PARALLEL ALGORITHMS FOR RECTILINEAR LINK DISTANCE PROBLEMS

Andrzej Lingas, Anil Maheshwari and
Jörg-Rüdiger Sack

TR-213, SEPTEMBER 1992

School of Computer Science, Carleton University
Ottawa, Canada, KIS 5B6

Parallel Algorithms for Rectilinear Link Distance Problems

Andrzej Lingas
Department of Computer Science
Lund University
Box 118, S-22100 Lund, Sweden
andrzej@dna.lth.se

Anil Maheshwari
Computer Systems and Communications Group
Tata Institute of Fundamental Research
Homi Bhabha Road, Bombay - 400 005, India
manil@tifrvax.bitnet

Jörg-Rüdiger Sack
School of Computer Science
Carleton University
Ottawa, Ontario K1S 5B6, Canada
sack@scs.carleton.ca

Abstract

We provide optimal parallel solutions to several link distance problems set in trapezoided rectilinear polygons. All parallel algorithms are deterministic and designed to run on the exclusive read exclusive write parallel random- access machine (EREW PRAM). Let P be a trapezoided rectilinear simple polygon with n vertices. In $O(\log n)$ time using $O(n/\log n)$ processors we can optimally compute

1. minimum rectilinear link paths, or shortest paths in the L_1 metric from any point in P to all vertices of P,
2. minimum rectilinear link paths from any segment inside P to all vertices of P,
3. the rectilinear window (histogram) partition of P,
4. both covering radii and vertex intervals for any diagonal of P,
5. a data structure to support rectilinear link distance queries between any two points in P (queries can be answered optimally in $O(\log n)$ time by a uniprocessor).

Our solution to 5 is based on a new linear-time sequential algorithm for this problem which is also provided here. This improves on the previously best known sequential algorithm for this problem which used $O(n \log n)$ time and space. We develop techniques for solving link distance problems in parallel which are expected to find applications in the design of other parallel computational geometry algorithms. We employ these parallel techniques for example to almost optimally compute the link diameter, the link center and the central diagonal of a rectilinear polygon.

1 Introduction

The *link distance* between two points s and t inside a polygon P is the minimum number of segments (straight edges) required to connect s and t inside P. The link distance is an appropriate distance measure in environments such as motion planning, broadcasting transmission, or VLSI,

where making a turn is more expensive than moving along a straight-line motion (see [28]). The study of link distance problems has recently attracted a lot of attention in computation geometry, see e.g. [2, 3, 6, 10, 11, 12, 17, 20, 24, 25, 28, 29]. Many of these sequential algorithms run in linear time in triangulated polygons. Combined with the recent triangulation algorithm by Chazelle [5] these algorithms are now optimal.

The very recent parallel triangulation algorithm by Clarkson et al. [7], and Goodrich [13] have intensified the need for parallel algorithms which are optimal after triangulation or trapezoidal decomposition. For the Euclidean distance measure now optimal parallel algorithms have been developed for a variety of computational geometry problem set in triangulated polygons [13, 15]. These problems include the parallel construction of data structures for answering shortest path queries or shooting queries, solving visibility problems, constructing shortest paths trees, and relative convex hulls. At present no parallel algorithm is known for solving non-trivial link distance problems optimally in triangulated polygons. An efficient parallel algorithm for computing a minimum link path between two vertices in a simple polygon is due to Chandru *et al.* [6]. Their algorithm runs in $O(\log n \log\log n)$ time using $O(n)$ processors on the CREW-PRAM, where n is the number of vertices in the input polygon. Ghosh and Maheshwa developed a link center algorithm which runs in $O(\log^2 n \log\log n)$ time using $O(n^2)$ processors [12]. For details on PRAM models, see [18].

Even in the sequential setting link distance related problems seem to be more difficult to solve than the corresponding problems using the geodesic distance measure. The difficulties stem from the fact that several minimum link paths may exist connecting a given pair of vertices, while the minimum geodesic path is always unique.

In this paper we present optimal parallel algorithms for a variety of link distance problems set in trapezoided rectilinear polygons [23, 25]. A *rectilinear polygon* * is one whose edges are all aligned with a pair of orthogonal coordinate axes, which we take to be horizontal and vertical without loss of generality. Rectilinear polygons are commonly used as approximations to arbitrary simple polygons; and they arise naturally in domains dominated by Cartesian coordinates, such as raster graphics, VLSI design, robotic, or architecture. A rectilinear polygon is called *trapezoided* if both its vertical and horizontal visibility maps are given (see [G92, GSG90] for trapezodiation in parallel). Some of the problems discussed in this paper either explicitly or implicitly deal with the construction of one or more rectilinear paths. A (simple) *rectilinear path* inside a rectilinear polygon P is a simple path inside P that consists of *axis-parallel* (or orthogonal) segments only and does not cross itself. The *rectilinear link distance* between two points in P is defined as the minimum number of segments of any rectilinear path connecting the two points. The corresponding path is defined as *minimum rectilinear link path* and its computation arises e.g. in robotics and VLSI problems.

All our main algorithms are deterministic, run in $O(\log n)$ time and have an optimal time-processor product; they are designed for a PRAM of the exclusive-read exclusive write (EREW) variety. In particular we have solved:

Minimum rectilinear link paths: The sequential computation of rectilinear link paths has received considerable attention in computational geometry; see for example [3, 8, 9, 25]. de Berg [3] proposed an optimal sequential algorithm for computing a minimum rectilinear link path between two vertices in a rectilinear polygon P. Previous to our result, the only known method for computing a minimum link path in a rectilinear polygon in parallel was to compute

*Rectilinear polygons have also been called *orthogonal polygons*, *isothetic polygons* and *rectanguloid polygons* in the literature.

a shortest path between two nodes of a graph defined over the rectilinear polygon. This requires the parallel computation of the transitive closure of the graph using at least a quadratic number of processors.

We first present a simple algorithm to compute a minimum rectilinear link path between two vertices. Our approach is based on polygon partitioning. The minimum link path produced by our algorithm is shown to be a shortest path in the L_1-metric and both are computed in $O(\log n)$ time using $O(n/\log n)$ processors which is optimal after trapezoidation. A more challenging problem which has been well-studied in computational geometry is the determination of distances (in e.g. the Euclidean or link metric) from a point or vertex to all vertices inside a polygon [29, 30]. In the context of link distance problems the study is motivated for example as follows. Suppose that a broadcast station is placed at some point inside a polygonally bounded domain; each vertex represents a location and must be reached by a signal originating from the broadcast station. The objective is to determine the total number of retransmissions necessary to reach each location. We give an optimal algorithm to compute the rectilinear link distance from a point to each vertex of a rectilinear polygon. Our algorithm can also be used to solve the above broadcasting problem for rectilinear domains. Our result also implies an optimal algorithm for computing the shortest path from a point to all vertices in the L_1 metric.

To solve this and other rectilinear link distance problems we require the information about the rectilinear link distances and the rectilinear link paths from a diagonal or from segment in P to all vertices of P. We show that these minimum link distance problems can be solved in $O(\log n)$ time using $O(n/\log n)$ processors. The required information is provided by the rectilinear window (or histogram) partition from a diagonal of P introduced next. The problem of computing the link distance from a diagonal or segment to all vertices of P may also find applications. For example, assume that a robot is mounted on a track and that its arm is built out of telescopic links. The question of how many rectilinear links the robot must have to reach all vertices can be solved using our result.

Rectilinear window (or histogram) partition: A fundamental tool used for solving a number of link distance problems is the window partition developed by Suri [29]. Its analogon for rectilinear polygons is the rectilinear window partition or histogram partition introduced by Levcopoulos [21] who used it in the design of approximation algorithms of optimal polygon decompositions. In the sequential setting window and histogram partitions have been used to efficiently solve a variety of problems including link path computation and link distance queries [28, 29], the link center [10], central link segment problem [1], and the construction of bounded Voronoi diagrams [19]. For the parallel setting we provide an optimal algorithm for determining a histogram partition from any segment in P. Thus our method might be used to parallelize several known interesting sequential algorithms. We use this tool e.g. to compute the link diameter, answer rectilinear link distance queries, and for finding covering radius and intervals of segments as discussed next.

Segment covering radius and vertex intervals: Let d be a segment joining two boundary points of P. The *covering radius* of d is the value which minimizes the maximum rectilinear link distance from a point on d to each point in P. The covering radius is realized between a point on d and a vertex of P; its optimal parallel computation is described in this paper. The rectilinear link distance from a vertex v of P to d is the minimum rectilinear link distance from v to any point on d. In general, this distance is realized to more than one point on d; the set of all such points on d form a consecutive interval called the *vertex interval on d*. These intervals were instrumental in de Berg's [3] link diameter algorithm and rectilinear link distance query

algorithm. We give an optimal parallel algorithm for computing the vertex intervals on d for all vertices of P. We employ this algorithm to construct a data structure for answering link queries, finding the link diameter of a rectilinear polygon, and to compute the vertex-to-vertex link distances already mentioned.

Parallel construction of a data structure for rectilinear link distance queries: de Berg [3] gave an $O(n \log n)$ sequential-time and $O(n \log n)$ space algorithm for constructing a data structure to support rectilinear link distance queries between any two points in P. We describe an optimal parallel algorithm whose total work is $O(n)$ (queries can be answered optimally in $O(\log n)$ time by a uniprocessor). Our parallel algorithm therefore implies a linear-time sequential algorithm for solving this problem which improves on the $O(n \log n)$ time bound established by de Berg. [†] Link distance queries find applications e.g. in placements of mobile units or robots. Suppose that a constant number of mobile units are operating in a rectilinearly bounded domain. Mobile units are assumed to take significantly longer to turn than to move along a straight line. In case of an emergency encountered at some location (point) in the domain the unit having the shortest link distance is to be dispatched. The problem can be solved by using a constant number of link distance queries posed to our data structure.

Using the algorithms developed in this paper we can provide almost optimal solutions to the following problems:

Link diameter: The *link diameter* of a rectilinear polygon P is the maximum rectilinear link distance between any pair of points in P. It is realized between a pair of vertices of P. Knowledge of the link diameter helps to determine the worst constellation of a placement a mobile unit can have with respect to the location of an emergency. The link diameter is also instrumental in finding the link center and the link radius of polygons [20, 10, 17, 22]. Nilsson and Schuierer [22] gave a linear-time algorithm for finding the link diameter thus improving on an earlier result of de Berg. Using the techniques developed in this paper we can develop an almost optimal parallel implementation of their algorithm. The algorithm takes $O(\log n \log\log n)$ time and performs $O(n \log\log n)$ work. In addition to reporting the value of the diameter our algorithm reports a pair of vertices and a link path connecting them whose link distance is the diameter. The analysis of the algorithm leads to an interesting recurrence relation.

Link center, link radius, and central diagonal: An interesting geometrical min-max problem is to determine the set of points x in a polygon P at which the maximum link distance from x to any other point in P is minimized. The set of points x is called the link center; its determination has been studied in [20, 22, 10, 12, 17]. Most algorithms for computing the link center report as a by-product the value of the *link radius* which is the maximum link distance from a point in the link center to all points in P. To efficiently compute the link center of simple polygon Djidjev et al. [10] introduced the concept of a central diagonal of a simple polygon; subsequently termed *splitting chord* by [22] in the context of rectilinear polygons. Our results are $O(\log n \log\log n)$ time and $O(n \log\log n)$ work algorithms for the problems of computing the rectilinear link center, link radius, and central diagonal of a rectilinear polygon.

We use the following tools previously developed in parallel computing: Lowest common ancestor in a tree [26], tree operations including the Euler tour technique [32]. tree contraction and traversals [18], point location in planar subdivision [31], parenthesis matching and nearest smallers [4]. For the other tools such as parallel prefix, list ranking and doubling, see [16, 18].

[†]Independently, a linear-time algorithm has been discovered by Schuierer [27].

The paper is organized as follows: in Section 2 we introduce some notation and state some preliminaries. In Section 3 we present a simple optimal parallel algorithm for computing a minimum link path and a shortest L_1 path between two points inside a rectilinear polygon. In Section 4 we describe our parallel algorithms for determining the link distances from a diagonal (or segment) to all other diagonals of a rectilinear polygon and from a diagonal to all vertices. These algorithms are used for generalizing the algorithm of Section 3 to that of computing link distances from a point to all vertices. In Section 5 we describe the construction of a rectilinear window partition. The parallel and sequential construction of a data structure for answering point-to-point link distance queries is provided in Section 6. In Section 7 we discuss the parallel determination of the link diameter. In Section 8 we discuss an almost optimal parallel algorithm for computing the link center, link radius, and a central diagonal. In Section 9 we summarize the results obtained in this paper and discuss a few open problems.

2 Preliminaries

Throughout, all geometric objects (polygons, paths, boundary, distances, etc.) are implicitly assumed to be rectilinear (i.e., each of their constituent segments is parallel to one of the coordinate axes). We assume that the simple rectilinear polygon P is given as a clockwise sequence of vertices p_1, p_2, ..., p_n with their respective x and y coordinates. The symbol P is used to denote the (closed) region of the plane enclosed by P. Two points of P are said to be *visible* if the line segment joining them lies totally inside P. A polygon P is a *staircase polygon* if there are vertices p_i and p_j such that $bd(p_i, p_j)$ and $bd(p_j, p_i)$ are staircases. A vertex u of P is *reflex* if the internal angle at that vertex is greater than 180°, *convex* otherwise.

As a preprocessing step, in this paper, we require the horizontal and vertical visibility maps. By horizontal and vertical visibility maps we mean that each edge is extended (possibly to both sides) towards the polygon interior until the boundary of polygon reached. These extensions can be computed by the algorithm of Goodrich [13] and Goodrich *et al.* [14]. We insert the extension points of each edge as vertices on the boundary of polygon. Note that the number of new vertices introduced on the boundary is linear. From now onwards we assume that both horizontal and vertical visibility maps are provided as a part of the input. For simplicity we refer to a rectilinear polygon together with its visibility map as a *trapezoided rectilinear polygon.*

3 L_1-Shortest paths and minimum link paths

In this section present an optimal parallel algorithm for computing an L_1-shortest path and a minimum link path between two points s and t in a rectilinear simple polygon P. Let $SP(s,t)$ and $MLP(s,t)$ denote L_1-shortest path and minimum link path between s and t, respectively. Note that $SP(s,t)$ and $MLP(s,t)$ in a rectilinear polygon are not unique. We can easily observe that $MLP(s,t)$ computed by our algorithm is also $SP(s,t)$. Therefore, we only present a parallel algorithm for computing $MLP(s,t)$.

The overview of the algorithm is as follows. Using the trapezoidation of P we construct a subpolygon P' of P such that there exist a $MLP(s,t)$ which is completely contained inside P'. The region $P - P'$ is called *redundant.* The region P' is obtained by performing the parallel prefix on the weights associated with each vertex on $bd(P)$, where a vertex is assigned a weight

from the set $\{-1, 0, 1\}$. The subpolygon P' has a simple structure and minimum link path in P' is computed by an easily parallelizable greedy approach.

In the following lemma we characterize those subpolygons of P which are redundant for the computation of $MLP(s,t)$. A subpolygon is a *horizontal redundant polygon* if it is formed due to a horizontal visible pair, otherwise it is a *vertical redundant polygon.*

Lemma 3.1 *Let vv' be a diagonal inside P. If s and t do not belong to $bd(v,v')$ then there exist a $MLP(s,t)$ that does not contain any point of the subpolygon formed by $bd(v,v')$ and the segment vv', except possibly points of segment vv'.*

Proof: Without loss of generality assume that $v \in bd(s,v')$. Let R denote the subpolygon formed due to $bd(v,v')$ and the segment vv'. Assume that a part of $MLP(s,t)$ lies inside R. Let x be the first point encountered on $bd(R)$ which intersects $MLP(s,t)$ while traversing $MLP(s,t)$ from s to t. Note that $x \in vv'$. Since s and t do not belong to R, there exist a point $y \in MLP(s,t)$, $y \neq x$, which lies on vv'. We construct a new link path L from $MLP(s,t)$ which consists of atmost as many links as $MLP(s,t)$ by composing the following three : (1) link path from s to x (2) the segment xy and (3) link path from y to t. Note that the part of $MLP(s,t)$ inside the subpolygon R between x and y has been replaced by the segment xy in L. Now repeat the above procedure with L. Eventually, a link path is obtained which does not lie inside R and has atmost as many links as $MLP(s,t)$. □

The above lemma suggests a procedure for removing redundant polygons. First remove horizontal redundant polygons and then remove vertical redundant subpolygons. In the following we state the procedure for removing horizontal redundant polygons. The procedure for removing vertical redundant polygons is analogous. We describe the procedure for removing horizontal redundant subpolygons for $bd(s,t)$. The procedure for $bd(t,s)$ is analogous.

Let vv' be a horizontal diagonal and let $bd(v,v')$ does not contain s and t. Assign a weight of +1 to v and -1 to v' if $v \in bd(s,v')$. If $v' \in bd(s,v)$ then assign a weight of +1 to v' and -1 to v. The remaining vertices gets 0 as thier weight. The following lemma characterizes those vertices of $bd(s,t)$ which do not belong to any horizontal redundant polygons.

Lemma 3.2 *A vertex v of P does not belong to any horizontal redundant polygon if and only if the sum of the weights of the vertices on $bd(s,v)$ is 0.*

Proof: Consider any two horizontal redundant subpolygons P_x and P_w formed due to the horizontal diagonals xx' and ww', respectively. Observe that either $P_x \subset P_w$, or $P_w \subset P_x$ or $P_x \cap P_w = \emptyset$. Hence, the sum of weights for any vertex v on $bd(s,v) \geq 0$. If the sum of weights of vertices on $bd(s,v)$ for some vertex v is nonzero, then there exist a horizontal diagonal ww' such that $v \in bd(w,w')$ and the subpolygon P_w due to ww' is a horizontal redundant subpolygon. □

The above lemma suggests that by performing a parallel prefix computation we can remove the horizontal redundant polygons on $bd(s,t)$. Analogously, we can remove the horizontal redundant polygon on $bd(t,s)$ by choosing the weights appropriately. In the following lemma we characterize the structure of the polygon P' obtained after removing the redundant polygons.

Lemma 3.3 *The dual of the vertical and horizontal trapezoidation of P' is a path.*

Proof: Assume that the dual of the horizontal trapezoidation is not a path. Let T_s and T_t be the nodes in the tree, corresponding to the trapezoids s and t, respectively. Assume that the node T_v is not in the path from T_s and T_t. Observe that the trapezoid corresponding to T_v does not contain s and t and thus its a redundant horizontal trapezoid. A contradiction. □

The subpolygon P' will be either a staircase polygon or will be a composition of several staircase polygons. These staircase polygons will be separated by the edges with double reflex vertices, called as *eaves* [11]. In the following lemma we present an important property of the *eaves* and minimum link path in P'.

Lemma 3.4 *There exist a minimum link path between s and t that contains all eaves of the polygon P'.*

Proof: The proof is similar to the proof of Lemma 3.1. □

The above lemma suggests the following procedure for computing $MLP(s,t)$.

1. Extend each eave from both ends to $bd(P')$. The extension of eaves partitions P' into staircase polygons.
2. Find a minimum link path connecting the extensions of every pair of consecutive eaves to form $MLP(s,t)$.

Consider one such staircase polygon P'' obtained as above. The problem reduces to that of computing minimum link path connecting the two extensions of eaves vv' and ww' in P''. The following greedy procedure computes minimum link path connecting these two extensions in P''. Note that minimum link path consists of only horizontal and vertical segments. Start from the endpoint of vv' which is closest to ww'. Let it be v'. Shoot a ray in the direction perpendicular to vv' towards ww' from $z_1 = v'$. Observe that this ray intersects $bd(P'')$ exactly at one point and let that point be z_2. If $z_2 \in ww'$ then we have already computed the desired minimum link path. Otherwise, shoot a ray from z_2 now in the orthogonal direction towards ww' and let it intersect $bd(P'')$ at z_3. If $z_3 \in ww'$ then we are done otherwise repeat the above procedure to obtain z_4, z_5, ..., z_k such that $z_k \in ww'$. We call the path z_1z_2, z_2z_3, ..., $z_{k-1}z_k$ as a *greedy path* and z_1, z_2, ..., z_k as its turning points. In the following lemma we show that the greedy path computed as above is a minimum link path.

Lemma 3.5 *The greedy path is a minimum link path connecting the two extensions of eaves in a staircase polygon.*

Proof: Trivial and omitted. □

The greedy path can be computed in parallel as follows. We know that the turning points of the greedy path lie on the edges of P''. Further, for each edge e of P" we know that if a turning point lies on e then which will be the next edge in the path. For each edge of P'', assign a pointer to the edge where the next turning point lies. Since, a staircase polygon is $X - Y$ monotone, for each edge of P'' there is a unique pointer to the next edge. These pointers form a rooted tree T, where the nodes in T are the edges of P'' and two nodes u and v in T are

adjacent if there is a pointer from the edge corresponding to u to the edge corresponding to v in P''. The problem of computing greedy path reduces to that of finding a path in T from a leaf node to the root. Since paths in T can be computed using a linear number of operations in parallel [16], the greedy path can be computed optimally in parallel. We summarize the result in the following theorem.

Theorem 3.6 *A minimum link path between two given points in an n-vertex trapezoided rectilinear simple polygon can be computed in optimal $O(\log n)$ time using $O(n/\log n)$ processors on EREW PRAM.*

Proof: The correctness of the algorithm follows from Lemma 3.4 and 3.5. The complexity of the algorithm follows from the fact that prefix sums and paths in the tree can be computed in $O(\log n)$ time using a linear number of operations [18, 16]. □

Corollary 3.7 *The minimum link path computed, as above, between two points s and t in a rectilinear polygon is an L_1-shortest path between them.*

4 Optimal parallel algorithms for computing link distances

In this section we generalize the results obtained in Section 3. We present optimal parallel algorithms to compute link distances from a horizontal or vertical segment d (or a diagonal) within a rectilinear polygon P to all vertices of P and from a point (or a vertex) to all vertices of P. A *diagonal*(a, b) in P is a horizontal or vertical closed straight-line segment within P joining a point a on an edge e_i to a vertex of an edge e_j, where e_i and e_j are neither equal nor incident to each other. A diagonal is *maximal* if it is not properly contained in any other diagonal. Analogously, a diagonal is a *minimal diagonal* if it does not contain properly any other diagonal. The *link distance* from a diagonal d to a vertex v is defined as the minimum among the link distances from x to v, where $x \in d$. The *link distance* between two straight-line segments d and d' is defined as the minimum over link distances between x and y, where $x \in d$ and $y \in d'$ and is denoted by $LD(d, d')$.

4.1 Link distances from a minimal diagonal to all maximal diagonals

We may assume without loss of generality that the minimal diagonal d is horizontal. It splits P into two subpolygons : the top subpolygon and the bottom subpolygon. Consider the horizontal trapezoidation of the bottom subpolygon and the tree T dual to it. Root the tree at the trapezoid bounded by d. Now consider all the maximal horizontal diagonals within the subpolygon. The tree T induces the rooted tree U on the set of the maximal horizontal diagonals as follows: if the trapezoid u is the parent of the trapezoid t in T, then the maximal horizontal diagonal that separates u from t is the parent in U of the other maximal horizontal diagonal edging t. In this section first we present an optimal parallel algorithm for computing the link distance from d to all the horizontal maximal diagonals in the bottom subpolygon. Then using this information we compute the link distance from d to all vertices in the subpolygon. Algorithm 1 computes the link distance from d to all maximal horizontal diagonals of P. In the following theorem we show that Algorithm 1 correctly computes the link distance between the diagonals and it runs in $O(\log n)$-time using $O(n/\log n)$ processors.

1. Mark all diagonals that lie within the bottom polygon.
2. Compute the dual tree T of the horizontal trapezoidation of the bottom polygon.
3. Root T at the trapezoid incident to d.
4. Compute the rooted tree U.
5. For each non-root maximal diagonal u in U compute its furthest ancestor $f(u)$ in U that is the visible from it.
6. Compute the directed tree U' on the set of maximal diagonals in U such that u is a child of $f(u)$ if $f(u)$ is defined otherwise u is a child of the root diagonal d.
7. For each diagonal u in U' compute its distance $d(u)$ to the root diagonal d in of U' (i.e. the number of edges on the path to the root).
8. For each diagonal u in U define its link distance $md(u)$ to the root diagonal d as follows: If $f(u)$ is defined then $md(u) := 2d(f(u)) - 1$ else $md(u) = 1$.
9. Repeat Steps 1-8 for the top polygon in place of the bottom polygon.

Algorithm 1: Algorithm for computing the link distances from d to all maximal horizontal diagonals

Theorem 4.1 *Let d be a minimal horizontal diagonal in a (trapezoided) rectilinear simple polygon P. Algorithm 1 computes for all maximal horizontal diagonals of P their minimum link distance to d within P in $O(\log n)$-time using $O(n/\log n)$ EREW PRAM processors.*

Proof: The correctness of Algorithm 1 follows by induction on the value of $md(u)$. If $md(u) = 1$ then the link distance from u to d is indeed 1. Suppose $md(u) > 1$. Assume inductively that the link distance from $f(u)$ to d is $md(f(u))$. Any link path R from u to d within P has to vertically cross $f(u)$ or to have a vertical link starting from $f(u)$ towards d in U. To achieve the crossing point or the vertical link, starting from u, R needs exactly two links by the definition of $f(u)$. Hence, the total length $L(u)$ of R is $2+L(f(u))$ which by induction is 2 + 2(d(f(u))-1 which is $2d(u) - 1$ by $d(f(u)) = d(u) - 1$.

It remains to be shown that each step of the above algorithm can be implemented within the bounds claimed in the theorem thesis.

Step 1: Knowing for each horizontal diagonal e of P (in particular d) the vertex and the edge of P bridged by e and assuming that the vertices of P are numbered with consecutive integers $1, 2, ...n$, we can mark all minimal diagonals within the bottom polygon in $O(\log n)$-time using $O(n/\log n)$ processors.

Step 2: We build the tree T by assigning to each marked diagonal a single processor. First the processor checks whether its associated diagonal is the upper leftmost diagonal of P in a trapezoid in constant time. If so the name of the diagonal is distributed to all the diagonals on the boundary of the trapezoid as the identification of the trapezoid. It can be done

in logarithmic time with $O(n/\log n)$ processors by using parallel list ranking. Now, for each of the two trapezoids t adjacent to a diagonal its processor finds the next clockwise trapezoid adjacent to t and links it with the other trapezoid adjacent to the diagonal. In this way a circular list of trapezoids adjacent to t is created in constant time (or in logarithmic time using $O(n/\log n)$ processors by Brent's principle [18]).

Step 3: The tree T can be rooted at the trapezoid edged by d by using the Euler tour technique of [32]. It can be done optimally in logarithmic time [32].

Step 4: Again using the Euler tour technique, we number the trapezoids in T in preorder. To each minimal diagonal (i.e. marked diagonal), we assign the pair of preorder numbers of the adjacent trapezoids. Next, using the horizontal trapezoidation, we form maximal alternating chains (lists) of incident horizontal edges and minimal horizontal diagonals in constant time using a linear number of processors (or in logarithmic time using $O(n/\log n)$ processors by Brent's principle). Observe that each such a list (chain) can be identified with a maximal horizontal diagonal u. By using optimal list ranking, we assign the lowest preorder number $lp(u)$ of the trapezoids adjacent to the minimal diagonals included by u to u. For convention, $lp(d) = 0$. We specially mark all the preorder numbers that have been selected as the identifiers of maximal diagonals. Now, we can identify the marked preorder numbers of trapezoids adjacent to u that are different from $lp(u)$ as the children of $lp(u)$ in U. Thus the entire step can be done in logarithmic time using $O(n/\log n)$ processors by Brent's principle.

Step 5: The given vertical trapezoidation divides the horizontal edges of P into a linear number of maximal edge pieces whose insides are free from the vertical projection of vertices of P. For all maximal horizontal diagonals u, we form a list of such pieces covered by the alternating chain it corresponds to (see Step 4) in constant time using a linear number of processors. Also, for all pieces p, we compute the vertically opposite pieces $op(p)$ in constant time using a linear number of processors. By Brent's principle, the two above steps can be implemented in logarithmic time using $O(n/\log n)$ processors. By applying parallel list ranking to the lists of the pieces, we can assign to each of them the maximal diagonal u it belongs to. Further, for each piece p covered by a maximal diagonal u we find the lowest common ancestor $lca(p)$ of u and the maximal diagonal including $op(p)$ on the way to the root of U. Using Schieber and Vishkin's [26] parallel algorithm for the lowest common ancestor queries it can be done in logarithmic time with optimal number of processors. Next, for all maximal diagonals u, we compute the maximal diagonal $f^*(u)$ that is the most remote vertically $lca(p)$ where p is covered by u. $f^*(u)$ is our preliminary candidate for $f(u)$. By a standard processor-optimal technique for computing maximum in logarithmic time it can be done in logarithmic time using $O(n/\log n)$ processors. Now, we inductively define $f(u)$ as the most remote vertically maximal diagonal among $f^*(u)$ and the $f(c)$'s for children c of u. Note that this reduces the computation of $f(u)$ to finding a minimum of single values given by children and a single precomputed value at the node u. Therefore, we can use the tree contraction technique here and compute the required information by performing the work in logarithmic time using $O(n/\log n)$ processors [16, 18].

Step 6: The edges of U' are given by the pointers from u to $f(u)$.

Step 7: The distances $d(u)$ can be computed in logarithmic time optimally by using the Euler tour technique [32].

Step 8: This step takes constant time and $O(n)$ processors. Hence, it can be performed in logarithmic time with $O(n/\log n)$ processors by the Brent's principle.

4.2 Link distances to all vertices

Using the results of Theorem 4.1, we first present an algorithm for computing the link distances from vertices v to a horizontal diagonal d in P. Consider a maximal horizontal diagonal e including v. Next, let p_v be the maximal piece of e that belongs to the perimeter of P, includes v and it is free from vertical vertex projections (see Step 5 of Algorithm 1). Let $lca(p_v)$ be the lowest common ancestor of e and the maximal diagonal e' including the opposite piece $op(p_v)$ in the tree U (see Step 2 and 5 of Algorithm 1). We may assume w.l.o.g that $op(p_v)$ does not belong to d. Note that $lca(p_v)$ is the most remote vertically maximal diagonal where the first turning point of a minimum link path to d starting vertically from v could occur.

Any link path from v to d has to cross e'. By the definition of e', if it starts vertically from v it needs exactly two links to reach any point from where e' achieves its minimum link distance $LD(e', d)$ to d. Therefore, the minimum link distance to d starting vertically from v is either $LD(e', d) + 2$ or infinity if it is impossible to start vertically from v. It remains to observe that the minimum link distance to d starting horizontally from v is exactly $LD(e, d) + 1$. The above argumentation immediately yields the correctness of Algorithm 2 for the link distance between v and d.

1. Compute the diagonals e and e'.
2. Compute $LD(e, d)$ and $LD(e', d)$ by Algorithm 1.
3. $LD(v, d)$:= **if** one can start vertically from v towards d **then** $LD(e', d) + 2$ **else** ∞
4. $LD(v, d)$:= $\min(LD(e, d) + 1, LD(v, d))$

Algorithm 2: Algorithm for computing the link distance from a vertex v to a horizontal diagonal d

Now we analyze the complexity of the Algorithm 2. The maximal diagonal containing v can be found in constant time using the data structures built by Algorithm 1. Also the maximal diagonal e' can be found in constant time by using the above data structures, in particular Schieber and Vishkin's [26] parallel preprocessing for the lowest common ancestor queries. The minimum link distances in Step 2 are optimally computed by Algorithm 1. Hence, by Brent's principle, we obtain the following theorem.

Theorem 4.2 *Let d be a horizontal diagonal of a trapezoided rectilinear simple polygon P. Algorithm 2 computes the distance from d to all vertices of P in $O(\log n)$-time using the EREW PRAM with $O(n/\log n)$ processors.*

Next we discuss how to solve the problem of computing for a point in P the link distances to all vertices of P. For computing minimum link paths from a vertex in a simple polygon to all its

vertices a parallel algorithm has been presented in [12]; the algorithm runs in $O(\log^2 n \log\log n)$ time using $O(n)$ processors on the CREW PRAM. From a given point p in a rectilinear polygon P we here compute the horizontal and vertical chords by shooting in the four rectilinear directions towards the boundary of P. By Theorem 4.2 we can compute the link distances for each of the four chords to all vertices of P. If we wish to compute an approximation of the link distances from p only we are done. The correct distance from p to a vertex v can differ by at most one from the value obtained for v to one of the chords. The determination of the exact value is significantly harder (as is the case for many link distance problems see e.g. [29]). We need to compute the set of points, the interval, for v on the chord(s) having minimum link distance among all points on the chord(s) (recall the introduction). Given that information and the corresponding value for the link distance the problem is solved. In Lemma 6.3 we will show that for a given segment in P all vertex intervals and the corresponding link distances can be optimally computed in $O(\log n)$. It then follows,

Theorem 4.3 *Let p be a point in a trapezoided rectilinear simple polygon P. The distance from p to all vertices of P can be computed in $O(\log n)$-time using the EREW PRAM with $O(n/\log n)$ processors.*

Corollary 4.4 *Let s be a rectilinear segment in a trapezoided rectilinear simple polygon P. The distance from s to all vertices of P can be computed in $O(\log n)$-time using the EREW PRAM with $O(n/\log n)$ processors.*

5 Optimal parallel algorithm for computing window partition

The window partition of a simple polygon was introduced by Suri [29] as a technique for preprocessing a polygon that leads to efficient sequential algorithms for solving a number of link distance problems. Window partition has been effectively used for the sequential computation of vertex-vertex link distance, link center [10], link query problems [2], etc.

Its analog for rectilinear polygons is the histogram partition introduced by Levcopoulos [21]. In the histogram partitioning of a rectilinear polygon P, we partition P with respect to a diagonal d in P into regions over which the link distance from d is same. First compute the visibility polygon from d in P, which is a *histogram* with base d denoted as $H(d)$. The histogram $H(d)$ is the set of points which can be reached from d by a link. Remove the histogram $H(d)$ from P and this partitions P into several subpolygons. The link distance two is realized from those points of $P - H(d)$ which are visible from some boundary edge of $H(d)$. So for each boundary edge of $H(d)$, which is not an edge of $bd(P)$, called as *window*, compute the histogram in $P - H(d)$. This procedure of partitioning P into histograms is repeated till whole of P is covered. Finally, a partition of P into histograms is obtained. This partition of P is termed as the *histogram partition.*

In this section we present an optimal parallel algorithm for computing the histogram partition of P with respect to the diagonal d. As a consequence this fundamental tool is now available for solving in particular link distance problems in parallel. The first step of our algorithm is to compute the link distance from d to all vertices of P by Algorithms 1 and 2. Recall that, in the preprocessing step, the extension points of each edge to $bd(P)$ have been inserted as vertices on $bd(P)$. We show an important order property of the link distance of vertices of P and using this property we compute the histogram partition of P. The diagonal d partitions P into two

subpolygons P_1 and P_2. We restrict our attention to the subpolygon P_1. The algorithm and the arguments for P_2 are analogous. To keep the notation simpler, assume that P_1 is the subpolygon formed by $bd(p_1, p_n)$ and the diagonal p_1p_n, where p_1 and p_n are the endpoints of d. Algorithm 3 computes the histogram partition of P_1 with respect to the diagonal d. The correctness of Algorithm 3 follows from following lemmas.

1. Compute the link distance from d to all vertices of P_1 by Algorithms 1 and 2.
2. Construct an array, where the kth location in the array is the link distance of p_k from d.
3. Assign an open parenthesis to p_1 and a closing parenthesis to p_n.
4. Assign an open parenthesis to a vertex p_i if the link distance of p_{i+1} is more than p_i.
5. Assign a closing parenthesis to a vertex p_i if the link distance of p_{i-1} is greater than p_i.
6. Compute the matching parenthesis by the algorithm of Berkman *et. al* [4].
7. Construct a circular list of vertices belonging to each histogram in the histogram partition as follows.
   (a) Using the algorithm of Berkman *et al.* [4], compute for each vertex p_i of P_1 a pointer to the vertex p_j such that (i) $j > i$, (ii) link distance from d to p_i is greater than or equal to that of p_j and (iii) j is minimum.
   (b) If a vertex p_i points to a vertex p_j and the link distance from d to p_i and p_j is not same then remove the pointer from p_i to p_j.
   (c) Perform the list ranking on the above pointer configurations to obtain a circular list for each histogram in the histogram partition of P.

Algorithm 3: Algorithm for computing the histogram partition

Lemma 5.1 *Let the link distance of p_i and p_j from d be a and b, respectively, where p_i and p_j are two arbitrary vertices of P_1 and $i < j$. For each a' between a and b, there exist vertices on $bd(p_i, p_j)$ having link distance a' from d.*

Proof: trivial. □

Lemma 5.2 *The bracket sequence associated with vertices $p_1, ..., p_k$ is well-formed.*

Proof: The proof is by induction on the link distance L from diagonal d in P. Let P be a rectilinear polygon with link distance $L = 1$ from d. Then P is a histogram and thus no brackets are assigned to it. The bracket sequence is empty and the result follows in this case.

Now let P be a rectilinear polygon with link distance L from d and assume that for all polygons P' and diagonals d' in P' the result holds. Then compute the histogram from d. This induces a number of lids in the histogram together with their associated pockets. The link distance from a lid to its pocket is at most $L - 1$ and thus the bracket sequence assigned to it

inductively is well-formed. The link distance of any vertex properly contained in a pocket is one larger when computed from d than from the lid. Thus the brackets assigned in each pocket are also brackets for P computed from d. A lid is entered at a vertex p_i which is at link distance 1 from d and whose successor is at link distance 2; thus an opening parenthesis is assigned to p_i by step 4. One exiting the lid the reverse holds and, by step 5, a closing bracket is assigned to the vertex. (Note that we have included the Steiner points of the trapezoidation as vertices.) The enclosure of a well-formed bracket sequence by an opening and closing bracket is itself well-formed. All well-formed sequences of the pockets are encountered in order and thus appear in that order in the sequence of brackets associated with P. The concatenation of well-formed bracket sequences is itself well-formed which completes the proof.

□

Corollary 5.3 *Vertices p_ip_j forms the base of a histogram in the histogram partition of P if and only if they form a matching parenthesis pair.*

Now we analyze the complexity of the above algorithm. Link distance from d to all vertices of P can be computed in $O(\log n)$ time using $O(n/\log n)$ processors by the algorithm of Section 4. Opening and closing parenthesis to the appropriate vertices can be assigned in $O(\log n)$ time using $O(n)$ operations. Matching parenthesis and the computation of pointers from each vertex to its next can be done by the algorithm of Berkman *et al.* [4] in $O(\log n)$ time using $O(n)$ operations. Hence, the overall complexity of the above algorithm is $O(\log n)$ time using $O(n/\log n)$ processors. We summarize the results in the following theorem.

Theorem 5.4 *A histogram (or window) partition of an n-vertex trapezoided rectilinear polygon with respect to a diagonal can be computed in optimal $O(\log n)$ time using $O(n/\log n)$ processors on the EREW PRAM.*

6 Results on link queries

In this section we present an optimal parallel algorithm to preprocess a simple rectilinear polygon P such that the rectilinear link distance between two query points s and t in P can be computed efficiently. We show that the query data structure can be computed in optimal $O(\log n)$ time using $O(n/\log n)$ processors on the EREW PRAM. Given this data structure, a processor can answer link distance queries between two points in $O(\log n)$ time. Our results imply a linear time sequential algorithm for preprocessing rectilinear polygons for link distance queries. The previous best known algorithm for this problem is due to de Berg [3] and it requires $O(n \log n)$ time and $O(n \log n)$ space to build the query data structure. We achieve the improvement in running time and space for constructing the query data structure by partitioning the rectilinear polygon into disjoint histograms rather than recursively partitioning it.

Let e be a horizontal diagonal inside P that partitions P into two subpolygons P_1 and P_2. For all query point pairs where one of the query points is in P_1 and the other in P_2, any link path between the two query points intersects e. For these pairs we compute the link distance between the query points and e and then appropriately compose these two link distances to obtain the link distance between the query points. For all other pairs the problem reduces to that of finding the link distance in a subpolygon of P (this is the more challenging case).

In the following we first show how to compose the two link distances, if the query points are in the different subpolygons of e. For the diagonal e and a vertex v of P, let $e(v,l)$ be the part of e that can be reached from v with a path π of length l such that the last segment of π is perpendicular to e. Let the rectilinear link distance from v to e be defined as the distance from v to a closest point on e: $d(v,e) = min\{d(v,q)|q \in e\} = d_v$. The following lemma of deBerg [3] enables us to compose the two link distances.

Lemma 6.1 *(deBerg [3]) Let the diagonal e cut P into two subpolygons such that s and t lie in different subpolygons, and let $d(s,e) = d_s$ and $d(t,e) = d_t$. Then $d(s,t) = d_s + d_t + \Delta$, where*

$$\Delta = \begin{cases} -1 & if\ e(s,d_s) \cap e(t,d_t) \neq \emptyset \\ 0 & if\ e(s,d_s) \cap e(t,d_t) = \emptyset \wedge \\ & (e(s,d_s+1) \cap e(t,d_t) \neq \emptyset \vee e(s,d_s) \cap e(t,d_t+1) \neq \emptyset) \\ +1 & otherwise \end{cases}$$

The above lemma suggests that to compose the two link distances it is sufficient to compute the *fast interval* $e(v,d_v)$ and the *slow interval* $e(v,d_v+1)$ for each vertex v in the subpolygon, where e is the diagonal and $d_v = d(v,e)$. Following lemma of deBerg [3] shows that for any vertex v at distance $d_v > 2$ from e, there exists a vertex v_{next} such that any point on $e(v,d_v)$ can be optimally reached via v_{next}. Similarly, a vertex v_{next2} exists such that any point on $e(v,d_v+1)$ can be reached via v_{next2}.

Lemma 6.2 *(deBerg [3])*

1. *Let v be a vertex of P with $d(v,e) = d_v > 2$. Then a vertex v_{next} of P exists such that $d_{v_{next}} = d_v - 1$ or $d_{v_{next}} = d_v - 2$ and $e(v_{next}, d_{v_{next}}) = e(v,d_v)$. Moreover, for every point $x \in e(v,d_v)$ there exists a shortest path $\pi = l_1 l_2 ... l_{d_v}$ from v to x with $v_{next} \in l_2$.*

2. *Let v be a vertex of P with $d(v,e) = d_v > 1$. Then a vertex v_{next2} of P exists such that $d_{v_{next2}} = d_v$ and $e(v_{next}, d_{v_{next}}) = e(v,d_v+1)$. Moreover, for every point $x \in e(v,d_v+1)$ there exists a shortest path $\pi = l_1 l_2 ... l_{d_v}$ from v to x with $v_{next2} \in l_2$.*

Using the above lemmas, Algorithm 4 computes the intervals $e(v,d_v)$ and $e(v,d_v+1)$ and the vertices v_{next} and v_{next2} for each vertex v of P in optimal $O(\log n)$ time using $O(n)$ operations.

Lemma 6.3 *Algorithm 4 computes $e(v,d_v)$, $e(v,d_v+1)$, v_{next} and v_{next2} for each vertex v of P in optimal $O(\log n)$ time using $O(n/\log n)$ processors on the EREW PRAM.*

Proof: The correctness of the algorithm follows from Lemma 6.2. Now we analyze the parallel complexity of the algorithm. The histogram partition of P can be computed in $O(\log n)$ time using $O(n)$ operations by Algorithm 3. Various visibility information within a trapezoided histogram can be computed in optimal $O(n)$ operations. The pointer jumping technique requires $O(\log n)$ time using $O(n)$ operations [16]. Hence, the overall complexity of the algorithm follows. □

Lemma 6.4 *If the query pair (s,t) is located on different sides of the diagonal e of P, then the link distance between s and t can be computed in optimal $O(\log n)$ time.*

1. Compute the histogram partition of P with respect to e by Algorithm 3.
2. Project all vertices of each histogram onto its base.
3. For each histogram compute the vertex-edge visible pairs.
4. For a vertex v with $d_v > 2$, there are exactly two possibilities for v_{next}. Let $v \in H_{d_v}$, i.e., the histogram at a distance d_v in the histogram partitioning of P. Either turn immediately while entering into the histogram H_{d_v-1} from H_{d_v} and v_{next} is a vertex of the base of H_{d_v}, or turn as late as possible and v_{next} is a vertex of the edge of H_{d_v-1} that is visible from v.
5. For all vertices v with $d_v \leq 2$, compute $e(v, d_v)$ and For all vertices v with $d_v > 2$ assign a pointer to its next vertex v_{next}.
6. Using the pointer jumping technique [16], for each vertex w assign a pointer to a vertex v for which $e(v, d_v)$ is known and $d_v \leq 2$. Assign $e(w, d_w) = e(v, d_v)$.
7. Perform analogous steps for computing $e(v, d_v + 1)$.

Algorithm 4: Algorithm for computing intervals

Proof: Suppose that we can locate v_{next} and v_{next2} vertices of s and t, then we can compute their intervals on e by Algorithm 4 and compose them by Lemma 6.1 to obtain the link distance between s and t. Now we describe the data structures to compute v_{next} and v_{next2} vertices for a query point v. One data structure is required to compute the edge of H_{d_v-1} that is hit by an axis-parallel query ray entering H_{d_v-1} through the base of H_{d_v}, another data structure is used to compute the edge of H_{d_v} that is hit by a ray parallel to the base of H_{d_v}. Note that H_{d_v} is the histogram in the histogram partition of P, containing v, at link distance d_v from diagonal e. So we need to preprocess each histogram for ray shooting queries with rays that are parallel to the base of the histogram. This can be achieved by adding segments that are parallel to the base from every reflex vertex of each histogram to the opposite side. These segments can be added in optimal parallel work. Note that the total number of segments introduced is linear. We can locate the histograms containing the query points in $O(\log n)$ time by the algorithm of Tammasia and Vitter [31]. Hence, the lemma follows. □

Next we state the procedure for computing the link distance between the query points when they are located on the same side of the diagonal e. Algorithm 5 describes the procedure for computing a data structure for answering such link distance queries. Without loss of generality assume that the query pair is located in the subpolygon P_1. Further, to simplify the notation we denote P_1 by P.

Lemma 6.5 *Algorithm 5 runs in $O(\log n)$ time using $O(n/\log n)$ processors on the EREW PRAM and the size of the data structure computed by it is linear.*

Proof: There are at most a linear number of histograms in the histogram partition of P. Therefore, the number of nodes in T_H is at most $O(n)$. The lowest common ancestor data structure of Schieber and Vishkin [26] and the planar point location data structure of Tamassia

1. Compute the histogram partition of P with respect to e by Algorithm 3.
2. Construct the dual tree T_H of the histogram partition of P. The nodes in T_H are histograms and there is an edge between two histograms, if the corresponding histograms are incident to a common lid.
3. Construct a data structure to answer lowest common ancestors queries in T_H by the algorithm of Schieber and Vishkin [26].
4. Construct a planar point location data structure over the histograms in the histogram partition of P by the algorithm of Tamassia and Vitter [31].
5. Compute the data structure to locate v_{next} and v_{next2} for query points in the relevant histograms as follows.
   (a) Compute the v_{next} vertex for each vertex v of P by Algorithm 4.
   (b) For each vertex v of P, assign a pointer to its v_{next} vertex; this defines a tree referred to by T_{next}. The parent of v in T_{next} is v_{next}.
   (c) Create a dummy node to root the forest T_{next}.
   (d) For each vertex v in the tree T_{next} compute the corresponding histogram on whose base v forms a fast interval.
   (e) Assign a label to each vertex v in T_{next} as follows. Let H be the histogram corresponding to v. The vertex v is assigned a label i if the depth of the node corresponding to H in T_H is i.
   (f) Compute the preorder numbering of the nodes in T_{next} by the algorithm of Tarjan and Vishkin [32] and store the nodes (pointers to them) in the consecutive location in a linear array A.
   (g) Construct the data structure to answer lowest common ancestor queries in T_{next} by the algorithm of Schieber and Vishkin [26].
   (h) Perform the analogous steps for v_{next2} replacing fast intervals by slow intervals and T_{next} by T_{next2}.

Algorithm 5: Algorithm for computing the link query data structure.

1. Locate the histograms containing s and t by the algorithm of Tamassia and Vitter [31].

2. Compute the node corresponding to the lowest common ancestor histogram of the histograms containing s and t in T_H by the algorithm of Schieber and Vishkin [26].

3. Let H be the histogram corresponding to the lowest common ancestor node in T_H. If s and t are located in H then compute the link distance between them as follows :
   Without loss of generality assume that the base of H is horizontal and the interior of H is above the base.
   If $s = t$ then $LD(s,t) = 0$ else if s and t are visible then $LD(s,t) = 1$.
   Otherwise, let t' be the maximal vertical segment passing through t in H and let s' be the maximal horizontal segment passing through s in H.
   If s' and t' intersects then $LD(s,t) = 2$ else it is 3.
   The segments s' and t' are computed using the horizontal and vertical trapezoidation of H.

4. Let w_s (or w_t) be the lids (or windows) in H of the pocket containig s (respectively, t) and let H_s (respectively, H_t) be the histogram with base w_s (respectively, w_t). Compute the slow and fast intervals from s (resp., t) on w_s (resp., w_t) by Algorithm 7.

5. Let link distance from s to w_s be d_s and the link distance from t to w_t be d_t. Let d_s and d_t denote the link distance from s to w_s and t to w_t, respectively. The link distance between s and t, $LD(s,t) = d_s + d_t + \Delta$, where Δ is as given by Algorithm 8.

Algorithm 6: Algorithm for answering link queries.

1. Compute the next vertex of the query point s by the procedure discussed in Lemma 6.3 and call it v.

2. **Procedure for locating the vertex u (corresponding to the histogram H_s) on the path from v to the root of T_{next} for which (1) $LD(u, w_s) \leq 2$ and (2) the interval formed by u and v on w_s is same.**
   Perform a binary search in the array A computed in Algorithm 5 in the following manner.
   (a) Compute the middle element, say x, of A.
   (b) Compute the lowest common ancestor, x' of x and v by the algorithm of [31].
   (c) If the label of x' is same as that of u, then $u = x'$ and the binary search ends.
   (d) Otherwise ** Label of $u \neq$ lable of x'.**
   If the preorder number of $x \geq v$ or the level of x' is greater than u, then perform a binary search in the left half of A else in the right half of A.

3. Assign to s the interval formed by u on H_s.

4. Repeat this procedure to compute intervals of t on w_t.

Algorithm 7: Algorithm for computing intervals.

1. Let the base of histogram H be horizontal and assume that the interior of H is above its base. Compute whether the pockets w_s and w_t are visible (w_s and w_t are said to be visible if there exist a segment dd' which lies completely inside H and $d \in w_s$ and $d' \in w_t$). This is done as follows: Let a and b be the lower endpoints of the pockets w_s and w_t, respectively. Let a' and b' be the maximal horizontal segment from a and b respectively in H. Compute whether a' intersects w_t or b' intersects w_s.

2. (*** Pockets w_s and w_t are not visible ***)
Neither a' intersects w_t nor b' intersects w_s. Let endpoints of the fast and slow interval from s on w_s be (s_1, s_2) and (s_1, s_3), respectively. Let s_2s_2' and s_3s_3' be the maximal horizontal segment through s_2 and s_3 in H. Let w_t' be the maximal vertical segment containing w_t in H. If s_2s_2' intersects w_t' then $\Delta = 1$
else if s_3s_3' intersects w_t' then $\Delta = 2$
else $\Delta = 3$.

3. (*** Pockets w_s and w_t are visible ***)
Either a' intersects w_t or b' intersects w_s. Project the interval endpoints s_1, s_2, s_3 on w_t and analogously project t_1, t_2, t_3 on w_s.
If s_1s_2 and t_1t_2 are visible then $\Delta = -1$. If s_2s_3 and t_1t_2 are visible then $\Delta = 0$. If s_2s_3 and t_2t_3 are visible then $\Delta = 1$. If s_1s_3 and t_1t_3 are not visible then $\Delta = 2$.

Algorithm 8: Algorithm for computing Δ.

and Vitter [31] require $O(\log n)$ construction time using a linear number of operations and space. Since the vertices v_{next} and v_{next2} for each vertex v are unique, the number of nodes in trees T_{next} and T_{next2} is linear. The preorder numbering of vertices in the tree can be computed by using Euler tour technique of Tarjan and Vishkin [32] in $O(\log n)$ time using linear number of operations and storage. □

Lemma 6.6 *A single processor can answer link distance queries in $O(\log n)$ time by executing Algorithm 6.*

Proof: The histograms containing the query points s and t can be located in $O(\log n)$ time by the algorithm of [31]. Lowest common ancestor of two nodes in a tree can be computed in constant time [26]. The required visibility informations can be computed in at most $O(\log n)$ time in a trapezoided histogram. The binary search in the array A to locate the vertex u takes $O(\log n)$ time. The algorithm for computing Δ requires at most $O(\log n)$ time since the projection of interval points to the lids of the pockets can be achieved with in the claimed complexity bound. □

Theorem 6.7 *A data structure which supports rectilinear link distance queries in any n-vertex trapezoided rectilinear polygon can be constructed in $O(\log n)$ time using $O(n/\log n)$ processors on the EREW PRAM. Using this data structure a single processor can answer link distance queries between two points in $O(\log n)$ time.*

Proof: The complexity bounds follow from Lemmas 6.4, 6.5 and 6.6. Now we show the correctness. If the query point are located in different subpolygons of e then the correctness follows from Lemma 6.4. If the query points are located within a histogram of the histogram partition of P, the correctness is obvious. Assume that the query points s and t lie in different histograms and let H be the lowest common ancestor histogram of them in the dual tree T_H (Step 2, Algorithm 6). Observe that the link path from s to t passes through the histogram H. Now we compute the intervals on the lids of the pockets of H containing s and t and appropriately compose the two link distances. The binary search (Step 2, Algorithm 7) computes the correct node u since the labels of vertices in any path from a node to the root are monotonically decreasing. Hence, the correctness follows. □

Since our parallel algorithm performs a total work of $O(n)$. As a consequence we obtain a linear time sequential algorithm for computing the link query data structure. This improves on the $O(n \log n)$ bound established for this problem by de Berg [3]. It is also possible to design a linear time sequential algorithm which is simpler than one obtained by straight-forward adaptation of our parallel algorithm. The procedure for computing intervals on the lids of the pockets from s and t to H_s and H_t can be modified as follows.

Label the vertices of the polygon according to its postorder number in the tree T_{next}. With each node in T_{next} store a range containing the postorder label of vertices in its subtree. Note that as a consequence of the postorder labelling of vertices the range associated with each node is an interval. For each node v the histogram is known on whose base the corresponding vertex of v forms an interval. We form a linked list for each histogram containing the labels of the vertices which forms intervals at its base. This can be done by a postorder traversal of T_{next} during which we append the interval associated with each node encountered to the linked list of its associated histogram. Observe that the intervals appearing in the linked lists are in sorted order. Now to locate the interval corresponding to a query point q on the base of a particular histogram H, we perform a binary search in the linked list (array) associated with H and locate the vertex containing the label for s. Once we know the vertex then the computation of intervals on the base of histogram is straightforward. This procedure runs in linear time and it produces the sorted lists for each histogram. This procedure doesn't work optimally in parallel because to produce the sorted lists for each histogram, we require an optimal procedure to do stable integer sort where the integers are in the bounded range. Unfortunately, there does not exist an optimal parallel algorithm to do stable integer sort.

Theorem 6.8 *A data structure exists which allows rectilinear link distance queries between two query points in an n-vertex rectilinear polygon to be answered in $O(\log n)$ time. The data structure can be computed in optimal $O(n)$ time using $O(n)$ space.*

7 Link diameter

In this section we present a parallel algorithm for computing the link diameter of an n-vertex simple rectilinear polygon P. The link diameter of P is defined as $Diam(P) = max\{LD(s,t) | s,t \in P\}$, where $LD(s,t)$ denotes the link distance between two points s and t in P. The diameter is realized between a pair of vertices of P. In addition to computing the value $Diam(P)$ we are interested in producing a pair of vertices and a minimum link path of length $Diam(P)$ between these.

The sequential method developed by de Berg [3] (see Algorithm 9) for computing $Diam(P)$ runs in $O(n \log n)$ time and is easily parallelizable.

1. If P is a rectangle then $Diam(P) = 2$, otherwise go to Step 2.
2. Compute a cut segment e of P that cuts P into two subpolygons P_1 and P_2, such that $|P_1|, |P_2| \leq \frac{3}{4}n + 2$.
3. Compute $Diam(P_1)$ and $Diam(P_2)$, recursively.
4. Compute $M = max\{LD(v, w) | v \in P_1, w \in P_2\}$.
5. Let $Diam(P) := max(Diam(P_1), Diam(P_2), M)$.

Algorithm 9: de Berg's algorithm for computing the link diameter

Nilsson's and Schuierer's [22] algorithm differs from that of de Berg in Step 3 and is based on the following observations. The value of M is at least $d_1 + d_2 - 1$. W.l.o.g. assume that $d_1 \geq d_2$ then $Diam(P_2) \leq 2d_2 + 1$. If $Diam(P_2) \leq d_1 + d_2 - 1$ then there is no need to recur on P_2. This implies that the values of $Diam(P_2)$ which are of interest to us lie in the range from $d_1 + d_2 - 1$ to $2d_2 + 1$. In [22] a linear time (in the size of P_2) algorithm is presented for computing whether $Diam(P_2) \leq 2d_2 - 1 \leq M$ and if $D(P_2) > 2d_2 - 1$ then the exact value of $Diam(P_2)$. The recurrence relation for the time complexity $T(n)$ of the algorithm of [22] is $T(n) = T(\frac{3}{4}n) + O(n)$ and which is $O(n)$. Hence the rectilinear link diameter can be computed in optimal linear time in sequential.

We analyze the parallel complexity of the above algorithm. Assume that we can determine in optimal parallel work the exact value of $Diam(P_2)$ or whether $Diam(P_2) \leq M$. The straightforward parallel implementation of the above sequential algorithm will give the following recurrence for the time complexity $T(n) = T(\frac{3}{4}n) + O(\log n)$ of the parallel algorithm and which is $O(\log^2 n)$. It can be seen that the processor complexity of the algorithm is $O(n/\log n)$. It would appear that a straightforward parallel implementation of the sequential algorithm does not lead to any improvement in the complexity of the parallel algorithm.

We present a general technique to obtain an $O(\log n \log\log n)$ time parallel algorithm for any sequential algorithm whose time complexity is described by a recurrence relation $T(n)$ of the form $T(n) = T(cn) + O(n)$, where the constant $c < 1$ and for which the $O(\log n)$ recursion steps each can be implemented optimally in logarithmic (in the size of the subproblem) parallel time.

We analyze the computation in the recursion tree. The recursion tree is a binary tree with $O(\log n)$ levels. Once the computation at the root has been completed the algorithm needs to recur on either the left or the right subtree. During the computation at the second level, which is of size $O(cn)$, a fraction of the $O(n)$ processors are idle and this holds analogously for any subsequent level of the computation. So perform the computation simultaneously in several layers instead of performing it level by level (in a sort of speculative way). Notice that this is not a straight-forward application of Brent's principle. While the total work performed by the sequential algorithm is only linear a parallel algorithm does not know in advance which path of the recursion tree will be taken.

Assume now that the computation at each level takes $O(\log s)$ time using $O(s/\log s)$ processors, where s is the size of the subproblem to be solved at that level. Then the parallel algorithm we construct has $O(\log\log n)$ stages and at each stage it performs a total computation of $O(\log n)$ time using $O(n/\log n)$ processors. For the ith stage we are interested to solve the subproblems associated with a subtree of the recursion tree. The subtree of the ith stage is rooted at a vertex of level 2^{i-1} and it consists only of nodes on levels 2^{i-1} to $2^i - 1$. The total size of the subproblems associated with this subtree is easily seen to be $O(n)$.

For the ith stage let $T_v(i)$ be the recursion subtree rooted at v, where v is a vertex at level 2^{i-1}. Since the total size of the subproblems associated with $T_v(i)$ is $O(n)$ we can assign a linear number (in the size of the subproblem associated with a node) of processors to the subproblem associated with each node. Each node of $T_v(i)$ can solve its associated subproblem and determine which of its children will be active for the next step of the recursion in optimal parallel work. Now we know for each node in $T_v(i)$, which child will be active during the next recursion step. Using the standard method of doubling we can find the path from v to a leaf of $T_v(i)$ of the active nodes in the recursion. Let this leaf node be v'. In the $(i+1)^{st}$ stage, the recursion subtree will be rooted at v' and will have nodes from levels 2^i to $2^{i+1} - 1$. It is easy to see that each stage of the recursion can be implemented in $O(\log n)$ time using $O(n/\log n)$ processors. We summarize the above observations in the following theorem.

Theorem 7.1 *Let $T(n) = T(cn)+O(n)$, $c < 1$ be the recurrence relation for the time complexity of a sequential algorithm and assume that each step in the recursion can be implemented in $O(\log s)$ time using $O(s/\log s)$ processors where s is the size of the subproblem solved in that step. The above sequential algorithm can be transformed into an equivalent parallel algorithm which runs in $O(\log n \log\log n)$ time using $O(n/\log n)$ processors.*

By using the algorithm in Section 6 we compute the intervals $I_1 = \{e(v, d_2)|v \in P_2\}$ and $I_2 = \{e(v, d_2 + 1)|v \in P_2\}$. Then we test whether the inequality for $Diam(P_2)$ is met; if this is not so, we compute its exact value. The overall parallel complexity of this procedure is $O(\log n)$ time using $O(n/\log n)$ processors. We summarize our results in the following theorem.

Theorem 7.2 *The rectilinear link diameter of an n-vertex simple rectilinear polygon can be computed in $O(\log n \log\log n)$ time using $O(n/\log n)$ processors on the EREW PRAM. A link path connecting two vertices of P realizing the diameter can be found in $O(\log n)$ time using $O(n/\log n)$ processors on the EREW PRAM.*

8 Central diagonal and link center

In this section we present parallel algorithms for computing the link radius, a central diagonal and the link center of an n-vertex simple rectilinear polygon P. Djidjev et al. [10] introduced the concept of a central diagonal of a simple polygon P. A *central diagonal* is a diagonal say d for which the difference between the covering radii to the two subpolygons induced by d is minimized. They showed that a central diagonal exists for which the covering radius difference is at most one (in absolute value); it can be found in $O(n \log n)$ time. Furthermore they showed that the covering radii of a central diagonal differs from the link radius R of P by at most one. A central diagonal is near the link center or more precisely the link center of a simple polygon is in the 2-visibility region of the central diagonal. This was used to compute, in $O(n \log n)$ time,

the link center of a simple polygon [10] as well as to find a shortest central segment inside P [1] (see also [17]).

A *central segment* is a segment in P which minimizes the covering radius to both sides. A robot having telescoping links and moving along a track can reach every point of the simple polygon P with the minimum number of links if the track is placed at the central segment. A shortest central segment minimizes the track length. For simple polygons a shortest central segment can be found in $O(n \log n)$ time.

For rectilinear polygons Nilsson et al. [22] made analogous observations regarding the central diagonal; they called such a 'diagonal' a splitting chord. We prefer to keep the notation central diagonal. A central diagonal exists whose covering radius difference is at most one and whose covering radii on either side are R-1 or R, where R is the link radius of P.(The covering radius of a diagonal within a subpolygon is the maximum link distance of any vertex in the subpolygon to the diagonal.) A central diagonal can be found in linear time. Using the techniques developed in this paper the approach taken by Nilsson et al. is easily parallelizable; it is thus only sketched here.

To find a central diagonal first a diameter realizing path is constructed which can be done using Section 7. Let v_1, v_2 be determined as a vertex pair realizing the diameter as its link distance; again this can be done by using the results developed in Section 7. The segment at the lower median index position on the path from v_1 to v_2 is shown to be in the vicinity of a central chord [22]. Finding a central chord then reduces to computing covering radii to both sides from at most four chords. This can be done using Section 6 where the optimal parallel interval computation is given. To find these segments from the segment at the lower median index position the computations performed are easily parallelized. Except for the computation of the diameter all steps can be performed in optimal parallel time. Since the diameter computation takes $O(\log n \log\log n)$ time we get,

Theorem 8.1 *A central diagonal in an n-vertex simple rectilinear polygon can be computed in $O(\log n \log\log n)$ time using $O(n/\log n)$ processors on the EREW PRAM.*

The *link center* of a simple polygon P is the set of points x in P at which the maximal link distance from x to any other point in P is minimized. The link center-problem has several potential applications. It could be applied to locating a transmitter so that the maximum number of retransmission needed to reach any point in a polygonal region is minimized, or to chosing the best location for a mobile unit minimizing the number of turns needed to reach any point in a polygonal regions. In [10] an $O(n \log n)$ algorithm is given to determined the link center of a simple n-vertex polygon. In [22] a linear time sequential algorithm for computing the rectilinear link center of P is presented. The *rectilinear* link center of a rectilinear polygon is obtained by using the rectilinear link distance. We show that their algorithm can be parallelized by using the techniques developed in this paper. Now we sketch their algorithm and show how it can be parallelized.

First compute the link radius R of P. The *link radius* of P is the maximum link distance from a point in the link center to any vertex of P. It has been shown in [20] that $\lceil Diam(P)/2 \rceil \leq R \leq \lceil Diam(P)/2 \rceil + 1$. Using the above inequality, we know that R can have two possible values. Compute the link center of P by assuming first the lower value of R. Either this is the correct value of R or the algorithm will report that the link center is empty. In either case the exact value of R is known.

Compute the central diagonal d of P. Let d splits P into two subpolygons, P_1 and P_2. Let d_1 (or d_2) denote the maximum link distance from d to vertices in P_1 (respectively, P_2). Without loss of generality assume that d is vertical and P_1 is to the left of d and P_2 is to the right of d. Since d is the central diagonal, $d_1, d_2 \geq R - 1$. Compute the part of the link center in P_1 and P_2. Since the computations are analogous, we discuss the computation of the link center in P_2. If $R \leq d_1$, then the link center in P_2 is contained inside the histogram of d in P_2.

Computation of the link center in a histogram H is based on the following observations. For each window w of H, compute the region of H that can be reached from the vertices in the pocket of w by using at most R links. Once this region for each pocket is determined, the remaining task is to intersect these regions for all pockets. The region in H due to each pocket can be determined by knowing the slow and fast intervals from the vertices inside the pocket to the window w of the pocket. The region due to each pocket in H is shown to be monotone and thus can be intersected efficiently to obtain the link center in H.

If $d_1 = R - 1$ then the link center is in the 2-visibility region of d in P_2. There are two main cases depending on whether the fast intervals for vertices in P_1 have a non-empty intersection on d or not. In each case there are various subcases and finally the computation is reduced to that of computing the link center in a histogram. In the following we state the various procedures which are required by the algorithm in [22] to compute the link center.

1. Central diagonal d.
2. 1-visibility (or histogram) and 2-visibility polygons from d.
3. Slow and fast intervals for the vertices inside a pocket to its window w or to the diagonal d.
4. The point where a segment inside P hits the boundary of P when moved horizontally and vertically.
5. Intersection of horizontally (or vertically) monotone histograms.
6. Intersection of the fast intervals from vertices in P_1 (or P_2) on d. If the intersection is empty then compute two vertices such that their interval on d do not intersect.

Observe that each of the above steps have a parallel implementation. Steps 2 to 6 can be implemented in optimal $O(\log n)$ time using $O(n)$ operations. The computation of central diagonal requires $O(\log n \log\log n)$ time using $O(n/\log n)$ processors by Theorem 8.1. We summarize the result in the following theorem.

Theorem 8.2 *The rectilinear link center of an $n-$vertex rectilinear simple polygon can be computed in $O(\log n \log\log n)$ time using $O(n/\log n)$ processors on the EREW PRAM.*

It has been observed [20, 10] that any algorithm for constructing the link center which is based on knowledge of the *exact* value R of the link radius, i.e. the algorithm returns the empty set if R is estimated to be too small, can be used to find the link radius. The above algorithm is of that kind and thus we get:

Corollary 8.3 *The rectilinear link radius of an $n-$vertex rectilinear simple polygon can be computed in $O(\log n \log\log n)$ time using $O(n/\log n)$ processors on the EREW PRAM.*

9 Conclusions and open Problems

We have given optimal algorithms for a variety of fundamental problems involving the link distance in trapezoided rectilinear polygons. As yet no optimal EREW algorithm is know for trapezoiding rectilinear polygons and thus an obvious open problem is to find such an algorithm. A solution to this problem implies that our algorithms are optimal even for (untrapezoided) rectilinear polygons. We have also given applications of our algorithms to a number of other link distance problems yielding to almost optimal solutions for the link diameter, link center, link radius, and central diagonal problems. The total work performed by each of these algorithms is $O(n \log\log n)$ which is close to the optimal linear-time sequential bound. It remains open if the sequential bounds can be matched.

References

[1] L.G. Alexandrov, H.N. Djidjev and J.-R. Sack, *Finding a central link segment of a simple polygon in $O(n \log n)$ time*, Technical Report No. SCS-TR-163, Carleton University, 1989.

[2] E.M. Arkin, J.S.B. Mitchell and S. Suri, *Link queries and polygon approximation*, Proc. of the 3rd ACM-SIAM Symp. on Discrete Algorithms, 1991, pp. 269-279.

[3] M. de Berg, *On rectilinear link distance*, Computation Geometry: Theory and Applications, 1 (1991), pp. 13-34.

[4] O. Berkman, B. Schieber and U. Vishkin, *Some doubly logarithmic optimal parallel algorithms based on finding all nearest smaller values*, Technical Report UMIACS-TR-88-79, University of Maryland, 1988.

[5] B. Chazelle, *Triangulating a simple polygon in linear time*, Discrete and Computational Geometry, 4 (1991), pp. 485-524.

[6] V. Chandru, S.K. Ghosh, A. Maheshwari, V T Rajan and S. Saluja, *$\mathcal{NC}$-Algorithms for minimum link path and related problems*, Technical Report, Tata Institute of Fundamental Research, Bombay, 1992.

[7] K.L. Clarkson, R. Cole and R.E. Tarjan, *Randomized parallel algorithms for trapezoidal decomposition*, Proc. 7th ACM Symp. on Computational Geometry, 1991, pp. 152-161.

[8] K.L. Clarkson, S. Kapoor and P.M. Vaidya, *Rectilinear shortest paths through polygonal obstacles in $O(n(\log n)^2)$ time*, Proc. 3rd Annual ACM Symp. on Computation Geometry, 1987, pp. 251-257.

[9] P.J. de Rezende, D.T. Lee and Y.F. Wu, *Rectilinear shortest paths with rectangular barriers*, Discrete and Computation Geometry, 4 (1989), pp. 41-53.

[10] H.N. Djidjev, A. Lingas and J.-R Sack, *An $O(n \log n)$ algorithm for computing the link center of a simple polygon*, Discrete and Computational Geometry, 8 (1992), pp. 131-152.

[11] S.K. Ghosh, *Computing the visibility polygon from a convex set and related problems*, Journal of Algorithms, 12(1991), pp. 75-95.

[12] S.K. Ghosh and A. Maheshwari, *Parallel algorithms for all minimum link paths and link center problems*, SWAT 92, Lecture Notes in Computer Science, vol. 621, 1992.

[13] M.T. Goodrich, *Planar separators and parallel polygon triangulation*, Proc. ACM STOC, pp. 507-515, 1992.

[14] M.T. Goodrich, S. Shauck and S. Guha, *Parallel methods for visibility and shortest path problems in simple polygons*, Proc. 6th ACM Symposium on Computational Geometry, 1990, pp. 73-82.

[15] J. Hershberger, *Optimal parallel algorithms for triangulated simple polygons*, Proc. 8th ACM Symposium on Computational Geometry, 1992, pp. 33-42.

[16] J. JáJá, *An introduction to parallel algorithms*, Addison-Weseley Publishing Company, 1992.

[17] Y. Ke, *An efficient algorithm for link distance-problems*, Proc. 5th ACM Symposium on Computational Geometry, 1989, pp. 69-78.

[18] R. M. Karp and R. Vijaya Ramachandran, *Parallel Algorithms for Shared-Memory Machines*, Handbook of Theoretical Computer Science, Edited by J. van Leeuwen, Volume 1, Elsevier Science Publishers B.V., 1990.

[19] R. Klein and A. Lingas, *Manhattonian Proximity in a Simple Polygon*, Proc. 8th ACM Symp. on Computational Geometry, 1992, pp. 312-319.

[20] W. Lenhart, R. Pollack, J. Sack, R. Seidel, M. Sharir, S. Suri, G. Toussaint, S. Whitesides and C. Yap, *Computing the link center of a simple polygon*, Discrete and Computational Geometry, 3 (1988), pp. 281-293.

[21] C. Levcopoulos, *On approximation behaviour of the greedy triangulation*, Linköping Studies in Science and Technology, Ph.D. Thesis, No. 74, Linköping University, Sweden, 1986.

[22] B.J. Nilsson and S. Schuierer, *An optimal algorithm for the rectilinear link center of a rectilinear polygon*, Proc. 2nd Workshop on Algorithms and Data Structures, 1991, pp. 249-260.

[23] J. O'Rourke, *Art gallery theorems and algorithms*, Oxford University Press, 1987.

[24] J.H. Reif and J.A. Storer, *Minimizing turns for discrete movement in the interior of a polygon*, IEEE Journal of Robotics and Automation, RA-3 (1987), pp. 182-193.

[25] J-R. Sack, *Rectilinear Computational Geometry*, Ph.D. Thesis, McGill University, 1984.

[26] B. Schieber and U. Vishkin, *On finding lowest common ancestors: Simplification and Parallelization*, SIAM J. on Computing, 17(1988), pp. 1253-1262.

[27] S. Schuierer, Technical report, University of Freiburg, recent personal communication to the authors of this paper by B. Nilsson.

[28] S. Suri, *A linear time algorithm for minimum link path inside a simple polygon*, Computer Vision, Graphics and Image Processing, 35(1986), pp. 99-110.

[29] S. Suri, *Minimum link paths in polygons and related problems*, Ph.D. Thesis, Johns Hopkins University, 1987.

[30] S. Suri, *Computing furthest neighbors in simple polygons* , J. Comput. Sci., 39(1989), pp. 220-235.

[31] R. Tamassia and J.S. Vitter, *Optimal parallel algorithms for transitive closure and point location in planar structures*, Proc. 1st ACM Symposium on Parallel Algorithms and Architectures, pp. 339-408, 1989.

[32] R.E. Tarjan and U. Vishkin, *An efficient parallel biconnectivity algorithm*, SIAM Journal on Computing, **14**, pp. 862-874, 1982.

School of Computer Science, Carleton University
Recent Technical Reports

TR-194 **String Editing with Substitution, Insertion, Deletion, Squashing and Expansion Operations**
B John Oommen, September 1991

TR-195 **The Expressiveness of Silence: Optimal Algorithms for Synchronous Communication of Information**
Una-May O'Reilly and Nicola Santoro, October 1991

TR-196 **Lights, Walls and Bricks**
J. Czyzowicz, E. Rivera-Campo, N. Santoro, J. Urrutia and J. Zaks, October 1991

TR-197 **A Brief Survey of Art Gallery Problems in Integer Lattice Systems**
Evangelos Kranakis and Michel Pocchiola, November 1991

TR-198 **On Reconfigurability of Systolic Arrays**
Amiya Nayak, Nicola Santoro, and Richard Tan, November 1991

TR-199 **Constrained Tree Editing**
B. John Oommen and William Lee, December 1991

TR-200 **Industry and Academic Links in Local Economic Development: A Tale of Two Cities**
Helen Lawton Smith and Michael Atkinson, January 1992

TR-201 **Computational Geometry on Analog Neural Circuits**
Frank Dehne, Boris Flach, Jörg-Rüdiger Sack, Natana Valiveti, January 1992

TR-202 **Efficient Construction of Catastrophic Patterns for VLSI Reconfigurable Arrays**
Amiya Nayak, Linda Pagli, Nicola Santoro, February 1992

TR-203 **Numeric Similarity and Dissimilarity Measures Between Two Trees**
B. John Oommen and William Lee, February 1992

TR-204 **Recognition of Catastrophic Faults in Reconfigurable Arrays with Arbitrary Link Redundancy**
Amiya Nayak, Linda Pagli, Nicola Santoro, March 1992

TR-205 **The Permutational Power of a Priority Queue**
M.D. Atkinson and Murali Thiyagarajah, April 1992

TR-206 **Enumeration Problems Relating to Dirichlet's Theorem**
Evangelos Kranakis and Michel Pocchiola, April 1992

TR-207 **Distributed Computing on Anonymous Hypercubes with Faulty Components**
Evangelos Kranakis and Nicola Santoro, April 1992

TR-208 **Fast Learning Automaton-Based Image Examination and Retrieval**
B. John Oommen and Chris Fothergill, June 1992

TR-209 **On Generating Random Intervals and Hyperrectangles**
Luc Devroye, Peter Epstein and Jörg-Rüdiger Sack, July 1992

TR-210 **Sorting Permutations with Networks of Stacks**
M.D. Atkinson, August 1992

TR-211 **Generating Triangulations at Random**
Peter Epstein and Jörg-Rüdiger Sack, August 1992

TR-212 **Algorithms for Asymptotically Optimal Contained Rectangles and Triangles**
Evangelos Kranakis and Emran Rafique, September 1992

TR-213 **Parallel Algorithms for Rectilinear Link Distance Problems**
Andrzej Lingas, Anil Maheshwari and Jörg-Rüdiger Sack, September 1992