have an affirmative solution to the 2DC problem. Otherwise, if no “YY” entry is found in any subproblem, for all possible orientations, we conclude that the currently assumed configuration cannot arise, which implies a negative solution to the 2DC problem, because by now we have exhausted all possible cases.

Concerning the efficiency of this procedure, we note that the total width and height of all submatrices in any fixed recursive level is at most n, as is easily checked. In fact, these submatrices have pairwise-disjoint row ranges and pairwise-disjoint column ranges, and

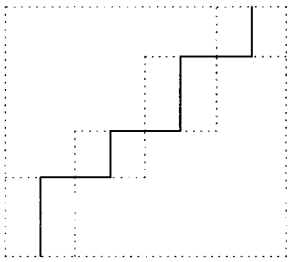

Fig. 4. The submatrices of M arising in a fixed recursive level of the algorithm, and the sequence Γ that connects their middle columns.

they lie within M in a bottom-left to top-right order; see Fig. 4. It follows that we can trace the middle columns of all submatrices in a fixed recursive level efficiently. For this, we construct a monotone sequence Γ of entries of M, consisting of the concatenation of the middle columns of the submatrices, interspersed with "horizontal moves" (along rows of M) that connect between these columns; see Fig. 4. The total length of Γ is at most n, so we can trace all its entries in a total of $O(n \log^2 n)$ time. Hence the total cost of the above procedure is $O(n \log^3 n)$.

We thus conclude:

Theorem 2. *The 2DC problem, for a set of n points in the plane and for any fixed radius r, can be solved in $O(n \log^3 n)$ time.*

Solving the 2-Center Problem

As already mentioned, we next apply the parametric searching paradigm of Megiddo [13]. To do so, we need to design an efficient parallel algorithm (in Valiant's comparisons model) for the 2DC problem, with the intention of simulating its execution at the unknown optimal radius of the 2-center solution. (We assume familiarity of the reader with the parametric searching paradigm. More details can be found, e.g., in [1].)

Most of the steps of the preceding 2DC algorithm are fairly routine to parallelize. The main difficulty is in parallelizing the dynamic maintenance of the intersection of disks, used in the various steps of the algorithm. This maintenance appears to be inherently sequential, but the data structures that we have used enable us to parallelize it efficiently. In doing so, we exploit the fact that the sequence of updates, in each application of this dynamic scheme in the algorithm, is known in advance. The parallel implementation proceeds as follows.

We first solve the following subproblem. Suppose we have the above tree structures for some subset P of S, and we have two other subsets, A^+, A^-, where we assume that $A^+ \cap A^- = \emptyset$. We want to compute, in a single step, the tree structures for the set $P \cup A^+ \setminus A^-$, where we can use $|A^+| + |A^-|$ processors for this task. We explain how to do it for the tree T that represents $K^+(P)$; the handling of the other tree is fully symmetric. We search for all leaves of T storing the points of $A^+ \cup A^-$. We next process the nodes encountered along all the search paths, level by level, in a bottom-up fashion. At each

node u we recompute the data stored at u as described for the sequential procedure. The total number of nodes at which we have to recompute the data is $(|A^+| + |A^-|)\log n$, and each recomputation takes $O(\log n)$, so we can perform this task with $|A^+| + |A^-|$ processors, in $O(\log^2 n)$ parallel steps.

We now apply the following procedure, which resembles the standard parallel prefix-sum algorithm. Let Γ be the sequence of updates to be performed on some initial set P. We construct a minimum-height binary tree Y whose leaves store the elements of Γ in order, and compute, for each node v of Y, the sets A_v^+, A_v^-, so that A_v^+ and A_v^- are disjoint, and, after all the updates of P stored at the leaves of the subtree of Y rooted at v, we have $P_{\text{new}} = P_{\text{old}} \cup A_v^+ \setminus A_v^-$, where P_{old} is the value of P before this sequence of updates, and P_{new} is its value after these updates. These sets are easy to construct in parallel, traversing Y in a bottom-up fashion, and computing the sets at a node v from the sets at its children. This can be done with $O(|\Gamma|)$ processors in $O(\log^2 n)$ parallel steps.

Once the sets A_v^+ and A_v^- are computed for all nodes v of Y, we perform a top-down traversal of Y. When we visit a node v in this traversal, we already have available the value $P_{\text{old}}^{(v)}$ of P before starting the sequence of updates stored at the subtree rooted at v. Let w_l, w_r be the left and the right child of v, respectively. We then have $P_{\text{old}}^{(w_l)} = P_{\text{old}}^{(v)}$ and $P_{\text{old}}^{(w_r)} = P_{\text{old}}^{(v)} \cup A_{w_l}^+ \setminus A_{w_l}^-$. As noted above, obtaining $K(P_{\text{old}}^{(w_r)})$ from $K(P_{\text{old}}^{(v)})$ can be done in parallel with $|A_{w_l}^+| + |A_{w_l}^-|$ processors in $O(\log^2 n)$ time.

There is however a new technical problem: Since we do not want to maintain multiple copies of the tree structures that store the various intersections $K(P)$, and since several updates are performed on these trees simultaneously at each parallel step of the algorithm, we need to organize these tree structures so that all these updates can be performed without interfering with each other. This is done using the following time-stamping scheme. Each node w of any of the two trees T used above stores a sequence σ_w of data items (each consisting of an appropriate x-range and an intersection point), sorted by the preorder of the nodes of Y that have modified w. When a new update step accesses w, it performs a binary search through σ_w, with the preorder index of the current node of Y, to find the item of σ_w that is relevant to the currently performed update. If w needs to be updated, the new value is inserted into σ_w at the appropriate place.

At the end of this top-down traversal, we have computed the value of $K(P)$ after each update operation in Γ. By the above analysis, this can be done with $O(|\Gamma|)$ processors in $O(\log^4 n)$ parallel steps (where one additional logarithmic factor is due to the cost of the time-stamping scheme, and one is due to the height of Y).

There are several other steps of the algorithm that also require parallelization, such as sorting the points of S in various orders, constructing the sets $K(S_{\text{L}})$ and their intersections with a collection of disks, and the initial construction of the tree structures of the dynamic scheme. All these steps are relatively easy to parallelize, and the cost of their parallel versions is dominated by the cost of the parallel procedure just decribed. We leave it to the reader to check the easy details.

We now plug all this into the parametric searching machinery. We note that the parallel implementation of the 2DC algorithm consists of $O(\log^5 n)$ parallel steps (the last step of the algorithm invokes the above dynamic scheme $O(\log n)$ times), and that each parallel step makes $O(\log n)$ calls to the sequential 2DC algorithm. Hence we obtain:

Theorem 3. *The* 2*-center of a set of n points in the plane can be computed in* $O(n \log^9 n)$ *time.*

Remark. We have not made a serious attempt to improve the performance of the above parametric searching procedure, because our main interest was in obtaining a near-linear solution of the 2-center problem. For example, it might be possible to adapt the more efficient technique for off-line dynamic maintenance of convex hulls, as described in [8]. It might also be possible to apply Cole's improved parametric searching technique [3], or to bypass parametric searching altogether by using either randomization (such as in [12]), or other geometric techniques (such as in [2] and [11]). We leave this as an open problem for further research. Some initial ideas toward this goal were suggested to us by Pankaj Agarwal and Matthew Katz, and we are grateful to them for sharing these ideas with us.

References

1. P. Agarwal and M. Sharir, Planar geometric location problems, *Algorithmica* **11** (1994), 185–195.
2. H. Brönnimann and B. Chazelle, Optimal slope selection via cuttings, Manuscript, 1994.
3. R. Cole, Slowing down sorting networks to obtain faster sorting algorithms, *J. Assoc. Comput. Mech.* **34** (1987), 200–208.
4. Z. Drezner, The planar two-center and two-median problems, *Transportation Sci.* **18** (1984), 351–361.
5. A. Efrat, A Simple Algorithm for Maintaining the Center of a Planar Point Set, M.Sc. Dissertation, The Technion, Haifa, 1993.
6. J. Hershberger, A faster algorithm for the two-center decision problem, *Inform. Process. Lett.* **47** (1993), 23–29.
7. J. Hershberger and S. Suri, Finding tailored partitions, *J. Algorithms* **12** (1991), 431–463.
8. J. Hershberger and S. Suri, Off-line maintenance of planar configurations, *Proc.* 2*nd ACM–SIAM Symp. on Discrete Algorithms*, 1991, pp. 32–41.
9. R. Z. Hwang, R. C. T. Lee, and R. C. Chang, The slab dividing approach to solve the euclidean P-center problem, *Algorithmica* **9** (1993), 1–22.
10. J. Jaromczyk and M. Kowaluk, An efficient algorithm for the euclidean two-center problem, *Proc.* 10*th ACM Symp. on Computational Geometry*, 1994, pp. 303–311.
11. M. Katz and M. Sharir, An expander-based approach to geometric optimization, *Proc.* 9*th ACM Symp. on Computational Geometry*, 1993, pp. 198–207.
12. J. Matoušek, Randomized optimal algorithm for slope selection, *Inform. Process. Lett.* **39** (1991), 183–187.
13. N. Megiddo, Applying parallel computation algorithms in the design of serial algorithms, *J. Assoc. Comput. Mach.* **30** (1983), 852–865.
14. N. Megiddo, Linear-time algorithms for linear programming in R^3 and related problems, *SIAM J. Comput.* **12** (1983), 759–776.
15. N. Megiddo and K. Supowit, On the complexity of some common geometric location problems, *SIAM J. Comput.* **13** (1984), 1182–1196.
16. M. Overmars and J. van Leeuwen, Maintenance of configurations in the plane, *J. Comput. System Sci.* **23** (1981), 166–204.
17. F. van der Stappen, Motion Planning Amidst Fat Obstacles, Ph.D. Dissertation, Utrecht University, 1994.

Received May 2, 1995, *and in revised form July* 8, 1995.