Algorithmica (1988) 3: 535–548

Algorithmica
© 1988 Springer-Verlag New York Inc.

Parallel Algorithms for Some Functions of Two Convex Polygons[1]

Mikhail J. Atallah[2] and Michael T. Goodrich[2]

Abstract. Let P and Q be two convex, n-vertex polygons. We consider the problem of computing, in parallel, some functions of P and Q when P and Q are disjoint. The model of parallel computation we consider is the CREW-PRAM, i.e., it is the synchronous shared-memory model where concurrent reads are allowed but no two processors can simultaneously attempt to write in the same memory location (even if they are trying to write the same thing). We show that a CREW-PRAM having $n^{1/k}$ processors can compute the following functions in $O(k^{1+\varepsilon})$ time: (i) the common tangents between P and Q, and (ii) the distance between P and Q (and hence a straight line separating them). The positive constant ε can be made arbitrarily close to zero. Even with a linear number of processors, it was not previously known how to achieve constant time performance for computing these functions. The algorithm for problem (ii) is easily modified to detect the case of zero distance as well.

Key Words. Computational geometry, Convex polygons, Parallel algorithms.

1. Introduction. Let P and Q be two convex, n-vertex polygons. We consider the problem of computing, in parallel, the following functions of P and Q when P and Q are disjoint: (i) the common tangents between P and Q, and (ii) the shortest distance between P and Q (and hence a line separating them). An easy modification of the algorithm for (ii) actually tests disjointness (it returns a nonzero distance and separating line iff they are disjoint). Throughout this paper, the model of parallel computation we use is the CREW-PRAM, i.e., it is the synchronous shared-memory model where concurrent reads are allowed but no two processors can simultaneously attempt to write in the same memory location (even if they are trying to write the same thing). Let c and d be any integers of our choice, and let $k = c^d$. We show that a CREW-PRAM having $n^{1/k}$ processors can compute the above-mentioned functions in time $O(k^{1+\varepsilon(c)})$ where $\lim_{c\to\infty} \varepsilon(c) = 0$; hence c can be chosen to be a constant that is large enough to make $\varepsilon(c)$ very close to zero. Our algorithms are nontrivial parallel generalizations of the known sequential algorithms [4], [5] for these problems.

Setting $k = 1$ in our common tangents result immediately implies an optimal $O(\log n)$ time, n processor parallel convex hull algorithm that is simpler than the ones recently given in [1] and [2]. The parallel convex hull algorithms given in [1] and [2] avoid the cleaner approach of recursively solving two subproblems

[1] This research was supported by the Office of Naval Research under Grants N00014-84-K-0502 and N00014-86-K-0689, and the National Science Foundation under Grant DCR-8451393, with matching funds from AT&T.
[2] Department of Computer Science, Purdue University, West Lafayette, IN 47907, USA.

Received October 25, 1986; revised April 17, 1987. Communicated by Bernard Chazelle.

of size $n/2$ each [3], [6], [7], because it was not known then how to find the common tangents between the two subsolutions in constant time and with n processors. Instead, these previous parallel convex hull algorithms partition the input points into $\sqrt{n}$ sets of size $\sqrt{n}$ each and, although asymptotically optimal, they are less natural than the standard solution [3], [6], [7] whose efficient parallel implementation is made possible by this paper. Essentially the same technique that we use for establishing our common tangents result is used to design a parallel algorithm for computing the shortest distance between P and Q.

The paper is organized as follows. Section 2 gives the algorithm for computing the two common tangents between P and Q such that P and Q are on the same side of each of these two tangents. Essentially the same algorithm can compute the other two common tangents (the ones such that P and Q are on opposite sides of each of them). Section 3 gives a similar result for computing the (shortest) distance between P and Q. Essentially the same algorithm can detect whether P and Q are actually disjoint or not (if not it would just return a zero value for the distance). Section 4 concludes.

2. Finding Common Tangents.

Let $P=(p_1,\ldots,p_n)$ and $Q=(q_1,\ldots,q_n)$ be two disjoint convex polygons, where the p_i's (resp. q_i's) are given in clockwise cyclic order. For convenience we assume that no three successive vertices of either polygon are colinear. Let c and d be any integers of our choice, $c=O(1)$. Let $k=c^d$. Our aim is to show that a CREW-PRAM with $n^{1/k}$ processors can compute the two common tangents beween P and Q (P and Q are on the same side of a common tangent) in time $O(k^{1+\varepsilon(c)})$, where $\lim_{c\to\infty}\varepsilon(c)=0$. By choosing c to be a large enough constant, we can make $\varepsilon(c)$ arbitrarily close to zero. As already mentioned, even the case $k=1$ of this result was previously an open question.

Since P and Q are disjoint, they are separable by a straight line. Such a separating line is not given as part of the input. However, a by-product of the algorithm we give in Section 3 is that an $n^{1/k}$ processor CREW-PRAM can, in $O(k^{1+\varepsilon(c)})$ time, find a straight line separating P and Q. For the rest of this section we assume that such a separating line (call it L) has already been found. Without loss of generality we assume that L is vertical, that P is to its left, and Q is to its right. We focus on the problem of computing the upper common tangent (that of computing the lower one being symmetrical), and we henceforth use P and Q to denote the upper portions of the two input polygons. For notational convenience we continue to assume that P and Q are n-gons, i.e., that $|P|=|Q|=n$. See Figure 1.

If we had n^2 processors available, then it would be trivial to find the desired common tangent in constant time (the detailed specification of such a brute-force algorithm is easy and is omitted). In view of this last remark, we may be tempted to give the following straightforward constant time, n processor "solution" (*which does not work*):

(i) Consider two evenly spaced $\sqrt{n}$-subsequences of the vertices of P and Q, obtaining the two $\sqrt{n}$-gons $P'=(p_{\sqrt{n}}, p_{2\sqrt{n}},\ldots,p_n)$ and $Q'=(q_{\sqrt{n}}, q_{2\sqrt{n}},\ldots,q_n)$. Use the above-mentioned brute-force approach to find

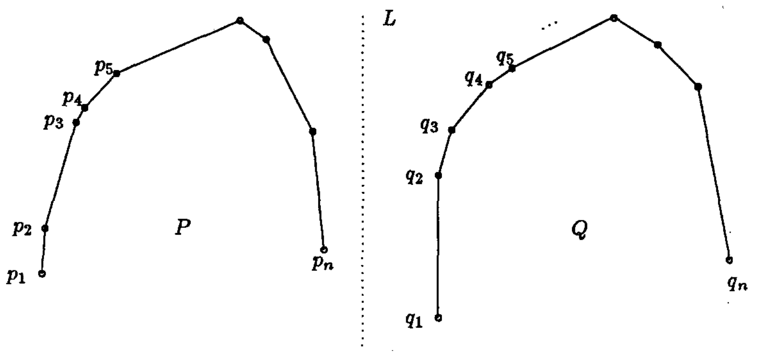

Fig. 1. The two polygons P and Q. Without loss of generality, the line separating P and Q is vertical and P is to the left of Q.

the common tangent to P' and Q' in constant time. Say it is the line joining $p_{i\sqrt{n}} \in P'$ to $q_{j\sqrt{n}} \in Q'$.

(ii) The vertices of P' (resp. Q') divide P (resp. Q) into $\sqrt{n}$ portions, call them $P_1, \ldots, P_{\sqrt{n}}$ (resp. $Q_1, \ldots, Q_{\sqrt{n}}$). Use the brute-force algorithm between the $2\sqrt{n}$ points in $P_i \cup P_{i+1}$ and the $2\sqrt{n}$ points in $Q_j \cup Q_{j+1}$ (i.e., between the portions of P adjacent to $p_{i\sqrt{n}}$ and the portions of Q adjacent to $q_{j\sqrt{n}}$).

The reason the above approach fails is that the "locality" property needed for step (ii) need not hold: indeed, the portion of P (resp. Q) containing the left (resp. right) point of tangency might be quite far from $p_{i\sqrt{n}}$ (resp. $q_{j\sqrt{n}}$). (We leave it to the reader to find an example of how this might happen.) The correct solution to the common tangent problem makes a more judicious use of the basic idea of the above (erroneous) steps (i) and (ii). It also makes use of the next two (easy) propositions.

Proposition 1. *Let p be a point external to Q. Then the upper tangent to Q passing through p can be computed in time $O(k)$ by an $n^{1/k}$ processor CREW-PRAM, where k is any integer of our choice.*

Proof. Let $t = n^{1-1/k}$. Let Q' consist of every tth vertex of Q, i.e., $Q' = (q_t, q_{2t}, \ldots, q_n)$. Since Q' has $n^{1/k}$ vertices and we have $n^{1/k}$ processors, it is trivial to find in constant time the upper tangent to Q' passing through p, say this tangent touches Q' at q_{it}. Let q_j be the vertex of Q at which the desired tangent touches Q. Test whether q_j is to the left of q_{it}, to the right of q_{it}, or at q_{it} (this test trivially takes constant time with one processor). If $q_j = q_{it}$ then we are done, so suppose (without loss of generality) that the test reveals that q_j is to the left of q_{it}, i.e., $j < it$ (the case $it < j$ is symmetrical). Then it is not hard to prove that we have $(i-1)t \leq j$ (we leave the proof to the reader). Therefore it suffices to find the upper tangent to polygon $(q_{it-t}, q_{it-t+1}, \ldots, q_{it-1})$ passing through p. Thus, by doing a constant amount of work, we have reduced the polygon size by a factor of $n^{1/k}$. Doing this at most k times finds the desired point of tangency. □

Proposition 2. *Let p be a vertex of P and let p_u be a vertex of P at which the common tangent between P and Q touches P. Then for any integer k of our choice, an $n^{1/k}$ processor CREW-PRAM can, in $O(k)$ time, determine whether p_u is to the left of p, to the right of p, or at p.*

Proof. Use Proposition 1 to find the tangent to Q passing through point p, let T be this tangent. If T is tangent to P then $p_u = p$. Otherwise, let γ be the vertex of P just to the left of p. It is obvious that p_u is to the left of p on P if and only if γ is above line T. □

The following preliminary algorithm shows that, for any integer c of our choice, an $n^{1/c}$ processor CREW-PRAM can find the common tangent to P and Q in $O(c^2)$ time.

Preliminary Algorithm A for Finding Upper Common Tangent

Input. The upper portions P and Q of two disjoint convex polygons separated by a vertical line L. Both $P = (p_1, \ldots, p_n)$ and $Q = (q_1, \ldots, q_n)$ are monotone in the x direction, i.e., the x component of p_i (resp. q_j) is smaller than that of p_{i+1} (resp. q_{j+1}). See Figure 1. *Note*: The assumption that we are already given L is not really needed, since Section 3 shows how to find such a line L.

Output. The upper common tangent to P and Q.

Step 0. Set $\hat{P} := P$, $\hat{Q} := Q$, $s := n^{1/2c}$.

Step 1. Repeat steps 2–6 until either $\hat{P}$ is a single point or $\hat{Q}$ is a single point. Without loss of generality, assume that it is $\hat{P}$ that ends up becoming a single point (call it p_u). Use Proposition 1 to find, in $O(c)$ time, the tangent to $\hat{Q}$ passing through p_u, and output the tangent thus found (this is the desired tangent between P and Q).

Step 2. Let $P' = (a_1, \ldots, a_s)$ be the polygon obtained by considering every $(|\hat{P}|/s)$th vertex of $\hat{P}$, i.e., the s vertices of P' divide $\hat{P}$ into s equal portions. Call these portions $A_1, \ldots, A_s$, so that a_i is adjacent in $\hat{P}$ to portions A_i and A_{i+1}. By definition, a_i belongs to A_i but not to A_{i+1}. Let $Q' = (b_1, \ldots, b_s)$ be analogously defined for $\hat{Q}$, and let the resulting portions of $\hat{Q}$ be $B_1, \ldots, B_s$. Use the already-mentioned brute-force method for finding the common tangent between P' and Q' (this is possible and takes constant time because we have s^2 processors). Say the tangent thus found joins $a_i \in P'$ to $b_j \in Q'$. (See Figure 2.)

Step 3. Test whether the common tangent to $\hat{P}$ and $\hat{Q}$ touches $\hat{P}$ in A_i. (This is done in $O(c)$ time by using Proposition 2 twice, once at vertex p_{i-1} and once at vertex p_i.) If the answer is "yes" then do $\hat{P} := A_i$, otherwise $\hat{P}$ remains unchanged.

Implementation Note. The assignment $\hat{P} := A_i$ is done in constant time simply by remembering the new first and last vertex of $\hat{P}$.

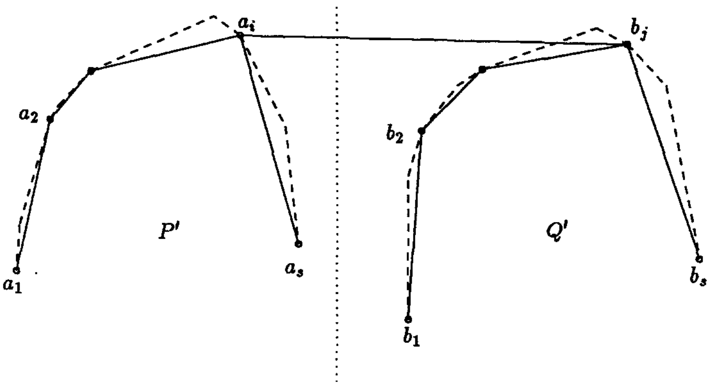

Fig. 2. The two subpolygons P' and Q' and their common supporting tangent a_ib_j. The polygons $\hat{P}$ and $\hat{Q}$ are shown by dashed lines.

Step 4. Test whether the common tangent to $\hat{P}$ and $\hat{Q}$ touches $\hat{P}$ in A_{i+1}. If it does then do $\hat{P} := A_{i+1}$, otherwise $\hat{P}$ remains unchanged.

Step 5. Test whether the common tangent to $\hat{P}$ and $\hat{Q}$ touches $\hat{Q}$ in B_j. If it does then do $\hat{Q} := B_j$, otherwise $\hat{Q}$ remains unchanged.

Step 6. Test whether the common tangent to $\hat{P}$ and $\hat{Q}$ touches $\hat{P}$ in B_{j+1}. If it does then do $\hat{Q} := B_{j+1}$, otherwise $\hat{Q}$ remains unchanged.
(*End of algorithm.*)

Note that the algorithm maintains the property that the tangent between P and Q is the same as the tangent between $\hat{P}$ and $\hat{Q}$. Thus the algorithm is correct.

Since every usage of Proposition 2 takes $O(c)$ time, the time complexity of the algorithm is equal to c multiplied by the number of times that steps 2–6 get executed. We now bound the number of times steps 2–6 are executed.

LEMMA 1. *Let a_i, b_j, $\hat{P}$, $\hat{Q}$, P', and Q' be as in step 2 of Algorithm A. Also, let p_uq_v be the common tangent to $\hat{P}$ and $\hat{Q}$ ($p_u \in \hat{P}$, $q_v \in \hat{Q}$). Then at least one of the following statements is true*:

(a) $p_u \in A_i$;
(b) $p_u \in A_{i+1}$;
(c) $q_v \in B_j$;
(d) $q_v \in B_{j+1}$.

PROOF. If $p_u = a_i$ or $q_v = b_j$ then the lemma holds, so assume that $p_u \neq a_i$ and $q_v \neq b_j$. By its definition, the line p_uq_v is above both a_i and b_j. Therefore at least one of p_u or q_v is above the line a_ib_j. Without loss of generality, assume that p_u is above the line a_ib_j. We prove that (a) or (b) must hold by a case analysis.

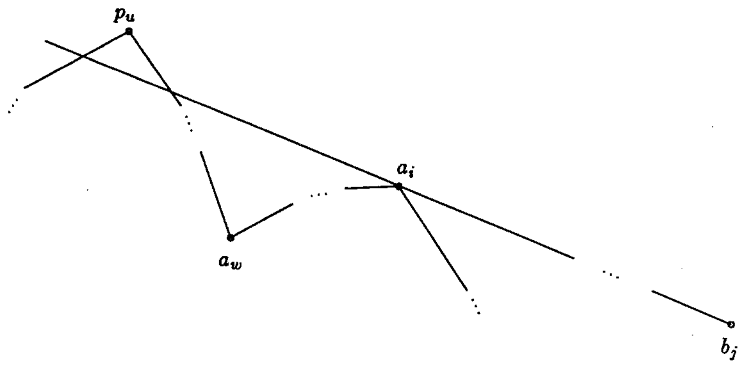

Fig. 3. We show that if $p_u \in A_w$, $w < i$, then convexity is violated.

Case 1. In $\hat{P}$, p_u is to the left of a_i. Then we claim that $p_u \in A_i$ (and hence (a) holds). Suppose to the contrary that $p_u \in A_w$ where $w < i$. By the definition of a_i and b_j, the vertex $a_w \in P'$ must lie on or below the straight line a_ib_j. The three vertices p_u, a_w, a_i occur in that order on $\hat{P}$ (see Figure 3). Consider the positions of these three vertices relative to the line a_ib_j: the first vertex is (by hypothesis) above that line, the second is (as we have just argued) on or below it, and the third is (by definition) on it. This contradicts the convexity of $\hat{P}$. Thus, (a) holds.

Case 2. In $\hat{P}$, p_u is to the right of a_i. An argument similar to that for case 1 shows that $p_u \in A_{i+1}$; hence, (b) holds.

If q_v is above line a_ib_j, then an argument similar to that above shows that one of (c) or (d) must hold. □

Corollary 1. *Steps* 2–6 *of Algorithm* A *are executed a total of at most* $4c - 1$ *times.*

Proof. Lemma 1 implies that, every time we execute steps 2–6, at least one of the statements $\hat{P} := A_i$, $\hat{P} := A_{i+1}$, $\hat{Q} := B_j$, $\hat{Q} := B_{j+1}$ is executed. This implies that at least one of $\hat{P}$ or $\hat{Q}$ decreases in size by a factor of $s = n^{1/2c}$, thus proving the corollary. □

We have thus established the following:

Theorem 1. *Algorithm* A *correctly computes the upper common tangent to P and Q. With* $n^{1/c}$ *processors, it runs in time* $O(c^2)$.

Corollary 2. *With n processors, the upper common tangent to P and Q can be computed in constant time.*

Let B_0 be the algorithm corresponding to Corollary 2, i.e. B_0 runs in $O(1)$ time with n processors. Now, we define a sequence of algorithms $B_1, B_2, \ldots$ such that

B_d uses n^{1/c^d} processors, and is defined as follows: B_d reads exactly like A except that:

(i) In B_d step 0 sets s equal to $n^{1/c}$ (instead of $n^{1/2c}$ in A).
(ii) In step 2, whereas A uses the brute-force method, B_d uses B_{d-1} (we can do this even though there are only n^{1/c^d} processors available, because B_{d-1} is being used on a subproblem of size only $n^{1/c}$).
(iii) Every usage of Proposition 1 or Proposition 2 now costs $O(c^d)$ time because we have only n^{1/c^d} processors.

Obviously, Lemma 1 still holds for B_d just as it did for A. Thus, every time steps 2–6 are executed in B_d, at least one of $\hat{P}$ or $\hat{Q}$ decreases in size by a factor of $s = n^{1/c}$. This implies that steps 2–6 in B_d are executed at most $2c-1$ times. If we let T_d be the time complexity of B_d, then we have

$$T_0 = c_1 \quad \text{and} \quad T_d = (2c-1)\cdot(T_{d-1} + c_2 \cdot c^d) + c_3 \cdot c^d,$$

where c_1, c_2, and c_3 are constants. This has solution $T_d = O(c^2(c-1)^{-1}(2c-1)^d)$. Choosing c to be a constant and using $k = c^d$ gives

$$T_d = O(k^{1+\varepsilon(c)}) \qquad \text{where} \quad \varepsilon(c) = \log_c(c^{-1}(2c-1)).$$

This establishes the following:

THEOREM 2. *Let P and Q be two disjoint convex n-gons. Let $k = c^d$ where c and d are any integers, $c = O(1)$. A CREW-PRAM having $n^{1/k}$ processors can compute the common tangents between P and Q in $O(k^{1+\varepsilon(c)})$ time, where* $\lim_{c\to\infty} \varepsilon(c) = 0$.

As already stated, the above algorithm can trivially be modified to compute the other two common tangents (the ones such that P and Q are on opposite sides of each of them). The details of these modifications are easy and are left to the interested reader.

3. Computing the Distance.

The input consists of the two disjoint convex polygons $P = (p_1, \ldots, p_n)$ and $Q = (q_1, \ldots, q_n)$, where the p_i's (resp. q_i's) are given in clockwise cyclic order and no three successive vertices of either polygon are colinear. We are interested in computing, in parallel, the shortest distance between P and Q. This distance is formally defined as follows:

$$d(P, Q) = \min_{u\in P, w\in Q} d(u, w),$$

where $d(u, w)$ denotes the Euclidean distance between points u and w, and the notation "$u \in P$" means that u is a point on the boundary of P (not necessarily a vertex of P). Our algorithm actually returns a pair of points u, w such that $d(P, Q) = d(u, w)$. Of course, once we have these points u, w, any perpendicular to the straight line segment joining u and w is a line separating P from Q. Therefore our algorithm for the closest distance immedately gives us the separating line L needed in Section 2. At the end of this section we briefly sketch the modifications needed for the algorithm to also work for the case of zero distance (in which case there is no separating line).

In order to simplify the exposition, we assume that the desired points u, w are unique. Our algorithm can easily be modified for the general case, e.g., by adopting a suitable convention for returning a unique u, w pair in case $d(P, Q)$ is the distance between two parallel segments of (respectively) P and Q (in that case there is an infinite number of choices for u, w, and this is the only case where u and w are not unique).

Let p be a point (not necessarily a vertex) on the boundary of P, and define q similarly for Q. Let T_p (resp. T_q) be the line perpendicular to the segment pq at point p (resp. q). It is quite trivial to see that $d(P, Q) = d(p, q)$ if and only if (i) T_p and T_q are tangent to (respectively) P and Q, and (ii) P and Q are on opposite sides of the region between T_p and T_q (i.e., this region separates them). This simple observation immediately implies that, with n^2 processors and in constant time, it is possible to compute the closest distance between P and Q and a pair of points achieving it (the detailed specification of this brute-force procedure is left to the reader). The algorithm we shall give uses these simple observations. It also makes use of the next two (easy) propositions.

PROPOSITION 3. *Let p be a point external to Q. Then the point $q \in Q$ such that $d(p, q) = d(p, Q)$ can be computed in time $O(k)$ by an $n^{1/k}$ processor CREW-PRAM, where k is any integer of our choice.*

PROOF. Let $t = n^{1-1/k}$. Let Q' consist of every tth vertex of Q, i.e., $Q' = (q_t, q_{2t}, \dots, q_n)$. Since Q' has $n^{1/k}$ vertices and we have $n^{1/k}$ processors, it is trivial to find in constant time the point $q' \in Q'$ such that $d(p, q') = d(p, Q')$. (Note that q' need not be a vertex of Q'.) If the perpendicular to line pq' at point q' is tangent to Q, then we can stop and declare point q' as the desired point q. Otherwise let α (resp. β) be the vertex of Q' that immediately precedes (resp. follows) point q' when the boundary of Q' is traced in a clockwise manner (see Figure 4). Note that in Q, there are $2t+1$ vertices between α and β (inclusive) if q' is a vertex, otherwise there are $t+1$ vertices between α and β (where the word "between" refers to the circular ordering $q_1, \dots, q_n$). We leave it to the reader to prove that, in Q, the desired point q occurs between α and β (inclusive). Let γ be the median of the (at most $2t+1$) vertices between α and β (inclusive): test whether the desired point q is at γ, between α and γ, or between γ and β (this test trivially takes constant time with one processor). If $q = \gamma$ then we are done, so assume (without loss of generality) that the test reveals that q is between α and γ. Hence we can focus our search for q to the section of Q between α and γ (excluding γ), which contains at most t vertices. Therefore by doing a constant amount of work, we have reduced the polygon size by a factor of at least $n^{1/k}$. Doing this at most k times finds the desired point q. □

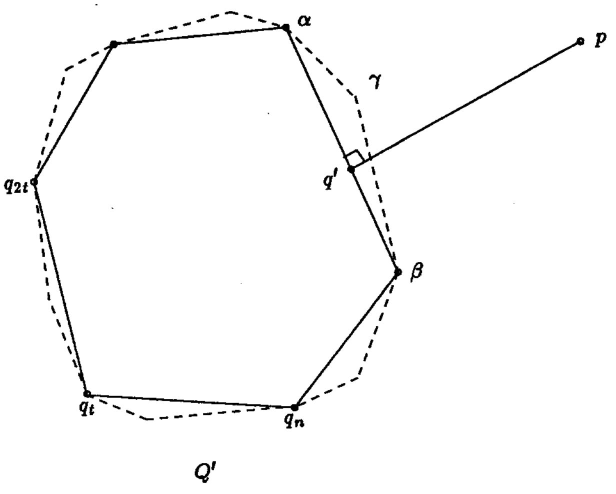

Fig. 4. Searching for the point in a convex polygon Q closest to a point p. The polygon Q is outlined by dashed lines, and the polygon Q' is outlined by solid lines.

PROPOSITION 4. *Let p_i and p_j be any two vertices of P, $i<j$, and let p_u be the vertex of P such that $d(p_u, Q) = d(P, Q)$. Then for any integer k of our choice, an $n^{1/k}$ processor CREW-PRAM can, in $O(k)$ time, locate where p_u occurs with respect to p_i and p_j in the sequence $p_1, p_2, \ldots, p_n$ (i.e., it can determine whether $u = i$, $u = j$, $i < u < j$, or none of these).*

PROOF. For any two indices $1 \le a, b \le n$, let $\sigma_{a,b}$ denote the sequence $d(p_a, Q)$, $d(p_{a+1}, Q), \ldots, d(p_b, Q)$ (assuming index $n+1$ equals 1). For example, $\sigma_{9,2} = d(p_9, Q), \ldots, d(p_n, Q), d(p_1, Q), \ldots, d(p_2, Q)$. Observe that, because of convexity, there exist two indices a and b, $1 \le a \le b \le n$, such that $\sigma_{a,b}$ and $\sigma_{b,a}$ are both sorted, one in increasing order and the other in decreasing order. This implies that we can locate where p_u occurs with respect to any pair p_i, p_j in the sequence $p_1, \ldots, p_n$ by performing a constant number of distance computations of the type $d(p_l, Q)$. By Proposition 3, each such distance computation can be done within the desired time and processor bounds. □

The following preliminary algorithm shows that, for any integer c of our choice, an $n^{1/c}$ processor CREW-PRAM can find, in $O(c^2)$ time, the points $u \in P$ and $w \in Q$ such that $d(u, w) = d(P, Q)$.

PRELIMINARY ALGORITHM D FOR COMPUTING DISTANCE

Input. Two disjoint convex polygons $P = (p_1, \ldots, p_n)$ and $Q = (q_1, \ldots, q_n)$. The p_i's (resp. q_j's) are given in clockwise cyclic order.

Output. Points u, w such that $d(u, w) = d(P, Q)$.

Step 0. Set $\hat{P} := P$, $\hat{Q} := Q$, $s := n^{1/2c}$.

Step 1. Repeat the following steps until either $\hat{P}$ is a single point or $\hat{Q}$ is a single point. Without loss of generality, assume it is $\hat{P}$ that ends up becoming a single point (call it x): use Proposition 3 to find, in $O(c)$ time, the point $y \in Q$ such that $d(x, y) = d(x, Q)$. Output the points x and y (these are the desired points u, w).

Step 2. Let $P' = (a_1, \ldots, a_s)$ be the polygon obtained by considering every $(|\hat{P}|/s)$th vertex of $\hat{P}$, i.e., the s vertices of P' divide $\hat{P}$ into s equal portions. Call these portions $A_1, \ldots, A_s$, so that a_i is adjacent in $\hat{P}$ to portions A_i and A_{i+1}. By definition, a_i belongs to A_i but not to A_{i+1}. Let $Q' = (b_1, \ldots, b_s)$ be analogously defined for $\hat{Q}$, and let the resulting portions of $\hat{Q}$ be $B_1, \ldots, B_s$. Use the already-mentioned brute-force method for finding the points $a \in P'$ and $b \in Q'$ such that $d(a, b) = d(P', Q')$. Since we have s^2 processors, this takes constant time.

Let α_P (resp. β_P) be the vertex of P' that immediately precedes (resp. follows) a on the boundary of P'. (Figure 5 illustrates the case when a is not a vertex of P'.) If a is a vertex of P' then α_P and β_P are (respectively) its predecessor and successor vertices on P', and hence there are then $2|\hat{P}|/s + 1$ vertices of $\hat{P}$ between α_P and β_P (inclusive). If a is not a vertex of P' then α_P and β_P are consecutive vertices of P', point a is on the segment of P' that joins α_P to β_P, and there are $|\hat{P}|/s + 1$ vertices of $\hat{P}$ between α_P and β_P (inclusive). Let γ_P be the median of

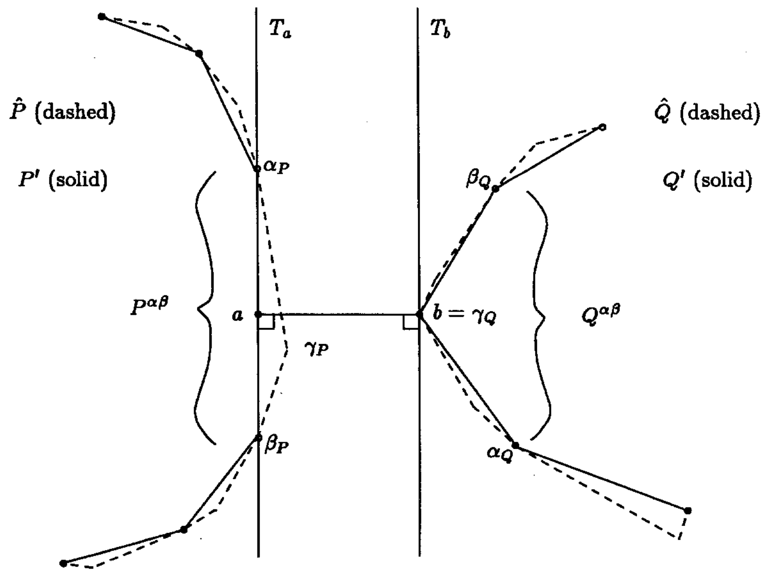

Fig. 5. Reducing the size of $\hat{P}$ and/or $\hat{Q}$. $P^{\alpha\beta}$ is the (dashed) portion of $\hat{P}$ between α_P and β_P (inclusive), and γ_P subdivides $P^{\alpha\beta}$ into $P^{\alpha\gamma}$ and $P^{\gamma\beta}$. $Q^{\alpha\beta}$, $Q^{\alpha\gamma}$, and $Q^{\gamma\beta}$ are defined analogously.

the (at most $2|\hat{P}|/s+1$) vertices of $\hat{P}$ that are between α_P and β_P (inclusive). (Note that if a is a vertex of P', then $\gamma_P = a$.) We use $P^{\alpha\beta}$ to denote the portion of P that is between α_P and β_P (excluding α_P and β_P). $P^{\alpha\gamma}$ and $P^{\gamma\beta}$ are analogously defined.

Let α_Q, β_Q, γ_Q, $Q^{\alpha\beta}$, $Q^{\alpha\gamma}$, and $Q^{\gamma\beta}$ be similarly defined for b, Q', and $\hat{Q}$. (Figure 5 illustrates the case when b is a vertex of Q'.)

Step 3. Use Proposition 4 to detect whether $u=\alpha_P$, $u=\beta_P$, $u=\gamma_P$, $u \in P^{\alpha\gamma}$, $u \in P^{\gamma\beta}$, or none of these. If u equals α_P (resp. γ_P, β_P) then set $\hat{P}$ equal to α_P (resp. γ_P, β_P) and go to step 4. Otherwise, if $u \in P^{\alpha\gamma}$ then do $\hat{P} := P^{\alpha\gamma}$ and go to step 4. Otherwise, if $u \in P^{\gamma\beta}$ then do $\hat{P} := P^{\gamma\beta}$ and go to step 4. Otherwise leave $\hat{P}$ unchanged. (An assignment like $\hat{P} := P^{\alpha\gamma}$ is done in constant time simply by remembering the new first and last vertex of $\hat{P}$.)

Step 4. Use Proposition 4 to detect whether $w=\alpha_Q$, $w=\beta_Q$, $w=\gamma_Q$, $w \in Q^{\alpha\gamma}$, $w \in Q^{\gamma\beta}$, or none of these. If w equals α_Q (resp. γ_Q, β_Q) then set $\hat{Q}$ equal to α_Q (resp. γ_Q, β_Q) and go to step 2. Otherwise, if $w = Q^{\alpha\gamma}$ then do $\hat{Q} := Q^{\alpha\gamma}$ and go to step 2. Otherwise, if $w \in Q^{\gamma\beta}$ then do $\hat{Q} := Q^{\gamma\beta}$ and go to step 2. Otherwise leave $\hat{Q}$ unchanged.
(*End of algorithm.*)

Since every usage of Proposition 4 takes $O(c)$ time, the time complexity of the algorithm is equal to c multiplied by the number of times that steps 2–4 get executed. We now bound the number of times steps 2–4 are executed.

LEMMA 2. *Let $a, b, P^{\alpha\beta}, Q^{\alpha\beta}, u$ and w be as in Algorithm* D. *Assume that $u \notin \{\alpha_P, \beta_P\}$ and $w \notin \{\alpha_Q, \beta_Q\}$. Then at least one of the following statements is true*:

(a) $u \in P^{\alpha\beta}$,
(b) $w \in Q^{\alpha\beta}$.

PROOF. Let T_a (resp. T_b) be the line perpendicular at a (resp. b) to the segment ab (see Figure 5). By the definition of a and b, T_a (resp. T_b) is tangent to P' (resp. Q'). Without loss of generality, T_a and T_b are vertical, P' is to the left of T_a, and Q' is to the right of T_b. If $u = a$ or $w = b$ then the lemma holds, so assume that $u \neq a$ and $w \neq b$. By the definition of u and w, we must have $d(u, w) \leq d(a, b)$. This implies that u is to the right of T_a or w is to the left of T_b (possibly both). Hence, it suffices to show that if u (resp. w) is to the right (resp. left) of T_a (resp. T_b), then (a) (resp. (b)) holds. We prove this by contradiction. Suppose that u is to the right of T_a and (a) does not hold, i.e. $u \notin P^{\alpha\beta}$. Without loss of generality, assume that u is below γ_P. Now, by walking from vertex γ_P clockwise along the boundary of $\hat{P}$, we encounter vertex β_P before reaching u. Since γ_P is on or to the right of T_a, β_P on or to the left of T_a, and u to the right of T_a, this contradicts the convexity of P. A similar argument shows that if w is to the left of T_b, then (b) holds. □

COROLLARY 3. *Steps* 2–4 *of Algorithm* D *are executed a total of at most* $4c-1$ *times.*

PROOF. Lemma 2 implies that, every time we execute steps 2–4, at least one of $\hat{P}$ or $\hat{Q}$ decreases in size by a factor of at least $s = n^{1/2c}$, thus proving the corollary. □

We have thus established the following:

THEOREM 3. *Algorithm* D *correctly finds points* $u \in P$ *and* $w \in Q$ *such that* $d(u, w) = d(P, Q)$. *It uses* $n^{1/c}$ *processors and it runs in time* $O(c^2)$, *where* c *is any integer of our choice.*

We now define a sequence of algorithms $\mathrm{E}_0, \mathrm{E}_1, \ldots$ in a manner entirely analogous to the way we defined sequence $\mathrm{B}_0, \mathrm{B}_1, \ldots$ in the previous section. The analysis, which is very similar to that done in Section 2 (and hence is omitted), establishes the following:

THEOREM 4. *Let* P *and* Q *be two disjoint convex n-gons. Let* $k = c^d$ *where* c *and* d *are any integers,* $c = O(1)$. *A CREW-PRAM having* $n^{1/k}$ *processors can compute points* $u \in P$ *and* $w \in Q$ *such that* $d(u, w) = d(P, Q)$ *in* $O(k^{1+\varepsilon(c)})$ *time, where* $\lim_{c\to\infty} \varepsilon(c) = 0$.

COROLLARY 4. *Let* P *and* Q *be two disjoint convex n-gons. Let* $k = c^d$ *where* c *and* d *are any integers,* $c = O(1)$. *A CREW-PRAM having* $n^{1/k}$ *processors can compute a straight line* L *which separates* P *from* Q *in* $O(k^{1+\varepsilon(c)})$ *time, where* $\lim_{c\to\infty} \varepsilon(c) = 0$.

Given a straight line L and a convex polygon P, an easier and somewhat more efficient version of the above method can be used to determine if L and the boundary of P intersect or not, and if so to give both points of this intersection. This is done in time $O(k)$ if we have $n^{1/k}$ processors, where k is any integer of our choice. The details are left to the reader.

The algorithm given in this section can easily be made to work for the case of possibly zero distance (i.e., when we do not know ahead of time whether P and Q are disjoint). The only modification needed for this is in step 2 of algorithm D. We only briefly sketch it next. In step 2 of algorithm D, insert the following computation right after the definition of P' and Q'. Let the line L be the one joining point a_1 of P' to point b_1 of Q'. Compute the intersections of L with the boundaries of P' and Q'. Since we have s^2 processors, this takes constant time. The pattern of these four intersections trivially enables one processor to test, in constant time, whether L contains a point common to both P' and Q'. If the answer to this test is "yes" then the distance is zero (i.e., P and Q intersect) and we are done. So assume that the answer to this test is "no". In that case P' and Q' intersect iff there are pairs of boundary edges $e_1 \in P'$ and $e_2 \in Q'$ that intersect each other. If there are many such pairs of intersecting edges e_1, e_2, then convexity implies that there is exactly one pair e_1, e_2 with the following property: if L_1 is the line containing e_1 and L_2 is the line containing e_2, then a walk from a_1 to b_1 along L encounters the four points a_1, $L_1 \cap L$, $L_2 \cap L$, and b_1 in that order. (See

[8] for a discussion of how this follows from convexity; the sequential intersection detection algorithm of [8] actually finds such a pair e_1, e_2 by binary search.) Therefore we can use the s^2 processors available to test by brute force whether P' and Q' intersect or not, in constant time. If they do intersect then we are done, if they do not then we proceed exactly as in the rest of step 2 and algorithm D: find the points $a \in P'$ and $b \in Q'$ such that $d(a, b) = d(P', Q') \ldots$ etc.

As a final remark, we observe that the problem of computing the shortest distance between the *boundaries* of P and Q has an $\Omega(\log n)$ lower bound on the CREW-PRAM, by the following straightforward construction. We use the fact that there is a known $\Omega(\log n)$ lower bound for the problem of computing the logical OR of n bits $x_1, \ldots, x_n$ [9]. Therefore it suffices to show that a CREW-PRAM can compute the logical OR of n bits in time equal to a constant plus the time for computing the distance between the boundaries of P and Q. We do this by converting the problem of computing the OR of $x_1, \ldots, x_n$ to that of computing the distance between the boundaries of the following two convex n-gons P and Q. P is any regular convex n-gon (i.e., all its sides have same length). Let Q_0 consist of the regular convex n-gon whose vertices are the middle points of the n boundary edges of P. The convex n-gon Q is obtained from Q_0 by "deforming" Q_0 very slightly so as to move some of its vertices slightly into the interior of P, others just outside of P. This deformation is governed by the boolean variables $x_1, \ldots, x_n$: we associate the ith vertex of Q_0 with x_i and, in the above-mentioned deformation of Q_0 into Q, it is x_i which governs whether its corresponding point of Q_0 will move slightly inside of P (if $x_i = 0$) or outside of P (if $x_i = 1$). The polygon Q so obtained has the property that its boundary is at a distance of zero from the boundary of P iff the logical OR of the x_i's is 1. (Note that Q need not be regular; it is actually regular only if the x_i's are all 0, or all 1.) The construction of P and Q from $x_1, \ldots, x_n$ can clearly be done in constant time with n processors. This establishes the claimed lower bound.

4. Conclusion.

We gave new parallel algorithms for computing some functions of convex polygons in the CREW-PRAM model. In particular, we showed that constant time suffices for computing these functions even if we have a sublinear number of processors (in fact $n^{1/c}$ processors suffice for any constant integer c). Even with a linear number of processors, it was not previously known how to achieve constant time performance for computing these functions.

References

[1] A. Aggarwal, B. Chazelle, L. Guibas, C. Ó'Dúnlaing and C. Yap, Parallel Computational Geometry, *Proceedings of the 26th Annual IEEE Symposium on Foundations of Computer Science*, October 1985, pp. 468–477.

[2] M. J. Atallah and M. T. Goodrich, Efficient Parallel Solutions to Some Geometric Problems, *Journal of Parallel and Distributed Computing*, Vol. 3, 1986, pp. 492–507.

[3] A. Chow, Parallel Algorithms for Geometric Problems, Ph.D. dissertation, Computer Science Department, University of Illinois at Urbana-Champaign, 1980.

[4] H. Edelsbrunner, Computing the Extreme Distances Between Two Convex Polygons, *Journal of Algorithms*, Vol. 6, 1985, pp. 213-224.

[5] M. H. Overmars and J. Van Leeuwen, Maintenance of Configurations in the Plane, *Journal of Computer and Systems Sciences*, Vol. 23, 1981, pp. 166-204.

[6] F. P. Preparata and S. J. Hong, Convex Hulls of Finite Sets of Points in Two and Three Dimensions, *Communications of the Association for Computing Machinery*, Vol. 20, No. 2, 1977, pp. 87-93.

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

[8] B. M. Chazelle and D. P. Dobkin, Detection is Easier than Computation, *Proceedings of the 12th ACM Annual Symposium on Theory of Computing*, 1980, pp. 146-153.

[9] S. Cook and C. Dwork, Bounds on the Time for Parallel RAM's to Compute Simple Functions, *Proceedings of the 14th ACM Annual Symposium on Theory of Computing*, 1982, pp. 231-233.