The Visual Computer

Parallel algorithms for shortest path problems in polygons

Hossam ElGindy[1]*
and Michael Goodrich[2]

[1] School of Computer Science, McGill University, Montreal, Quebec H3A 2K6, Canada
[2] Department of Computer Science, The Johns Hopkins University, Baltimore, Maryland, USA

Given an n-vertex simple polygon we address the following problems: (i) find the shortest path between two points s and d inside P, and (ii) compute the shortest-path tree between a single point s and each vertex of P (which implicitly represents all the shortest paths). We show how to solve the first problem in $O(\log n)$ time using $O(n)$ processors, and the more general second problem in $O(\log^2 n)$ time using $O(n)$ processors for any simple polygon P. We assume the CREW RAM shared memory model of computation in which concurrent reads are allowed, but no two processors should attempt to simultaneously write in the same memory location. The algorithms are based on the divide-and-conquer paradigm and are quite different from the known sequential algorithms.

Key words: Computational Geometry – Parallel algorithms – Simple polygon – Shortest path

* Research supported by the Faculty of Graduate Studies and Research (McGill University) grant 276-07

1 Introduction

We study two variations of the Euclidean shortest path problem inside a simple polygon, giving efficient parallel algorithms for each. Specifically, given an n-vertex simple polygon P, we address the following problems: (i) The Interior Shortest Path **ISP** problem: find the path with the shortest Euclidean distance between two points s and d in P, and (ii) The All Interior Shortest Paths **AISP** problem: compute the shortest-path tree between a single point s and each vertex of P (which implicity represents all the shortest paths). As in [1–4, 6, 8, 9] we are interested in solving these geometric problems as fast as possible using only a linear number of processors.

The traditional sequential way of solving shortest path problems in the presence of polygonal barriers is to construct the *visibility graph* [13, 20] and then perform the shortest path computations on this graph. This approach results in an algorithm which runs in $O(n^2)$ time [13, 20]. By restricting the input points to be interior to a single polygon, however, previous researchers [5, 7, 10, 11] have been able to design efficient sequential solutions which run much faster than the traditional approach. Namely, ElGindy [7] has shown that if the input polygon is monotone with respect to some line, then one can solve the **ISP** problem in $O(n)$ time, which is optimal. If the input polygen is an arbitrary polygon, then after a preprocessing step which takes $O(n \operatorname{loglog} n)$ time [18] one can solve both the **ISP** problem [5, 11] and the **AISP** problem [10] in $O(n)$ additional time. Unfortunately, all these algorithms use techniques similar to polygon boundary-tracing [12], which seem to be inherently sequential. Thus, we need a different approach if we are to design efficient parallel algorithms for these problems.

Using known parallel algorithms as the building blocks, the traditional approach to shortest path problems can be applied in the parallel setting. This appraoch results in a parallel algorithm which runs in $O(\log^2 n)$ time using matrix-multiplication number of processors [2, 14, 16, 17] (which is currently about $O(n^{2.5})$). For the special case of finding the shortest path between two points inside a monotone simple polygon, ElGindy [8] has shown that one can achieve this same $O(\log^2 n)$ time bound using only $O(n)$ processors. We improve on this result by showing how to find the shortest path between two points in an arbitrary simple polygon in $O(\log n)$ time using $O(n)$ processors. We also show how to construct the shortest path tree between a point and all the vertices of a simple

polygon in $O(\log^2 n)$ time using $O(n)$ processors. This tree implicitly stores the shortest path between the source point and each vertex, and can be used to compute all the shortest path distances in $O(\log n)$ additional time. Both of our algorithms are based on divide-and-conquer approaches which are quite different from known sequential algorithms, and which rely on additional geometric properties present in the problems.

The computational model we use for all of our algorithms is the CREW PRAM shared memory model in which synchronous processors share a common memory that allows for concurrent reads but does not allow two processors to simultaneously write to the same memory cell.

In the next section we give some of the definitions and preliminary observations we will be using throughout the remainder of this paper. For terms not defined there the reader is referred to the book Preparate and Shamos [15]. In Sect. 3 we present an algorithm for finding the shortest path between two points in a simple polygon, and in Sect. 4 we present an algorithm for finding the shortest path tree between a point and every vertex of a simple polygon. We finally conclude with some open problems in Sect. 5.

2 Preliminaries

Let $P=(p_1, p_2, \ldots, p_n)$ be a simple polygon. The shortest path interior to P between two points u and v, denoted by $Path(u, v)$, is only defined if u and v are not exterior to P. The *shortest path distance* between u and v is the length of $Path(u, v)$. Let $e_i=(p_i, q_i)$ and $e_j=(p_j, q_j)$ be two distinct edges of P. We call a shortest path between an endpoint of e_i and an endpoint of e_j a *point-to-point path* between e_i and e_j. We define the *common path* between e_i and e_j, denotes $CPath(e_i, e_j)$, to be the maximal common intersection of the four point-to-point paths $Path(p_i, p_j)$, $Path(p_i, q_j)$, $Path(q_i, p_j)$, and $Path(q_i, q_j)$. We define the *fundamental paths* between e_i and e_j, denoted $Paths(e_i, e_j)$, to be the five paths $Path(p_i, p_j)$, $Path(p_i, q_j)$, $Path(q_i, p_j)$, $Path(q_i, q_j)$ and $CPath(e_i, e_j)$ (Refer to Fig. 1.) We conclude this section by presenting a property of these fundamental paths that is essential to the development of efficient parallel shortest path(s) algorithms. The proof is straightforward, and is thus omitted.

Lemma 1. *Let P be an n-vertex simple polygon, and let e_i and e_j be two distinct boundary edges of P. The portions of any point-to-point path between e_i and e_j, which are not on the common path $CPath(e_i, e_j)$, are convex chains.*

3 Shortest path between two points

In this section we present a parallel algorithm for computing the internal shortest path **ISP** between two points s and d inside a simple polygon P. The algorithm runs in $O(\log n)$ time using $O(n)$ processors in the CREW PRAM model of computation. The algorithms proceeds in two phases. In the first

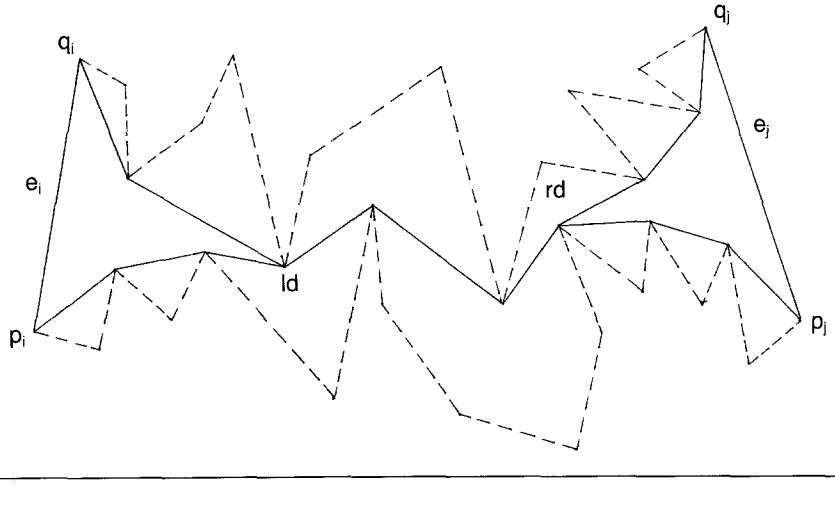

Fig. 1. The fundamental paths between two edges. The polygonal edges are shown dashed and fundamental paths are shown solid. $CPath(e_i, e_j)$ is the solid portion between ld and rd

phase we compute a triangulation TP of P and construct the dual graph TP' of TP, where each node in TP' corresponds to a triangle in TP and two nodes in TP' are adjacent if and only if the corresponding triangles in TP share an edge. TP' must be a tree since P is a simple polygon. We then determine the simple path S' in TP' from s' to d', where $s'(d')$ denotes the node of TP' that corresponds to the triangle containing $s(d)$. S' corresponds to a simple polygon S that is completely contained in P and is called a *triagulated sleeve*. Let $\{e_1, e_2, \ldots, e_m\}$ be the set of triangulation edges of S, where $e_1(e_m)$ lies on the triangle containing $s(d)$. We define a total order on the edges so that $e_i < e_j$ if and only if e_j is closer to s than the edge e_j. In the second phase, we employ the divide-and-conquer paradigm to compute the $Path(s, d)$ in the triangulated sleeve corresponding to S'.

Before presenting details of the algorithm, we describe the geometric operation which forms the basis for the divide-and-conquer approach to this problem.

Lemma 2. *Let S be a triangulated sleeve with diagonals e_i, e_j, and e_k such that $e_i < e_j < e_k$. Given $Paths(e_i, e_j)$ and $Paths(e_j, e_k)$, we can compute $Paths(e_i, e_k)$ in $O(1)$ time using $O(n)$ processors, where n is the number of vertices in S between e_i and e_k.*

Proof. Four different situations may occur:

(1) $CPath(e_i, e_j) = \Phi$ and $CPath(e_j, e_k) = \Phi$
(2) $CPath(e_i, e_j) \neq \Phi$ and $CPath(e_j, e_k) = \Phi$
(3) $CPath(e_i, e_j) = \Phi$ and $CPath(e_j, e_k) \neq \Phi$
(4) $CPath(e_i, e_j) \neq \Phi$ and $CPath(e_j, e_k) \neq \Phi$

Situations (2–4) can be viewed as special cases of the first situation with one or two of the edges degenerating to a single point. Therefore we will only present a proof of the lemma for the first situation.

In this situation, each of the paths $Path(p_i, p_j)$, $Path(p_j, p_k)$, $Path(q_i, q_j)$, and $Path(q_j, q_k)$ are convex chains. Therefore we can compute the common tangents between each pair of the four paths in $O(1)$ time using $O(n)$ processors, where n is the total number of vertices, by using the algorithms presented by Atallah and Goodrich [4] for manipulating disjoint convex chains.

In the following case analysis, we show that these tangents are sufficient for constructing $Paths(e_i, e_k)$ from $Paths(e_i, e_j)$ and $Paths(e_j, e_k)$.

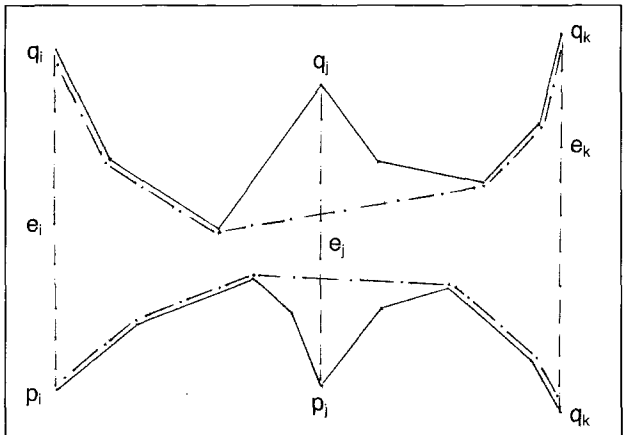

Fig. 2. The case of empty $CPath(e_i, e_k)$ of Lemma 2

If the supporting line of $Path(p_i, p_j) \cup Path(p_j, p_k)$ does not intersect $Path(q_i, q_j) \cup Path(q_j, q_k)$ and the supporting line of $Path(q_i, q_j) \cup Path(q_j, q_k)$ does not intersect $Path(p_i, p_j) \cup Path(p_j, p_k)$, then $CPath(e_i, e_k)$ is also empty. We can then construct $Paths(e_i, e_k)$ in $O(1)$ additional time by "cutting" and "pasting" the appropriate pieces of the fundamental paths (Refer to Fig. 2).

Otherwise, the critical separating line of $Path(p_i, p_j)$ and $Path(q_j, q_k)$ must intersect $Path(q_i, q_j) \cup Path(p_j, p_k)$, the critical separating line of $Path(p_j, p_k)$ and $Path(q_i, q_j)$ must intersect $Path(p_i, p_j) \cup Path(q_j, q_k)$, or both. Due to the similarity of the three cases, we will only discuss the first.

Case 1. (The critical separating line of $Path(p_i, p_j)$ and $Path(q_j, q_k)$ only intersects $Path(p_j, p_k)$ as in Fig. 3a): Let v_3 be the intersection of the critical separating line of $Path(p_i, p_j)$ and $Path(q_j, q_k)$ with the chain $Path(q_j, q_k)$, and let v_1 and v_2 be the intersections of the chains $Path(p_j, p_k)$ and $Path(q_j, q_k)$ with their critical separating line L respectively. It follows from the geometry that the line L must intersect $Path(p_i, p_j)$, and the vertices v_2, v_3 must appear in the shown order on the boundary of $Path(q_j, q_k)$. Therefore the subset of $Path(q_j, q_k)$ joining v_2 and v_3 forms $CPath(e_i, e_k)$.

Case 2. (The critical separating line of $Path(p_i, p_j)$ and $Path(q_j, q_k)$ only intersects $Path(p_j, p_k)$ as in Fig. 3b): Let v_1 and v_2 be the intersections of the chains $Path(q_j, q_k)$ and $Path(p_j, p_k)$ with their critical supporting lines L respectively, and let v_3 be the intersection of the critical separating line of

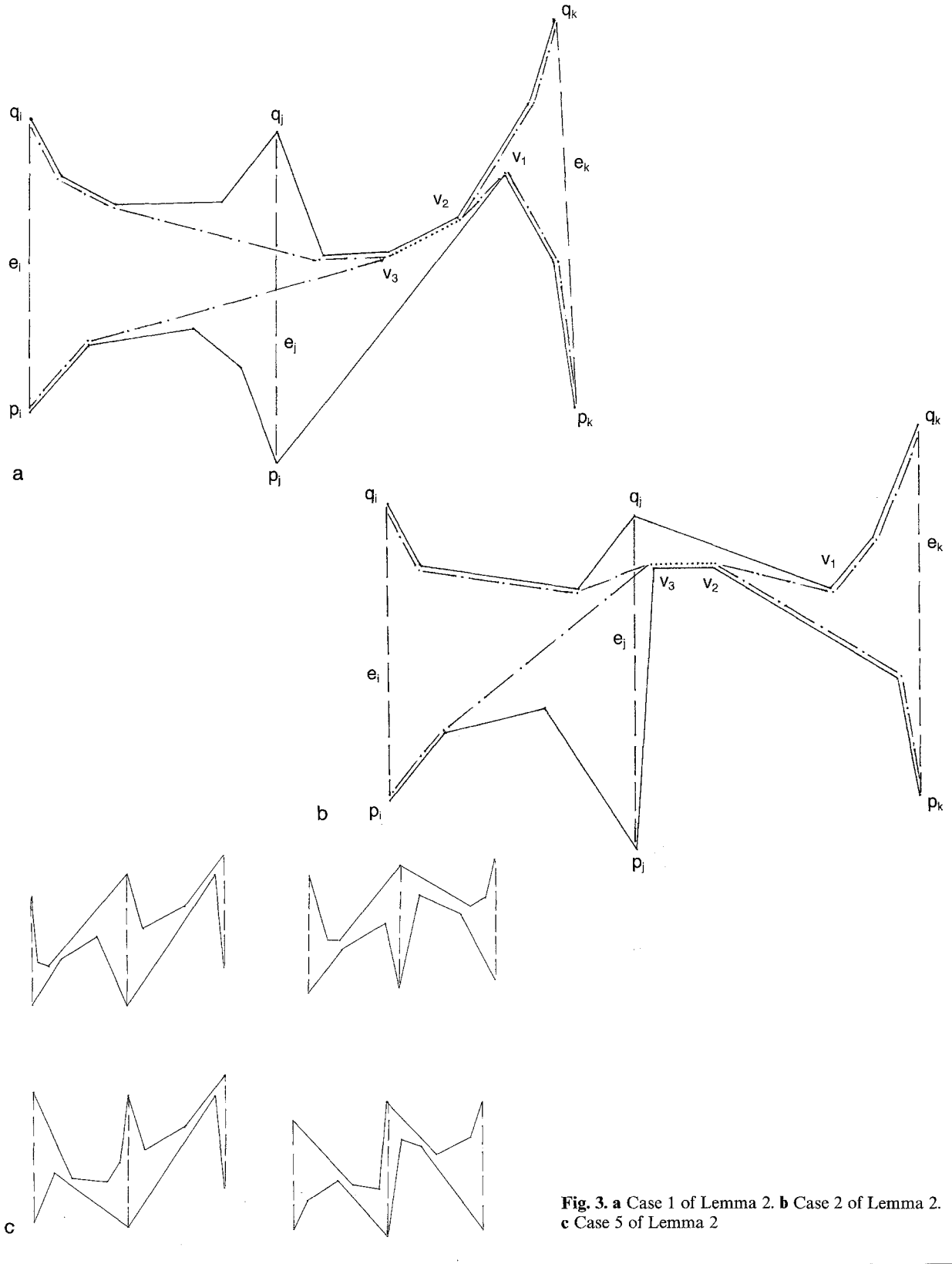

Fig. 3. a Case 1 of Lemma 2. **b** Case 2 of Lemma 2. **c** Case 5 of Lemma 2

$Path(p_j, p_k)$ and $Path(q_i, q_j)$ with the chain $Path(p_j, p_k)$. It follows from the geometry that the line L must intersect $Path(q_i, q_j)$, and the vertices v_2, v_3 must appear in the shown order on the boundary of $Path(p_j, p_k)$. Therefore the subset of $Path(p_j, p_k)$ joining v_2 and v_3 forms $CPath(e_i, e_k)$.

Case 3, 4. (The critical separating line of $Path(p_i, p_j)$ and $Path(q_j, q_k)$ only intersects $Path(q_i, q_j)$): These cases are mirror images of the previous two cases.

Case 5. (The critical separating line of $Path(p_i, p_j)$ and $Path(q_j, q_k)$ intersects both $Path(p_j, p_k)$ and $Path(q_i, q_j)$: Only the four different situations shown in Fig. 3c may occur. It is easy to show that the critical separating and supporting lines between the four convex chains are sufficient to find $CPath(e_i, e_k)$ and to compute the remaining fundamental paths.

Using the algorithms in [4], we can check which of these cases apply and compute the vertices $v_1, v_2, \ldots, v_6$ in $O(1)$ time. We can then construct the fundamental paths between e_i and e_k by "cutting" and "pasting" the appropriate pieces of the old fundamental paths in $O(1)$ time. Thus the lemma follows. □

We now present a detailed description of the algorithm.

Algorithm ShortestPath (P, s, d)

Input: A simple polygon P and two points s and d interior to P.
Output: A list of vertices which form the shortest path from s to d in P.

1st phase:
Step 1.1 Compute a triangulation TP of P. Such process can be performed in $O(\log n)$ time using $O(n)$ processors [9].
Step 1.2 Construct the dual TP' of TP, and let $s'(d')$ denote the node of TP' that corresponds to the triangle containing $s(d)$.
Step 1.3 Determine the path S' in TP' from s' to d', and arrange the set of edges $\{e_1, e_2, \ldots, e_m\}$ which form the corresponding triangulated sleeve in order of increasing distance from s. Such process can be performed in $O(\log n)$ time using $O(n)$ processors [19].

2nd phase:
Step 2.1 Compute the fundamental paths $Path(e_1, e_m)$ as follows:
Step 2.1.1 If $e_1 = e_m$, then return the obvious value.
Step 2.1.2 Find the median diagonal e_k between e_1 and e_m.
Step 2.1.3 Compute $Paths(e_1, e_k)$ and $Paths(e_k, e_m)$ in a recursive fashion.
Step 2.1.4 Merge $Paths(e_1, e_k)$ and $Paths(e_k, e_m)$, using the method described in Lemma 2, to form the fundamental paths $Paths(e_1, e_m)$.
Step 2.2 Merge points s and d with $Paths(e_1, d_m)$, using the method described in Lemma 2, to construct the shortest path between s and d in $O(1)$ time using $O(n)$ processors.

End of ShortestPath

Lemma 3. *The algorithm* **ShortestPath** *correctly computes the interior shortest path between s and d in P.*

Proof. Lee and Preparata [12] proved that the shortest path between s and d is completely contained in the triangulated sleeve S. It follows from the correctness of the algorithms of [9, 19] and from Lemma 2 that the algorithm **ShortestPath** computes the reports correctly the interior shortest path between s and d in P. □

It follows from the triangulation algorithms in [9, 19] that the first phase of the algorithm can completed in $O(\log n)$ time using $O(n)$ processors. The dominant operation of the second phase is merging two sets of fundamental paths to form a new set which can be performed in $O(1)$ time using $O(n)$ processors. Since this operations is performed $O(\log n)$ time, we can now state the main result of this section.

Theorem 1. *Given an n-vertex simple polygon P and two points s and d interior to P we can compute the shortest path between s and d in P in* $O(\log n)$ *time using* $O(n)$ *processors in the* CREW PRAM *model.*

4 All shortest paths from a point

In this section we present a parallel algorithm for computing the shortest path tree from a source

point s to the vertices of a simple polygon P which can be used to compute all the shortest path distances in $O(\log n)$ additional time. Such a tree can be completely represented by the $first(p_i, s)$ for each vertex p_i of P, where $first(p_i, s)$ denotes the vertex in $Path(p_i, s)$ adjacent to p_i. Therefore, the problem we solve in this section is to compute $first(p_i, s)$ for each vertex p_i of P.

The algorithm proceeds in two phases. In the first phase we compute a triangulation TP of P and construct the dual graph TP' of TP, where each node in TP' corresponds to a triangle in TP and two nodes in TP' are adjacent if and only if the corresponding triangles in TP share an edge. TP' must be a tree since P is a simple polygon. We then process TP' so that we can efficiently locate and delete its centroid. The *centroid* is defined as a node v whose deletion divides TP' into subtrees such that the value of $\max_w \{|TP'(v, w)|: w \text{ is adjacent to } v\}$ is minimized, where $TP'(v, w)$ is the subtree that contains w. The size of each subtree is clearly a fraction of $|TP'|$. In the second phase, we employ the divide-and-conquer paradigm to compute the $first(p_i, s)$ for each vertex of P.

Before presenting details of the algorithm, we describe how to process a tree in $O(\log n)$ time so that locating and deleting its centroid can be performed in $O(1)$ time.

Lemma 4. *Let T be a free tree with n nodes and constant degree c. In $O(\log n)$ time using $O(n)$ processors we can assign labels to the vertices of T so that in $O(1)$ time we can find the centroid vertex v of T. Moreover, we can update the labels of the vertices in each subtree in $O(1)$ time so that we can repeat this centroid decomposition procedure in $O(1)$ time using linear number of processors for each subtree in parallel.*

Proof. Let T be a free tree and v be an interior node. The deletion of v decomposes T into a collection of subtrees $\{T(v, w): w \text{ is adjacent to } v\}$, where $T(v, w)$ denotes the subtree that contains w. A centroid of T is defined to be a vertex v which minimizes the value of $C_v = \max_w \{|T(v, w)|: w \text{ is adjacent to } v\}$.

Efficiency of our method for reporting a centroid results from the following two properties of a centroid:

(A) There can be at most two centroids of T, and if so, these two must be adjacent (in effect, an edge is the centroid). We will break ties by defining the centroid to be the vertex with the smaller name.

(B) A vertex v which locally minimizes C_v, relative to all the vertices adjacent to v, must be a centroid. Thus, it is sufficient to compute C_v for every node v in T, since given C_v for each v in T we can test if v is the centroid in $O(1)$ time with one processor by comparing C_v to C_w for each of the (at most c) nodes adjacent to v.

The main idea of our method of computing the C_v values is to use the Euler tour technique [19]. We replace each edge in T by two oppositely directed edges, and let L be the Euler tour of the resulting directed graph, starting at an arbitrary node. We represent L as an array listing the vertices of T as they are visitet by the tour. L can be constructed from T in $O(\log n)$ time using $O(n)$ processors by a simple list ranking procedure, since each vertex appears at most c times in L. We assign the integer "$+1$" to the first occurrence of a vertex v in L and the integer "0" to every other occurrence of v in L after that. We then compute the prefix sum for each element of L, which can be done in $O(\log n)$ time using $O(n)$ processors. Let f_v (resp. l_v) denote the prefix sum of the first (resp. last) occurrence of v in L, for each $v \in T$. Using the values of the prefix sums we can compute C_v as follows:

$$C_v = \max_w \{|T(v, w)|: w \text{ is adjacent to } v\}$$

where

$$|T(v, w)| = \begin{cases} l_w - f_w & \text{if } f_w > f_v \\ |T| - l_w + f_w & \text{if } f_w < f_v \end{cases}$$

Therefore, we can find the centroid of T, denoted by cv, in $O(\log n)$ time using $O(n)$ processors.

Deleting cv decomposes T into subtrees $T_1, T_2, \ldots, T_c$, where T_i contains the node w_i adjacent to cv for $i = 1, \ldots, c$. Also $|T_i| \leq \alpha |T|$ for $i = 1, \ldots, c$, where $\alpha < 1$. We can easily compute the new values of f_v for the vertices in each subtree as follows:

$$f_v = \begin{cases} f_v & \text{if } f_v < f_{cv} \text{ and } l_v < l_{cv} \\ f_v - f_{cv} - 1 & \text{if } f_v > f_{cv} \text{ and } l_v < l_{cv} \\ f_v & \text{if } f_v < f_{cv} \text{ and } l_v > l_{cv} \\ f_v - f_{cv} & \text{if } f_v > f_{cv} \text{ and } l_v > l_{cv} \end{cases}$$

The new values of l_v for the vertices in each subtree are updated using a similar expression.
This can clearly be done in $O(1)$ time using $O(n)$ processors, and allows us to repeat this centroid decomposition in constant time for each T_i using $O(|T_i|)$ processors. □

We now present a detailed description of the algorithm.

Algorithm AllShortestPaths(P, s)

Input: A simple polygon P and a point s interior to P.
Output: $first(p_i, s)$ for each vertex p_i of P.

1st phase:
Step 1.1 Compute a triangulation TP of P. Such process can be performed in $O(\log n)$ time using $O(n)$ processors [9].
Step 1.2 Construct the dual TP' of TP, and let s' denote the node of TP' that corresponds to the triangle containing s.
Step 1.3 Delete s' from TP' (which divides TP' into $TP'_1, \ldots, TP'_c$ with $c \leq 3$). Let $e_j = (p_{j_1}, p_{j_2})$ be the edge shared by the triangle corresponding to s' and the triangulation corresponding to TP'_j with $1 \leq j \leq 3$.
Step 1.4 Process each subtree separately so that we can perform controid location and deletion in $O(1)$ time as shown in Lemma 4.

2nd phase:
Step 2.1 for each of the subtrees $TP'_j (1 \leq j \leq 3)$ compute $first(p_i, p_{j_1})$ and $first(p_i, p_{j_2})$ where $p_i \in P_j$, in parallel as follows:
Step 2.1.1 If the subtree consists of one node, then we return the obvious values.
Step 2.1.2 Locate the centroid cv'_j of TP'_j and perform the centroid decomposition as described in Lemma 4. Let $cv_j = (e_{(j,k)})$ $(1 \leq k \leq 3)$ be the triangle corresponding to the node cv'_j, and let $TP_{(j,k)}$ be the triangulation sharing the edge $e_{(j,k)} (1 \leq k \leq 3)$ with the triangle cv_j. Without loss of generality, we assume that $TP_{(j,1)}$ is the triangulation sharing the edge e_j with the triangle containing s.
Step 2.1.3 Compute $first(p_i, p_{j_1})$ and $first(p_i, p_{j_2})$ (where $p_i \in P_{(j,1)}$) in a recursive fashion, and then construct the set of fundamental paths between $e_{(j,1)}$ and e_j, $Paths(e_j, e_{(j,1)})$.
Step 2.1.4 Compute $first(p_i, p_{(j,k)_1})$ and $first(p_i, p_{(j,k)_2})$ (where $p_i \in P_{(j,k)}$, and $k = 2, 3$) in a recursive fashion.
Step 2.1.5 For each vertex $p_i \in P_{(j,k)}$ $(k = 2, 3)$ we compute $first(p_i, p_{j_1})$ and $first(p_i, p_{j_2})$ by performing four binary searches in $Paths(e_j, e_{(j,1)})$ (one for every fundamental path except $CPaths(e_j, e_{(j,1)})$. The details of the method for deciding which of the tangents to use in updating each *first* label are similar to that used in Lemma 3 and are thus omitted.
Step 2.2 Compute $first(p_i, s)$ for each $p_i \in P$ in parallel. This can be done in $O(1)$ time by observing that if $first(p_i, p_{j_1}) = first(p_i, p_{j_2})$, where $j = 1$, 2, or 3, then $first(p_i, s) = first(p_i, p_{j_1})$; otherwise we can find $first(p_i, s)$ by testing if s is contained in the angle $(first(p_i, p_{j_1}), p_i, first(p_i, p_{j_2}))$.

End of All Shortest Paths

It follows from Lemma 4 and the triangulation algorithm of Goodrich [9] that the first phase of the algorithm can completed in $O(\log n)$ time using $O(n)$ processors. The dominant operations of the second phase are: (A) constructing the set of fundamental paths which can be performed in $O(\log n)$ time using $O(|P_1|)$ processors by a straightforward doubling procedure, and (B) updating *first* label for each vertex in $P_{(j,k)} (k = 2, 3)$ which can be done in $O(\log n)$ time using $O(|P_{(j,2)}| + |P_{(j,3)}|)$ processors. Since these operations are performed $O(\log n)$ time, we can now state the main result of this section.

Theorem 2. *Given an* n-vertex *simple polygon* P *and a point* s *inside* P *we can find* $first(p_i, s)$ *for each vertex* $p_i \in P$ *in* $O(\log^2 n)$ *time using* $O(n)$ *processors in the CREW PRAM model.*

5 Concluding remarks

We have shown that the ISP and AISP problems belong to the class of **NC** problems (i.e., we can compute the shortest path(s) in *polylog* time using *polynomial* number of processors assuming a CREW PRAM model of computation.) Specifi-

cally, we give an algorithm for the ISP problem which runs in $O(\log n)$ time using $O(n)$ processors and an algorithm for the AISP problem which runs in $O(\log^2 n)$ time using $O(n)$ processors. It is interesting to note that, once we have computed for each vertex p_i in P the first vertex of P on the shortest path from p_i to the source s (in the AISP problem), we can then compute all the shortest path distances in $O(\log n)$ time using $O(n)$ processors by a straightforward doubling procedure.

The parallel algorithms presented in the previous sections do not provide an optimal speed-up when compared to the sequential algorithms in [5, 7, 12] where the shortest path is computed in $O(n)$ time after an $O(n \text{ loglog } n)$ preprocessing time. A natural challenge is to develop a parallel algorithm with an optimal speed-up. An interesting related problem is to show that reporting the shortest path beween two points in the presence of obstacles also belongs to the class of **NC** problems.

References

1. Aggarwal A, Chazelle B, Guibas L, O'Dunlaing C, Yap C (1985) Parallel computational geometry. Proc 25th Ann Symp Foundat Comput Sci, pp 468–477
2. Atallah MJ, Cole R, Goodrich MT (1987) Cascading divide-and-conquer: a technique for designing parallel algorithms. Proc 28th Ann Symp Foundat Comput Sci, pp 151–160
3. Atallah MJ, Goodrich MT (1988) Efficient parallel solutions to some geometric problems. J Parallel Distrib Comput 3:492–507
4. Atallah MJ, Goodrich MT (1986) Parallel algorithms for some functions of two convex polygons. 24th Ann Allerton Conf Commun Control Comput, Urbana-Champaign (October 1986), pp 758–767
5. Chazelle B (1982) A theorem on polygon cutting with applications. Proc 23rd Ann Symp Foundat Comput Sci, pp 339–349
6. Chow A (1980) Parallel algorithms for geometric problems. PhD Diss, Comput Sci Dept, Univ Illinois at Urbana-Champaign
7. ElGindy H (1985) Hierarchical decomposition of polygons with applications. PhD Diss, School Comput Sci, McGill Univ (June 1985)
8. ElGindy H (1986) A parallel algorithm for the shortest path problem in monotone polygons. Tech Rep MS-CIS-86-49, Dept Comput Inf Sci, School Eng Appl Sci, Univ Pennsylvania (June 1986)
9. Goodrich MT (1988) Triangulation a simple polygon in parallel. J Algorithms (in press)
10. Guibas L, Hershberger J, Leven D, Sharir M, Tarjan R (1986) Linear time algorithms for visibility and shortest path problems inside simple polygons. Proc 2nd Symp Comput Geom, York Heights (June 1986), pp 1–13
11. Lee DT, Preparata FP (1984) Euclidean shortest paths in the presence of rectilinear barriers. Networks 14:393–410
12. Lee DT, Preparata FP (1984) Computational geometry – a survey. IEEE Trans Comput C-33:872–1101
13. Lozano-Perez T, Wesley MA (1979) An algorithm for planning collision-free paths among polyhedral obstacles. Commun ACM 22:560–570
14. Pan V, Reif J (1985) Efficient parallel solution of linear systems. Proc 17th ACM Symp Theor Comput, pp 143–152
15. Preparata FP, Shamos MI (1985) Computational geometry: an introduction. Springer, Berlin Heidelberg New York
16. Quinn MJ, Deo N (1984) Parallel graph algorithms. Comput Surv 16:319–346
17. Savage C (1984) Parallel algorithms for graph theoretic problems. PhD Diss, Math Dept, Univ Illinois, Urbana, III
18. Tarjan RE, Van Wyk CJ (1986) An $O(n \text{ loglog } n)$-time algorithm for triangulating simple polygons. Tech Rep CS-TR-052-86, Dept Comput Sci, Princeton Univ
19. Tarjan RE, Vishkin U (1985) An efficient parallel biconnectivity algorithm. SIAM J Comput 14:862–874
20. Welzl E (1985) Constructing the visibility graph for n line segments in $O(n^2)$ time. Inf Proc Lett 20:167–171