Optimal 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.tifr.res.in

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

Keywords: Computational geometry, Algorithms and data structures, Parallel computation, Link distance, Rectilinear polygons.

Abstract

We provide optimal parallel solutions to several link distance problems set in trapezoided rectilinear polygons. All our main 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 optimally compute (on a CREW PRAM) the link diameter, the link center and the central diagonal of a rectilinear polygon.

*This research work was partially supported by TFR.
†The research of the third author was partially supported by Natural Sciences and Engineering Council of Canada.
*Independently, Schuierer [31] obtained a linear-time sequential algorithm.

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 [32]). The study of link distance problems has recently attracted a lot of attention in computational geometry, see e.g. [2, 3, 6, 10, 11, 12, 17, 20, 28, 29, 32, 33]. 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 problems 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 Maheshwari 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 rectilinear link distance problems set in trapezoided rectilinear polygons [27, 29]. 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 [13, 14] for trapezoidation 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. 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. A corresponding path is called a *minimum rectilinear link path* and its computation arises e.g. in robotics and VLSI problems.

Unless otherwise specified all 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, 29]. de Berg [3] proposed an optimal sequential algorithm for computing a minimum rectilinear link path between two vertices in a rectilinear polygon P. Parallel algorithms for optimally computing a rectilinear link path between

[†] Rectilinear polygons are also called *orthogonal polygons*, *isothetic polygons* and *rectanguloid polygons* in the literature.

two points inside a trapezoided rectilinear polygon have been proposed by [23, 24]; they run in $O(\log n)$ time using $O(n/\log n)$ processors on the EREW PRAM.

A 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 [33, 34]. 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. It also yields 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 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 [33]. Its analog 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 [32, 33], 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 the 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 (or link distance)* 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 interval called the *vertex interval on* d. These intervals are 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. This algorithm enables us 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] presents 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); where work is the product of the number of processors and parallel time. 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. Independently, a linear-time algorithm has been discovered by

Schuierer [31]; (vertex-vertex queries can be answered in constant time). 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 provide optimal CREW-PRAM 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 a simple polygon [20, 10, 17, 25]. Nilsson and Schuierer [25] 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 develop an optimal parallel implementation of their algorithm. The algorithm takes $O(\log^* n \log n)$ time and performs $O(n)$ 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 and may provide a tool for designing other optimal parallel algorithms from existing sequential ones.

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, 25, 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 a simple polygon, Djidjev et al. [10] introduce the concept of a central diagonal of a simple polygon; subsequently termed *splitting chord* by [25] in the context of rectilinear polygons. Our results are $O(\log^* n \log n)$ time and $O(n)$ work-optimal (CREW-PRAM) 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 [30], tree operations including the Euler tour technique [36], tree contraction and traversals [18], point location in planar subdivision [35] and parenthesis matching [4, 22]. For the other tools such as parallel prefix, list ranking and doubling, see [16, 18].

The paper is organized as follows: in Section 2 we introduce some notation and state some preliminaries. In Section 3 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. In Section 4 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 5. In Section 6 we discuss the parallel determination of the link diameter. In Section 7 we discuss an optimal CREW-PRAM algorithm for computing the link center, link radius, and a central diagonal. In Section 8 we summarize the results obtained in this paper and discuss a few open problems.

2 Preliminaries

Throughout, all geometric objects (polygons, paths, boundaries, 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 also used to denote the (closed) region of the plane enclosed by P. Let $bd(P)$ denote the boundary of P. If u and v are two points on $bd(P)$ then the clockwise boundary of P from u to v is denoted as $bd(u, v)$. Two points of P are said to be *(rectilinearly) visible* if there is a (rectilinear) line segment joining them that lies totally inside P. A vertex u of P is *reflex* if the internal angle at that vertex is greater than 180°, *convex* otherwise.

A line segment c interior to P is a *chord* if c is axis parallel and the end points of c are on $bd(P)$. A *histogram* is a rectilinear polygon that has one distinguished edge, called its *base*, whose length is equal to the sum of the lengths of the other edges that are parallel to it. We define a histogram H inside P having an axis parallel chord c in P as its base to be the maximum area histogram interior to P with c as its base. A *window* is a maximal segment of the boundary of the histogram H which is not part of the boundary of P. A window w of H partitions P into two subpolygons. The subpolygon of P, not containing H, is referred to as the *pocket* associated with w.

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 the polygon is reached. These extensions can be computed by the algorithms of Goodrich [13] and Goodrich *et al.* [14]. We insert the extension points of each edge as vertices on the boundary of the 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 Optimal parallel algorithms for computing link distances

In this section 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* in P is a horizontal or vertical closed straight-line segment within P joining a point 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 $LD(d, d')$.

3.1 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 a 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 minimal diagonals that lie within the bottom polygon.
2. Compute the tree T dual to 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 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 $md(u) := 2d(u) - 1$ 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 3.1 *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 either crosses $f(u)$ vertically or has a vertical link starting from $f(u)$ towards d in U. To reach 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 $2 + 2d(f(u)) - 1$ 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 the diagonal associated to it 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 containing d as part of its boundary by using the Euler tour technique of [36]. It can be done optimally in logarithmic time [36].

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 (see, Figure 1). 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 [30] 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 vertically remote $lca(p)$ 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 vertically remote 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 [36].

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.

3.2 Link distances to all vertices

Using the results of Theorem 3.1, 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 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 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.

Assume that $lca(p_v)$ is e'. Any link path from v to d crosses e' (see, Figure 2). By the definition of e', if a minimum link path to d 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 argument immediately implies 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: An 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 [30] 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 3.2 *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 first shoot in the four rectilinear directions towards the boundary of P thereby computing the horizontal and vertical chord containing p. By Theorem 3.2 we can compute the link distances for the horizontal and the vertical chord containing p 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. [33]). 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 5.3 we will show that for a given rectilinear segment in P all vertex intervals and the corresponding link distances can be optimally computed in $O(\log n)$. It then follows,

Theorem 3.3 *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 3.4 *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.*

Corollary 3.5 *Let s be a rectilinear segment in a trapezoided rectilinear simple polygon P. Then the L_1 shortest path from any point or from s to all vertices of P can be computed optimally in $O(\log n)$-time using the EREW PRAM with $O(n/\log n)$ processors.*

4 An optimal parallel algorithm for computing window partition

The window partition of a simple polygon was introduced by Suri [33] as a technique for preprocessing a polygon that leads to efficient sequential algorithms for solving a number of link distance problems. Window partitions have 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. Next remove the histogram $H(d)$ from P; this results in a partition of 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 window of $H(d)$ 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 a given diagonal d. As a consequence this fundamental tool is now available in particular for solving 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; for an illustration see Figure 3. The correctness of Algorithm 3 follows from following lemmas.

Lemma 4.1 *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 4.2 *Let P be a rectilinear polygon and d a diagonal of P. Then the bracket sequence computed by Algorithm 3 is well-formed.*

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 matching parenthesis by the algorithm of [4] or [22].
7. Construct the circular list of the vertices belonging to each histogram in the histogram partition.

Algorithm 3: Algorithm for computing the histogram partition

Proof: The proof is by induction on the link distance (or covering radius) 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, in Step 3, Algorithm 3 computes one well-formed bracket pair for P. The result thus follows in this case.

Now let P be a rectilinear polygon with link distance L from d. Assume that for all polygons P' and diagonals d' in P' with link distance at most $L-1$ the result holds. Then compute the histogram from d. This induces a number of windows in the histogram together with their associated pockets. The link distance from a window 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 window. Thus the brackets assigned in each pocket are also brackets for P computed from d. A window 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. On exiting the window 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 (as produced in Step 3) 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 4.3 *Vertices p_i and p_j form 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 3. Opening and closing parenthesis to the appropriate vertices can be assigned in $O(\log n)$ time using $O(n)$ operations. Parenthesis matching can be done by the algorithm of [4, 22] in $O(\log n)$ time using $O(n/\log n)$ operations. Using the standard doubling technique, we can construct the circular list of vertices belonging to each histogram in $O(\log n)$ time using $O(n/\log n)$ 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 4.4 *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.*

5 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.

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.

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 due to de Berg [3] enables us to compose the two link distances.

Lemma 5.1 *(de Berg [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 $d_v = d(v,e)$. The following lemma of de Berg [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 5.2 *(de Berg [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 5.3 *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.*

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: The correctness of the algorithm follows from Lemma 5.2. 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 5.4 *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.*

Proof: Suppose that we can locate the vertices v_{next} and v_{next2} of s and t, then we can compute their intervals on e by Algorithm 4 and compose them by Lemma 5.1 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}. (Analogously for rays entering through the base.) 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 Tamassia and Vitter [35]. 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. Algorithm 6 describes the procedure for answering of queries. Algorithm 6 requires two procedures which are also described in detail. 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.

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 window.
3. Construct a data structure to answer lowest common ancestors queries in T_H by the algorithm of Schieber and Vishkin [30].
4. Construct a planar point location data structure over the histogram partition of P by the algorithm of Tamassia and Vitter [35].
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 [36] 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 [30].
   (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.

Lemma 5.5 *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: The number of histograms in the histogram partition of P is linear. Therefore, the number of nodes in T_H is at most $O(n)$. The lowest common ancestor data structure of Schieber and Vishkin [30] and the planar point location data structure of Tamassia and Vitter [35] 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 [36] in $O(\log n)$ time using linear number of operations and storage. We need a suitable representation

1. Locate the histograms containing s and t by the algorithm of Tamassia and Vitter [35].
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 [30].
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 window in H of the pocket containing 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 (respectively, t) on w_s (respectively, w_t) by Algorithm 7.
5. Let the link distance from s to w_s be d_s and the link distance from t to w_t be d_t. The link distance $LD(s,t)$ between s and t is $d_s + d_t + \Delta$, 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 5.3 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 (Step 5f) 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 [35].
   (c) If the label of x' is same as that of u, then $u = x'$ and the binary search ends.
   (d) Otherwise ** Label of $u \neq$ label 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. Decide 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 the 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. Similarly, let endpoints of the fast and the slow interval from t on w_t be (t_1, t_2) and (t_1, t_3), respectively. Let t_2t_2' and t_3t_3' be the maximal horizontal segment through t_2 and t_3 in H. Let w_s' be the maximal vertical segment containing w_s in H.
   If (s_2s_2' intersects w_t' or t_2t_2' intersects w_s') then $\Delta = 1$
   else if (s_3s_3' intersects w_t' or t_3t_3' intersects w_s') 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 $\Delta = -1$
   else if s_2s_3 and t_1t_2 are visible then $\Delta = 0$
   else if s_1s_2 and t_2t_3 are visible then $\Delta = 0$
   else if s_2s_3 and t_2t_3 are visible then $\Delta = 1$
   else if s_1s_3 and t_1t_3 are not visible then $\Delta = 2$.

Algorithm 8: Algorithm for computing Δ.

of T to support Euler tour technique. For each node v_{next} in T, we require all of its children to be in a linked list. It is possible to compute the appropriate linked lists by using simple geometric properties of rectilinear polygons as mentioned in Step 4 of Algorithm 4, modifying Step 5 of Algorithm 1, and the trapezoidation of P. □

Lemma 5.6 *A single processor can answer link distance queries in $O(\log n)$ time by executing Algorithm 6.*

Proof: To prove the correctness, we show that Algorithm 7 correctly computes the required intervals, since the proof of the rest is straightforward. In the following we show the correctness of Algorithm 7.

Observe that the label of a node in T_{next} is its level number in T_{next}. Hence, the labels of nodes in T_{next} decrease monotonically along any path from the leaf node to the root of T_{next}. We are interested in locating a node with a specific label (i.e. the label of u) along the path from v to the root of T_{next}. From the above observations, it can be seen that the binary search described in Algorithm 4 locates the appropriate node in $O(\log n)$ time.

Now we analyze the complexity of the algorithm. The histograms containing the query points s and t can be located in $O(\log n)$ time by the algorithm of [35]. Lowest common ancestor of two nodes in a tree can be computed in constant time [30]. 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 windows of the pockets can be achieved within the claimed complexity bound. □

Theorem 5.7 *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 5.4, 5.5 and 5.6. Now we show the correctness. If the query point are located in different subpolygons of e then the correctness follows from Lemma 5.4. 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 windows 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 $O(n)$ we obtain a linear time sequential algorithm for computing the link query data structure. 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 (see [23]). Independently, a linear-time algorithm has also been proposed by Schuierer [31].

Theorem 5.8 *A data structure can be computed in linear time, which allows rectilinear link distance queries between two query points in an n-vertex rectilinear polygon to be answered in $O(\log n)$ time.*

6 Link diameter

In this section we present a work-optimal parallel algorithm for computing the link diameter of an n-vertex simple rectilinear polygon P. Our approach yields a general technique for designing optimal parallel algorithms. 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 [32]. In addition to computing the value $Diam(P)$ we are interested in producing a pair of vertices whose link distance realizes the diameter and a minimum link path of length $Diam(P)$ between these.

A sequential method for computing the rectilinear link diameter of a rectilinear polygon was developed by de Berg [3] (see Algorithm 9); it runs in $O(n \log n)$ time. A parallel algorithm for computing link diameter can be designed by providing a parallel implementation of each step of Algorithm 9 as follows.

In Step 2, an appropriate cut segment e is computed from the given horizontal and vertical trapezoidation of P. In Step 4, the value of M is computed by analyzing several cases in the algorithm of [3]. The majority of the computation entails the computation of intersections of slow and fast intervals on e (from the vertices in the subpolygons P_1 and P_2) and the computation of dominance relationships among the end points of slow and fast intervals. It can be seen that intersections and dominance pairs can be computed in parallel by using parallel prefix operations. So this naive parallel algorithm runs in $O(\log^2 n)$ time using $O(n/\log n)$ processors.

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 d_1:=max $\{LD(v,e)|$ where v is a vertex of $P_1\}$ and compute d_2:= max $\{LD(v,e)|$ where v is a vertex of $P_2\}$, 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 and Schuierer [26] gave a linear time algorithm for this problem. Based on the following observations they presented an algorithm which differs from de Berg's algorithm in only the third Step. They observe that the value of M is at least $d_1 + d_2 - 1$. W.l.o.g. one may assume that $d_1 \geq d_2$ then $Diam(P_2) \leq 2d_2 + 1$. Thus there is no need to recur on P_2 if $Diam(P_2) \leq d_1 + d_2 - 1$. This implies that these values for $Diam(P_2)$ which are of interest for the diameter computation lie in the range from $d_1 + d_2 - 1$ to $2d_2 + 1$. They [26] presented a linear time (in the size of P_2) algorithm for determining whether $Diam(P_2) \leq 2d_2 - 1 \leq M$ and for computing the exact value of $Diam(P_2)$ in case $D(P_2) > 2d_2 - 1$. The recurrence relation for the time complexity $T(n)$ of their entire algorithm is $T(n) = T(\frac{3}{4}n) + O(n)$ which is $O(n)$.

We analyze the parallel complexity of the above linear time algorithm. Assume that we can determine in optimal parallel work either whether $Diam(P_2) \leq M$ or, if this is not the case, the exact value of $Diam(P_2)$. Then a straightforward parallel implementation of the above sequential algorithm will give rise to the following recurrence for the time complexity $T(n)$ of the parallel algorithm; $T(n) = T(\frac{3}{4}n) + O(\log n) = O(\log^2 n)$. 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 develop a more complex $O(\log^* n \log n)$ time implementation of the algorithm which is work optimal. This implementation may be seen as a general technique for implementing algorithms of the same sequential flavor. The sequential algorithm has a time complexity which is described by a recurrence of the form $T(n) = T(cn) + O(n)$, with $c < 1$. A sequential execution of the algorithm can be seen as traversing a path from the root of a binary tree to a leaf node. Linear (in the subproblem size) time is spent at level i to determine which of the two subproblems at level $i+1$ needs to be solved. For the parallel execution, each of the $O(\log n)$ sequential recursion steps can be implemented optimally in logarithmic (in the size of the subproblem) parallel time. Furthermore, the subproblems are either known in advance or can be determined at no additional cost (this is clarified later).

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 level 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 we can perform the computation simultaneously in several levels instead of performing it level by level (in a sort of speculative way). Thus unlike the sequential computation the parallel computation performs work on more than one node per level of the tree. While the total work performed by the sequential algorithm is only linear the difficulty arises because a parallel algorithm does not know in advance which path of the recursion tree will be taken. Notice that this is not a straight-forward application of Brent's principle.

The algorithm consists of two phases. During the first phase we sequentially process the first $O(\log^* n)$ levels using $O(n/\log^* n \log n)$ processors on each level. After this step the problem size is reduced to at most $O(n/2^{\log^* n})$. Then, in a second phase, we employ a parallel algorithm to solve all subproblems on several adjacent levels at the same time. This is repeated $O(\log^* n)$ times taking $O(\log n)$ time each and requiring $O(n/\log^* n \log n)$ processors in total. To prove the claimed complexity bounds we begin by showing that the first phase can be executed in $O(\log^* n \log n)$ time using $O(n/\log^* n \log n)$ processors.

In the first phase of the algorithm we solve one subproblem on each of the first $O(\log^* n)$ levels. We allocate for each level $O(n/\log^* n \log n)$ processors which perform their work level by level. Analysis of the first phase shows that $O(\log^* n \log n/2^i)$ time is taken to execute levels $2^i \leq \log^* n$ and for the levels $\log\log^* n < i \leq \log^* n$, $O(\log n)$ time is taken. Hence, the total time taken to execute all levels (level 0 to $\log^* n$) sums to $O(\log^* n \log n)$.

The second phase of our parallel algorithm has $O(\log^* n)$ stages. During the ith stage we solve all subproblems associated with a subtree of the recursion tree. The subtree of the ith stage is rooted at a vertex of level $a = 2^{2^{\cdot^{\cdot^{\cdot^{2^2}}}}}$ (i $2's$) and it consists only of nodes on levels a to 2^a. The total size of the subproblems associated with this subtree is easily seen to be $O(n)$. For the ith stage let T be the recursion subtree rooted at v, where v is a vertex at level a. Since the total size of the subproblems associated with T 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. It is crucial that we know the subproblems that need be worked on. For the diameter algorithm this determination is given below. Each node of T 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, which child will be active during the next recursion step. We can find the path from v to a leaf of T of the active nodes in the recursion for example by using the standard method of doubling. 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^a to 2^{2^a}. 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. Since there are in all $O(\log^* n)$ levels, the total complexity of the above algorithm will be $O(log^* n \log n)$ time using $O(n/\log n)$ processors. Since after the first phase, the size of the problem left is only $O(n/2^{\log^* n})$, so the number of the processors required in the second phase is only $O(n/\log^* n \log n)$.

The analysis given so far is general and does not depend on the particular geometric properties of the diameter problem. We expect it to be useful for any algorithm whose sequential computation follows the above pattern. What remains to be shown for the computation of the link diameter is that all subproblems required at each level of the recursion can be determined within the claimed complexity bounds. Let a be a level in the recursion tree and T be the associated subtree rooted at some node on that level. Let $b = 2^a$ be the leaf level for T. Denote by P_i the polygon associated with $root(T)$. For phase 2 we took $b = 2^a$ and solved the entire subproblem for T in time $O(\log n)$ using $O(n/\log n)$ processors. We need to identify all subproblems in T in the same time/processor bounds. We do this level by level taking $O(1)$ time per level and thus in all taking $O(\log n)$ time using sufficient number of processors. It is known that there exists a diagonal in P splitting the polygon associated with $root(T)$ into two subpolygons of sizes (i.e. number of vertices) no less than $1/4s$ and no more than $3/4s$, where s is the number of vertices in P_i [3].

Assume that the dual tree of the horizontal and vertical trapezoidation of the polygon P are available. Note that the dual trees need not be binary trees. We simplify the exposition by first assuming the (arbitrary) CRCW computation model. Associate a processor with each diagonal; the processor computes in O(1) time the number of vertices in each of its two subpolygons. A diagonal is a *candidate diagonal* if the number of vertices in the two subpolygons fall within the bounds required. Processors associated with candidate diagonals are called *candidate processors*. All candidate processors write their name into a common location. One of them will succeed and by using the common read all candidates are informed of the so selected diagonal. Each vertex computes in which of the two subpolygons it lies and updates it vertex number (subtracting the number of vertices cut off in the other portion). All diagonals which cross the selected diagonal are also updated in the constant time. It is easy to observe that the above algorithm runs in the desired complexity on the CRCW PRAM.

Now we discuss the CREW implementation. Clearly, several candidates may exist. We need to show that these candidates can select one diagonal among themselves in $O(1)$ time. We claim that there are only a constant number of connected components containing such candidates in the dual trees. In each connected component, to select one candidate, we choose the highest (in the tree sense). Each node in the dual tree determines whether its parent is also a candidate in which case it does nothing; otherwise, it is the highest and it is the selected candidate in its connected component. We run this test on each of the constant number of connected components. As will be shown in the following lemma the number of connected components and thus the number of selected candidates is constant.

Since the dual trees are not constant degree trees, we need to convert them to binary trees in order to avoid concurrent writes. This can be done by the algorithm in [16]. Hence, each node in the dual tree can determine, in constant time without using any concurrent writes, whether it is the selected node in its component. The correctness follows from the following lemma.

Lemma 6.1 *In each dual tree (of a horizontal and vertical trapezoidation) there exist only a constant number of connected components of nodes representing diagonals which split the polygon into no less than* $1/4th$ *and no greater than* $3/4th$ *of its size.*

Proof: Let n be the total number of vertices in the polygon. Assume that the dual trees (corresponding to horizontal and vertical trapezoidation) are rooted at some node. First we show that along any path from a leaf to the root in the dual tree, there is at most one connected component of nodes representing diagonals which split the polygon appropriately.

Let a, b and c be three nodes along a leaf-to-root path, encountered in that order. Furthermore, assume that the diagonals corresponding to the nodes a and c are candidate diagonals. Then there are at least $1/4n$ vertices in the subpolygon split by the diagonal corresponding to a not containing the diagonal corresponding to b.Analogously for c. Thus, by definition, the diagonal corresponding to b is a candidate diagonal.

There are at least $1/4n$ nodes in each subpolygon corresponding to the subtree rooted at the root of each connected component. Since the dual tree is of size n, it follows that the number of connected components is constant. □

Now we show that we can solve each subproblem in optimal work. Using algorithm of Section 5 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 the exact value of $Diam(P_2)$. The overall parallel complexity of this procedure is $O(\log n)$ time using $O(n/\log n)$ processors. The main result of this section is stated in the following theorem.

Theorem 6.2 *The rectilinear link diameter of an n-vertex simple rectilinear polygon can be computed in $O(\log^* n \log n)$ time using $O(n/\log^* n \log n)$ processors on the CREW-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 an EREW-PRAM.*

The above technique is general and works for any sequential algorithm whose run time can be analyzed in the same way. The designer of the parallel algorithm must take care that the subproblems are available for phase 2 of the algorithm.

7 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 central diagonal for simple polygons. 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 [10]. Furthermore they showed that the covering radii of a central diagonal differs from the link radius R of P by at most one. Thus 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 any 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] (this bound has independently been claimed in [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. [25] 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 is 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 the results of Section 6. Let v_1, v_2 be determined as a vertex pair realizing the diameter as its link distance; again these can also be found using the results developed in Section 6. The segment at the lower median index position on the path from v_1 to v_2 has been shown to be in the vicinity of a central diagonal [25]. Finding a central diagonal then reduces to computing covering radii to both sides from at most four chords. This is done using the optimal parallel interval computation given in Section 5. Except for the diameter computation all steps can be performed in optimal parallel time. Since our diameter computation takes $O(\log^* n \log n)$ time we get,

Theorem 7.1 *A central diagonal in an n-vertex simple rectilinear polygon can be computed in $O(\log^* n \log n)$ time using $O(n/\log^* n \log n)$ processors on the CREW 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 arise when locating a transmitter so that the maximum number of retransmission needed to reach any point in a polygonal region is minimized, or when choosing the best location for a mobile unit minimizing the number of turns needed to reach any point in a polygonal regions. In [10] and [17] an $O(n \log n)$ time algorithm was given to determine the link center of a simple n-vertex polygon. In [25] a linear time sequential algorithm for computing the rectilinear link center of P was presented. The *rectilinear* link center of a rectilinear polygon is obtained by using the rectilinear link distance. We show that the algorithm of [25] can be parallelized by using the techniques developed in this paper. In the following 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 one of 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 split P into two subpolygons, P_1 and P_2. Let d_1 (or d_2) denote the maximum link distance from d to all vertices in P_1 (respectively, P_2). Without loss of generality assume that d is vertical, 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 lying in P_1 and in 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.

The computation of the link center in a histogram H is based on the following. 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 induced by 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 contained 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 not elaborated on here; and finally the computation is reduced to that of computing the link center in a histogram. In the following we state these procedures that are required by the algorithm in [25] to compute the link center and briefly discuss their parallel implementation.

Central diagonal d : Compute the central diagonal using the results of Theorem 7.1.

1-visibility (or histogram) and 2-visibility polygons from d: We first compute the window partition of the polygon with respect to d using the algorithm in Section 4. From the window partition we can easily compute the 1-visibility and the 2-visibility polygon from d.

Slow and fast intervals: We need to compute slow and fast intervals from vertices inside the pockets to their respective windows or to the diagonal d. The slow and fast intervals can be computed by Algorithm 4.

Sweeping a segment in the interior of P: In a few cases, during the computation of the rectilinear link center, we need to compute the first point where an internal segment s, swept, horizontally or vertically, inside P, intersects $bd(P)$. The desired point of intersection can be computed by first locating the edges of P which possibly can have an intersection when the segment is swept. After locating all such edges, pick the one which is at the closest to s.

Intersection of histograms: In order to compute the link center, we need to compute the intersection region of a constant number of histograms, where the base of all of them is either horizontal or vertical. Without loss of generality assume that their bases are horizontal. We compute the intersection region with linear (in the size of the histograms) amount of work in parallel as follows. First sort the vertices with respect to their x-coordinate. This can be done with linear amount of work, since the vertices for each histogram are already sorted with respect to their x-coordinates. Now sweep a vertical line from the left to the right, where the sweep line traces the region of intersection of the histograms. The computation of the sweep line can be simulated easily in parallel in linear amount of work. Hence, we can compute the intersection region of the histograms within the desired complexity.

Intersection of intervals: During the computation of the link center, we need to compute the intersection of the fast and slow intervals from vertices in P_1 (or P_2) on d and from the vertices in the pockets to their corresponding windows. The above intersection information can be computed from the knowledge of the end points of the intervals. The interval endpoints can be computed by Algorithm 4.

Using the above steps, the rectilinear link center of a rectilinear polygon can be optimally computed. We omit further details and state the result in the following theorem.

Theorem 7.2 *The rectilinear link center of an $n-$vertex rectilinear simple polygon can be computed in $O(\log^* n \log n)$ time using $O(n/\log^* n \log n)$ processors on the CREW-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 7.3 *The rectilinear link radius of an $n-$vertex rectilinear simple polygon can be computed in $O(\log^* n \log n)$ time using $O(n/\log^* n \log n)$ processors on the CREW PRAM.*

8 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 optimal CREW-PRAM solutions for the link diameter, link center, link radius, and central diagonal problems. The total work performed by each of these algorithms is $O(n)$.

Acknowledgements : Authors gratefully acknowledge Torben Hagerup for suggesting an improvement of the diameter algorithm over an earlier version of this paper. The authors would like to thank the referee for suggestions which resulted in an improved presentation of this paper.

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 Computational 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 behavior of the greedy triangulation*, Linköping Studies in Science and Technology, Ph.D. Thesis, No. 74, Linköping University, Sweden, 1986.

[22] C. Levcopoulos and O. Petersson, *Matching parenthesis in parallel*, Discrete and Applied Mathematics, 1992.

[23] A. Lingas, A. Maheshwari and J.-R. Sack, *Optimal parallel algorithms for rectilinear link distance problems*, Technical Report SCS-TR-213, School of Computer Science, Carleton University, 1992.

[24] K. M. McDonald and J. G. Peters, *Smallest paths in simple rectilinear polygons*, IEEE Transactions on Computer-Aided Design, Vol. 11, No. 7, 1992, pp. 864-875.

[25] 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, Springer Verlag, eds. F. Dehne, J.-R. Sack, and N. Santoro, 1991, pp. 249-260.

[26] B.J. Nilsson and S. Schuierer, *Computing the rectilinear link diameter of a polygon*, Computational Geometry- Methods, Algorithms, and Applications, Lecture Notes in Computer Science 553, Springer Verlag, eds.: H. Noltemeier and H. Bieri, 1991, pp. 203-216.

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

[28] 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.

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

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

[31] S. Schuierer, *Rectilinear path queries in a simple rectilinear polygon* , STACS'93, Lecture Notes in Computer Science, vol. 665, Springer-Verlag, 1993, pp. 282-293.

[32] 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.

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

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

[35] 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.

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