Parallel Techniques for Paths, Visibility, and Related Problems (Thesis)

Ziyi Danny Chen

Report Number:
92-051
PARALLEL TECHNIQUES FOR PATHS, VISIBILITY, AND RELATED PROBLEMS

Ziyi Danny Chen

CSD-TR-92-051
August 1992
PARALLEL TECHNIQUES FOR PATHS, VISIBILITY, AND RELATED PROBLEMS

A Thesis
Submitted to the Faculty

of

Purdue University

by

Ziyi Danny Chen

In Partial Fulfillment of the
Requirements for the Degree

of

Doctor of Philosophy

August 1992
To the memory of my father
ACKNOWLEDGMENTS

There are many people that I would like to thank. First, I would like to express my greatest appreciation to my major professor Mikhail J. Atallah. He has always given me the freedom to develop my own research interest and at the same time provided all the support, guidance, and encouragement I needed. Learning how to do research from him and discussing problems with him have become some of my most joyful experience at Purdue. Furthermore, I am greatly indebted to Professor Greg N. Frederickson and Professor Susanne E. Hambrusch, for serving as my committee members and for giving me many needed suggestions and helps during my study at Purdue. I also like to thank Professor Christoph M. Hoffmann for serving as my committee member.

I would like to thank the people in the CS department with whom I had discussed research problems, especially Professor Chandrajit Bajaj, Professor Chia-Hoang Lee, Scott McFaddin, Jyh-Jong Tsay, and Tony Tu. I would also like to thank all the people who have been my friends and helped me during my stay at Purdue. In particular, I thank Peter Ng, Thomas Tsang, Rose Ng, and Hiu Leung for many very needed helps during my early stage at Purdue and for many warm dinners and exciting badminton games, my office mates JiaXun Yu and Ching-Shoei Chiang for all the interesting chats and for giving me various helps, my office mate Susan Stratton for tolerating my frequent interruptions and kindly answering all my English questions, and Jindong Chen, Jung-Hong Chuang, Peggy Fahl, D.-J. Guan, Ajay Gupta, Hsin Pan, Bonita Rais, and many Chinese students for their friendship.

I would certainly like to thank my “American parents in law,” Dick and Gladys, for their love. I enjoy my weekend visits to their house, which always make me feel like being at home.
I can never thank enough my father and mother. They gave me the most love and support. My father’s love for education was my first motivation to pursue a Ph.D. degree. His inspiration, expectation, and confidence have been and will always be one of the greatest sources of support to me. My mother is great; she devoted all her time to raising and educating her eight children. I owe a great deal to my grandfather YueYe, whom I never have a chance to meet. It was his lifetime hard work that provided for my education for years. Many of my thanks go to my sister HuaYing; she and I shared the same dream of getting a higher education. She gave me the best support she could come up with. I thank all my brothers and sisters for their encouragement to my study. I cannot forget it was my uncle GengOn and aunt May who created the opportunity for me to study in the United States and provided a place for me to stay during my first three years in San Francisco.

Finally, I am deeply grateful to my dear wife Xiaobo, for her love and patience, for her understanding and appreciation of my work, and for driving many times on the icy and endless road between Detroit and West Lafayette to bring her support and encouragement to me.

This research was partially supported by the Office of Naval Research under Grants N00014-84-K-0502 and N00014-86-K-0689, the Air Force Office of Scientific Research under Grant AFOSR-90-0107, the National Science Foundation under Grant DCR-8451393, and the National Library of Medicine under Grant R01-LM05118.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF FIGURES</th>
<th>vii</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>ix</td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 The Parallel Computational Model</td>
<td>3</td>
</tr>
<tr>
<td>1.2 Some Basic Parallel Operations and Techniques</td>
<td>6</td>
</tr>
<tr>
<td>1.3 Previous Work</td>
<td>9</td>
</tr>
<tr>
<td>1.4 Our Thesis</td>
<td>12</td>
</tr>
<tr>
<td>2. RECTILINEAR SHORTEST PATHS WITH RECTANGULAR OBSTACLES</td>
<td>15</td>
</tr>
<tr>
<td>2.1 Preliminaries</td>
<td>18</td>
</tr>
<tr>
<td>2.2 Computing a Staircase Separator</td>
<td>26</td>
</tr>
<tr>
<td>2.3 Other Building Blocks</td>
<td>30</td>
</tr>
<tr>
<td>2.4 Computing the Lengths Matrix $D_p$ When $</td>
<td>P</td>
</tr>
<tr>
<td>2.5 Path Lengths between Arbitrary Points</td>
<td>36</td>
</tr>
<tr>
<td>2.5.1 Some Useful Observations</td>
<td>36</td>
</tr>
<tr>
<td>2.5.2 The $B(P)$-to-$V_R$ Path Lengths</td>
<td>44</td>
</tr>
<tr>
<td>2.5.3 The $V_R$-to-$V_R$ Path Lengths</td>
<td>45</td>
</tr>
<tr>
<td>2.5.4 Path Lengths with Arbitrary Query Points</td>
<td>51</td>
</tr>
<tr>
<td>2.6 Path Lengths When $</td>
<td>P</td>
</tr>
<tr>
<td>2.7 Computing the Actual Paths</td>
<td>55</td>
</tr>
<tr>
<td>2.8 A Note on the Sequential Time Complexity</td>
<td>58</td>
</tr>
<tr>
<td>3. VISIBILITY OF A SIMPLE POLYGONAL CHAIN FROM A POINT</td>
<td>61</td>
</tr>
<tr>
<td>3.1 Preliminaries</td>
<td>63</td>
</tr>
<tr>
<td>3.2 An Overview of the Algorithm</td>
<td>66</td>
</tr>
<tr>
<td>3.3 Visibility Chains and Their Intersections</td>
<td>68</td>
</tr>
<tr>
<td>3.3.1 Simple Geometric Facts</td>
<td>68</td>
</tr>
<tr>
<td>3.3.2 Simple Computational Observations</td>
<td>73</td>
</tr>
</tbody>
</table>
3.3.3 The Relative Positions of $A_i$ and $VIS(C_i)$ ........................................ 74
3.3.4 Computing the Portions of $VIS(C_i)$ Hidden by $A_i$ ............................. 78
3.4 The EREW PRAM Implementation ............................................................. 84
3.5 Applications ............................................................................................... 86

4. SOLVING GEOMETRIC PROBLEMS ON THE EREW PRAM .................. 87
   4.1 Notation and Basic Algorithm Structure .................................................. 90
   4.2 Rank Trees and Parallel Operations ....................................................... 91
   4.3 Computing the Convex Hull of Sorted Points ......................................... 94
   4.4 Triangulating a Trapezoidally Decomposed Polygon ............................. 97
   4.5 Other Geometric Algorithms ................................................................ 99

5. DETECTING WEAK VISIBILITY OF A SIMPLE POLYGON ....................... 102
   5.1 Preliminaries .......................................................................................... 105
   5.2 Some Useful Observations ..................................................................... 107
   5.3 Detecting the Weak Visibility of a Star-Shaped Polygon .......................... 113
      5.3.1 Phase 1.A ...................................................................................... 114
      5.3.2 Phase 1.B ...................................................................................... 129
      5.3.3 Computing the Union of Bad Intervals .......................................... 131
   5.4 Checking the Weak Visibility of a Polygon from an Edge ...................... 132
      5.4.1 The Preprocess ............................................................................... 133
      5.4.2 The Basic Idea ............................................................................... 134
      5.4.3 Phase 2.A ...................................................................................... 138
      5.4.4 Phase 2.B ...................................................................................... 144
   5.5 Detecting the Weak Visibility of a Simple Polygon .................................. 152
      5.5.1 The Reduction by Sack and Suri .................................................... 152
      5.5.2 Our Reduction Procedure ............................................................... 154
   5.6 Applications ............................................................................................ 162
      5.6.1 Computing Shortest Paths in a Weakly Visible Polygon ............... 163
      5.6.2 Triangulating a Weakly Visible Polygon ......................................... 166
      5.6.3 Solving the One-Cruising-Guard Problem ...................................... 167
      5.6.4 Other Applications ......................................................................... 170

6. CONCLUSION ................................................................................................. 172

BIBLIOGRAPHY ............................................................................................... 176

VITA .................................................................................................................. 187
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>A rectilinear shortest path between v and w.</td>
<td>16</td>
</tr>
<tr>
<td>2.2</td>
<td>Illustrating $MAX_{NE}(R')$ and $MAX_{SW}(R')$.</td>
<td>20</td>
</tr>
<tr>
<td>2.3</td>
<td>Illustrating Env($R'$) and the circular ordering on Bound(Env($R'$)).</td>
<td>22</td>
</tr>
<tr>
<td>2.4</td>
<td>Illustrating $B(Q)$.</td>
<td>23</td>
</tr>
<tr>
<td>2.5</td>
<td>Illustrating Monge and non-Monge matrices of path lengths.</td>
<td>24</td>
</tr>
<tr>
<td>2.6</td>
<td>Illustrating $NE(p)$ and $WS(p)$.</td>
<td>27</td>
</tr>
<tr>
<td>2.7</td>
<td>Illustrating the algorithm for Sep.</td>
<td>29</td>
</tr>
<tr>
<td>2.8</td>
<td>Illustrating array Vert.</td>
<td>31</td>
</tr>
<tr>
<td>2.9</td>
<td>Illustrating Lemma 2.8.</td>
<td>33</td>
</tr>
<tr>
<td>2.10</td>
<td>Illustrating the proof of Theorem 2.2.</td>
<td>35</td>
</tr>
<tr>
<td>2.11</td>
<td>Illustrating $U$, $U'$, $W$, and $W'$ at a node $v$ of $T$.</td>
<td>37</td>
</tr>
<tr>
<td>2.12</td>
<td>Illustrating the proof of Lemma 2.14.</td>
<td>42</td>
</tr>
<tr>
<td>2.13</td>
<td>Illustrating Lemma 2.15.</td>
<td>44</td>
</tr>
<tr>
<td>2.14</td>
<td>Illustrating the computation of (a) mode 1, and (b) mode 2.</td>
<td>49</td>
</tr>
<tr>
<td>2.15</td>
<td>Illustrating the partition of Bound($P$).</td>
<td>53</td>
</tr>
<tr>
<td>3.1</td>
<td>An example of the visibility chain VIS($P$).</td>
<td>62</td>
</tr>
<tr>
<td>3.2</td>
<td>Illustrating the definitions.</td>
<td>65</td>
</tr>
<tr>
<td>3.3</td>
<td>Illustrating the proof of Lemma 3.1.</td>
<td>70</td>
</tr>
<tr>
<td>3.4</td>
<td>Illustrating the two possible two-intersection cases.</td>
<td>72</td>
</tr>
<tr>
<td>Figure</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>3.5</td>
<td>81</td>
<td></td>
</tr>
<tr>
<td>4.1</td>
<td>94</td>
<td></td>
</tr>
<tr>
<td>4.2</td>
<td>97</td>
<td></td>
</tr>
<tr>
<td>5.1</td>
<td>103</td>
<td></td>
</tr>
<tr>
<td>5.2</td>
<td>105</td>
<td></td>
</tr>
<tr>
<td>5.3</td>
<td>112</td>
<td></td>
</tr>
<tr>
<td>5.4</td>
<td>117</td>
<td></td>
</tr>
<tr>
<td>5.5</td>
<td>119</td>
<td></td>
</tr>
<tr>
<td>5.6</td>
<td>133</td>
<td></td>
</tr>
<tr>
<td>5.7</td>
<td>135</td>
<td></td>
</tr>
<tr>
<td>5.8</td>
<td>137</td>
<td></td>
</tr>
<tr>
<td>5.9</td>
<td>141</td>
<td></td>
</tr>
<tr>
<td>5.10</td>
<td>143</td>
<td></td>
</tr>
<tr>
<td>5.11</td>
<td>145</td>
<td></td>
</tr>
<tr>
<td>5.12</td>
<td>153</td>
<td></td>
</tr>
<tr>
<td>5.13</td>
<td>160</td>
<td></td>
</tr>
<tr>
<td>5.14</td>
<td>163</td>
<td></td>
</tr>
<tr>
<td>5.15</td>
<td>164</td>
<td></td>
</tr>
<tr>
<td>5.16</td>
<td>169</td>
<td></td>
</tr>
</tbody>
</table>
ABSTRACT


This dissertation presents several deterministic techniques for efficiently solving shortest paths, visibility, and other related geometric problems in parallel. Specifically, we give techniques for designing efficient deterministic parallel algorithms for solving the following problems: computing rectilinear shortest paths that avoid rectangular obstacles in the plane, computing the visible portions of a simple polygonal chain from a point, and detecting the weak visibility of a simple polygon. These parallel techniques further enable us to solve many related geometric problems that include: computing the convex hull of a set of planar points sorted by the x-coordinates, computing the convex hull of a simple polygon, finding the kernel of a simple polygon, triangulating a set of planar points sorted by the x-coordinates, triangulating monotone polygons and star-shaped polygons, solving the all dominating neighbors problem, computing shortest paths inside a weakly visible polygon, triangulating a weakly visible polygon, checking the weak external visibility of a simple polygon, and solving the one-cruising-guard problem. Most of the parallel algorithms we obtain for these problems are optimal in the complexities of the total running time and the total amount of operations performed. Our results improve the previously known parallel algorithms for these problems in either the time complexity, or the processor complexity, or the parallel computational model required by the algorithms (in the sense of using a weaker parallel model). The parallel computational models used by our algorithms are the CREW PRAM or the EREW PRAM.
1. INTRODUCTION

Computational Geometry deals with the design and analysis of algorithms for solving problems of a geometric nature. Computational geometry has a very broad range of applicability: computer-aided design, pattern recognition, computer graphics, VLSI design, vision, robotics, operations research, statistics, and facilities planning and layout, just to name a few (see [96, 114] for references and examples). Knowledge and techniques stemming from research in the area of computational geometry have been successfully applied to develop solutions to problems in many scientific and engineering fields. Therefore, in the past two decades, computational geometry has attracted enormous research interest and has been one of the fastest-growing fields in computer science.

Real-time applications require that computational solutions to problems run as fast as possible and simultaneously minimize the amounts of computer resources used (e.g., memory and processors). Devising fast algorithms for solving problems has constantly been a big challenge to computer science. Quite often, problems occurring in scientific and engineering applications consist of large collections of geometric objects, which make the task of obtaining fast solutions to these problems even more challenging. Using sequential methods to solve these problems can be inadequate because for many of these problems, we are already at the limits of what can be achieved through sequential computation. Therefore, it is natural and important to study alternative approaches for obtaining fast algorithms. In this respect, parallel computation provides a promising avenue for reaching the goal of even faster computation.

It must be pointed out that considerable difficulties arise in attempting to translate sequential algorithmic techniques into a parallel processing environment. This is because many of these techniques seem to be inherently sequential. To solve efficiently
many geometric problems in parallel, it is often more appropriate to develop new paradigms that are suited for the design and implementation of parallel algorithms.

The focus of this thesis is on developing techniques for solving computational geometry problems efficiently in parallel. More specifically, we present parallel techniques for solving path planning, visibility, and other related geometric problems, and we show how to use these techniques to design deterministic parallel algorithms for a number of geometric problems. The algorithms we come up with are more efficient than the previously known parallel solutions to these problems, and are often optimal.

Path planning and visibility are two of the most fundamental topics in computational geometry. Path planning and visibility problems arise in many application areas, such as computer graphics, robotics, VLSI design, computer-aided design, plant and facility layout, and urban transportation (see [96, 114]). They also play very important roles in computational geometry because they frequently appear as subproblems in other problems (for example, in compacting geometric objects and in computing intersections of geometric figures). Path planning and visibility problems are related problems, in the sense that opaque objects in visibility are analogous to forbidden regions in path planning. Specifically, visibility problems tend to arise as subproblems in path planning problems, which are more general (for example, see [110]).

The rest of this chapter consists of four sections. Section 1.1 briefly describes the parallel computational model for which our algorithms were designed, and it discusses the concepts of efficiency and optimality in this model. Section 1.2 reviews certain parallel operations and techniques that have been used by numerous parallel algorithms, and it mentions some of the well-known algorithms for performing those operations. Section 1.3 gives a brief survey on some previous results in parallel computational geometry that are related to our work. Section 1.4 sketches the organization of this thesis and summarizes our main results.
1.1 The Parallel Computational Model

The commonly-accepted computational model for designing sequential algorithms is the Random Access Machine (RAM) (see [4]). The situation is different in parallel computation because there is no single parallel model. However, one of the most commonly-used model for studying parallel algorithms is the Parallel RAM (PRAM). All the work of this thesis is based in the PRAM model.

The PRAM is the synchronous parallel model which consists of a shared memory and a number of processors. It is assumed that on the PRAM, every processor can access any location in the shared memory in a constant number of time units. This assumption allows the PRAM processors to exchange information by simply reading from or writing to the shared memory. Note that it is possible, in the parallel execution of instructions, that multiple processors attempt to simultaneously access the same memory location, thus causing memory access conflicts. There are various policies on how memory access conflicts are resolved on the PRAM, and corresponding to those policies there are three different versions of the PRAM model.

The least powerful version of the PRAM is the Exclusive-Read Exclusive-Write (EREW) PRAM, which does not permit any simultaneous access to a memory location by more than one processor. A more powerful version, called the Concurrent-Read Exclusive-Write (CREW) PRAM, allows multiple processors to read data simultaneously from the same memory location, but does not permit more than one processor to write simultaneously to the same memory location. The most powerful version is the Concurrent-Read Concurrent-Write (CRCW) PRAM, which allows simultaneous accesses both for reading and writing by multiple processors to the same memory location. There are various methods for dealing with write conflicts on the CRCW PRAM. One example is that the concurrent-write accesses to a memory location are allowed only if all the processors involved try to write the same value. Another example is that among all the processors that attempt to write to the same memory location at a given time unit, one particular processor (e.g., the one with
the smallest label) is assumed to actually succeed in doing its writing. In general, simulating a CREW PRAM or a CRCW PRAM algorithm on the EREW PRAM costs a logarithmic factor slow-down. Similarly, simulating a CRCW PRAM algorithm on the CREW PRAM generally requires a logarithmic factor slow-down. For more discussion in the PRAM model, see [84].

The PRAM model has very few constraints on interprocessor communication and synchronization in comparison with the network parallel models such as arrays, hypercubes, and trees (e.g., see [98]). (In a network model, the accesses of a processor to the information in the global memory and/or in other processors are via a fixed communication network.) Therefore, the PRAM model provides several advantages in studying parallel algorithms.

First, the PRAM seems to be suitable for studying the inherent parallelism of a problem because, on the PRAM, a researcher does not need to worry about the interprocessor communication and processor synchronization on a specific network. The discovery of the inherent parallelism of a problem very often lays down the basis for designing efficient algorithms for that problem in many different parallel models. Second, a parallel paradigm designed on the PRAM is often more general than one in a particular network model (because the design of algorithms in a network model usually hinges heavily on the specific communication network of that model), and hence can be adapted for many network models. The simulation of a PRAM algorithm by a network model often retains good efficiency. For instance, many PRAM algorithms can be simulated in a hypercube model with only a logarithmic factor slow-down (see [98] for example). Third, the PRAM can simulate any network model algorithm within asymptotically the same time and processor bounds. Furthermore, it is possible to include issues faced in network models (e.g., communication and synchronization) into the framework of the PRAM. Because a PRAM algorithm can be simulated sequentially by a one-processor computer with asymptotically the same amount of total operations as that required by the PRAM algorithm, the study on
the PRAM sometimes reveals new insights or new avenues for solving a problem sequentially (e.g., see [42, 100]).

While the performance of a sequential algorithm is often measured by the worst-case time complexity (or the number of operations) taken by the algorithm, the performance of a PRAM algorithm is usually measured by two closely related criteria: the total running time and the number of processors used by the algorithm. Let the size of a problem \( \text{Prob} \) be \( n \). We denote the time and processor complexities of a PRAM algorithm \( A_p \) for solving \( \text{Prob} \) by \( T_p(n) \) and \( P_p(n) \), respectively, and the time complexity of the best sequential algorithm \( A_s \) for solving \( \text{Prob} \) by \( T_s(n) \). The total cost (or called the work) of algorithm \( A_p \) is \( W(n) = T_p(n) \times P_p(n) \). The speedup achieved by \( A_p \) is \( T_s(n)/T_p(n) \). Of course, the speedup \( T_s(n)/T_p(n) \) for \( A_p \) is obtained by paying the price of using \( P_p(n) \) processors.

If \( T_p(n) = O((\log_2 n)^{c_1}) \) and \( P_p(n) = O(n^{c_2}) \) for some constants \( c_1 \) and \( c_2 \), then problem \( \text{Prob} \) is said to be in the class \( \text{NC} \) [53, 120]. A problem in \( \text{NC} \) means that the problem can be solved very fast in the PRAM model.

One of the major tasks of parallel algorithm design for the PRAM model is to come up with parallel algorithms that are both fast and efficient, i.e., that run in polylogarithmic time and simultaneously have a \( T_p(n) \times P_p(n) \) product that is within a polylogarithmic factor of \( T_s(n) \). If a PRAM algorithm for \( \text{Prob} \) runs as fast as theoretically possible, and simultaneously has its \( T_p(n) \times P_p(n) \) product that is within a constant factor of \( T_s(n) \), then we say this algorithm is optimal. The goal of achieving fast and efficient deterministic PRAM algorithms has been elusive for many simple problems that are trivially in \( \text{NC} \). For example, topological sorting of a directed acyclic graph (DAG) and finding a breadth-first search tree of a graph are problems that are trivially in \( \text{NC} \) and yet it is not known whether either of them can be solved deterministically in polylogarithmic time using a quadratic number of processors. Obtaining optimal PRAM algorithms is, of course, an even harder task than finding fast and efficient PRAM algorithms. In computational geometry, for quite a period of time, the convex hull problem on planar points was one of the few problems for which
an optimal PRAM algorithm was known [2, 17, 18]. Recently, the "cascading divide-
and-conquer" technique [16, 43] has yielded a long list of optimal parallel algorithms
for geometric problems. This thesis shows that several path planning problems and
visibility problems, as well as a number of other related problems, have fast and
efficient parallel solutions, and many of these problems can even be solved optimally.

Of course, not every geometric problem is known to be in NC. In fact, there are
geometric problems (called P-complete problems) for which NC algorithms probably
do not exist (for example, see [13]). From a practical point of view, a problem not
having an NC algorithm does not necessarily mean that the problem is not interesting
from a parallel computation perspective. But this is not the main focus of this thesis.

This thesis concentrates on designing fast and efficient NC algorithms for solving
geometric problems. All our algorithms in this thesis use either the CREW PRAM
or the EREW PRAM model. For many interesting problems, their time lower bound
on the CREW PRAM (and hence on the EREW PRAM) is $\Omega(\log_2 n)$, regardless of
the number of processors involved. Such a lower bound even holds for some of the
problems that appear to be trivial in sequential computation (e.g., see [47, 82]).

Throughout this thesis, binary logarithms are used unless otherwise specified.

1.2 Some Basic Parallel Operations and Techniques

In this section, we review several basic parallel operations and techniques that
have been widely used in parallel algorithm design, and mention some of the well-
known parallel algorithms for performing these operations. Our algorithms in the
later chapters frequently make use of these operations and techniques. For the details
of the parallel algorithms on these operations, the reader is referred to [7, 64, 82, 98].

1. Parallel Prefix: Given an ordered sequence of $n$ elements $a_1, a_2, \ldots, a_n$ which
are all from a universal set $A$, and given a binary associative operator $\oplus$ in $A$, the
prefix sums on this sequence are defined by

$$s_i = a_1 \oplus a_2 \oplus \cdots \oplus a_i, \ i = 1, 2, \ldots, n.$$
Sequentially, the prefix sums can be easily computed in $O(n)$ time. The problem of computing the prefix sums in parallel is called \textit{parallel prefix}, and can be solved optimally in $O(\log n)$ time using $O(n/\log n)$ processors on the EREW PRAM \cite{88, 89}.

2. \textit{Parallel Sorting} and \textit{Parallel Merging}: Given a set of $n$ elements in which a total order is defined, the sorting problem is to arrange these $n$ elements in a sorted order (either nondecreasing or nonincreasing). Sorting $n$ elements sequentially requires $\theta(n \log n)$ time. Sorting in parallel can be done optimally in $O(\log n)$ time using $O(n)$ EREW PRAM processors \cite{5, 43}. The best parallel sorting algorithm is obtained by Cole \cite{43}. Given two sorted sequences of $n$ elements each, the merging problem is to obtain a sorted sequence of $2n$ elements from these two sequences. The merging problem requires $O(n)$ time sequentially. The parallel merging problem can be solved on the CREW PRAM in either $O(\log n)$ time using $O(n/\log n)$ processors \cite{122} or in $O(\log \log n)$ time using $O(n)$ processors \cite{27}, and on the EREW PRAM in $O(\log n)$ time using $O(n/\log n)$ processors \cite{26, 79}.

3. \textit{Parallel List Ranking}: Given a linked list $L$ of $n$ nodes that are stored in an array $A$, such that for each node $a_i$ on $L$, $A(i)$ contains a pointer to the node following $a_i$ on $L$, the list ranking problem is to compute, for each node $a_i$ on $L$, the number of nodes that precede $a_i$ on $L$. This problem is easily solved in $O(n)$ time sequentially. Parallel list ranking can be done in $O(\log n)$ time and $O(n)$ EREW PRAM processors by using the "recursive doubling" technique \cite{133}. Known optimal parallel algorithms solve this problem in $O(\log n)$ time using $O(n/\log n)$ EREW PRAM processors \cite{46, 11}.

4. \textit{Parallel Tree Contraction}: Given a rooted tree whose nodes are stored in an array, reduce the tree to a single node by performing a sequence of node-removal operations. A node-removal operation is performed either at a leaf or at an internal node with only one child. The parallel version of this problem requires that the operations performed in the same parallel step be independent (i.e., if a node is being removed, its parent cannot be removed in the same step). This problem is solved easily in $O(n)$ sequential time. An efficient parallel algorithm for this important problem
was given in [102]. Several optimal parallel algorithms later solved this problem in $O(\log n)$ time using $O(n/\log n)$ EREW PRAM processors [1, 46, 63, 87].

5. **Brent's Theorem** [28]: Brent's theorem is a technique for reducing the number of processors required by a parallel algorithm for solving a problem. This technique is possibly applicable if a parallel algorithm designed for a problem has many processors idle for a long period of time. Brent's theorem is as follows.

**Theorem 1.1** (Brent [28]) Any synchronous parallel algorithm which takes a total of $T$ time and which performs a total of $W$ operations can be simulated by $P$ processors in time $O(W/P + T)$.

There are actually two qualifications to Brent's theorem before we can apply it to the PRAM: (i) at the beginning of the $i$-th parallel step of the original parallel algorithm, $i = 1, 2, \ldots, T$, we must be able to compute the amount of work $W_i$ done in that step, in time $O(W_i/P)$ and with $P$ processors, and (ii) we must know how to assign each processor to its task. Both qualifications (i) and (ii) to the theorem will be easily satisfied in our algorithms in this thesis.

6. **The Parallel Divide-and-Conquer**: The divide-and-conquer technique is a very common strategy for algorithm design. It is normally applied to the kind of problems that can be partitioned into a number of subproblems whose sizes are smaller than that of the original problem and whose structures are similar to that of the original problem, such that the solution to the original problem can somehow be obtained from the solutions to these subproblems. The basic parallel divide-and-conquer strategy consists of three stages: (i) partitioning the original problem into $k$ subproblems of nearly equal size each (the divide stage), (ii) solving each subproblem recursively in parallel (the recursion stage), and (iii) obtaining the solution to the original problem by somehow "merging" the output from the recursive calls to the $k$ subproblems (the conquer stage). The divide-and-conquer strategy is one of the most commonly-used techniques in sequential algorithm design, and is certainly one of the very important techniques for parallel computation.
1.3 Previous Work

Since the beginning of the last decade, parallel computational geometry has attracted a great deal of research interest. In this section, we give a brief survey of the previous work that is related to our results in this thesis. We review only the related deterministic PRAM algorithms. The problems that we mainly focus on are shortest paths and visibility problems. The comparisons between the related work and our results on specific problems will be made in the opening introductions of the subsequent chapters. In the rest of this section, the problem size is assumed to be \( n \) unless otherwise specified.

For a problem of computing shortest paths that avoid a set of geometric obstacles in the plane, parallel algorithms can often be obtained by first computing the visibility graph \( G(V, E) \), and then applying parallel graph algorithms for computing shortest paths on \( G(V, E) \), where \( V \) is the set of obstacle vertices and \( E \) contains every pair of obstacle vertices which are visible to each other. A parallel graph algorithm for computing shortest paths usually makes use of parallel matrix multiplications and involves a transitive closure computation (for example, see [82]). Note that multiplying two \( n \times n \) matrices in parallel is in general quite expensive; it uses \( n^M \) processors, where \( M \) reflects the method used to multiply two matrices in polylogarithmic time (currently, the smallest value for \( M \) is 2.376 [48]). The parallel transitive closure computation usually requires doing matrix multiplications a logarithmic number of times. Hence, such a parallel algorithm for computing geometric shortest paths is not efficient when compared to the time complexities of many sequential shortest paths algorithms in computational geometry.

So far, not many efficient parallel geometric shortest paths algorithms are available. There are several efficient parallel geometric shortest paths algorithms [29, 57, 72] that deal with paths inside a simple polygon; these algorithms are able to exploit certain geometric structures of a simple polygon. In [57], the shortest path between two points inside a polygon \( P \) is computed in \( O(\log n) \) time using \( O(n) \) CREW PRAM
processors, and the single-source shortest path tree from a vertex \( v \) to the other vertices of \( P \) is computed in \( O(\log^2 n) \) time using \( O(n) \) CREW PRAM processors. In [72], a simple polygon \( P \) is preprocessed in \( O(\log n) \) time using \( O(n) \) CREW PRAM processors, after which an implicit representation of the shortest path between any two query points inside \( P \) can be found in \( O(\log n) \) time using one processor. The problem of computing, in parallel, the single-source shortest path tree from a vertex is solved in \( O(\log n) \) time using \( O(n) \) CREW PRAM processors in [72]. In [29], a path between two points inside a simple polygon that consists of the minimum number of line segments is computed in \( O(\log n \log \log n) \) time using \( O(n) \) CREW PRAM processors.

There are two parallel algorithms for computing a rectilinear shortest path between two points that avoids disjoint rectangular obstacles in the plane [58, 75]. The algorithms in [58, 75] are both based on the parallel shortest paths algorithms for planar graphs and run in \( O(\log^3 n) \) time using \( O(n^{1.5}/\log^2 n) \) processors. But, it is not efficient to use these two algorithms to answer many queries about rectilinear shortest paths between arbitrary points in the plane.

For visibility problems, some work in the PRAM model has been done. Using the well-known “cascading divide-and-conquer” technique, Atallah et al. optimally solve in parallel the problem of computing the visible portions of non-intersecting line segments from a point in the plane [16]; their algorithm runs in \( O(\log n) \) time using \( O(n) \) EREW PRAM processors. Another optimal parallel algorithm for this problem is given in [23]; this algorithm runs in \( O(\log n) \) time using \( O(n) \) CREW PRAM processors, and is based on the many-way divide-and-conquer technique.

The kernel of a simple polygon \( P \) is the maximum subset of \( P \) such that the whole polygon \( P \) is visible from every point in that subset [95]. Cole and Goodrich [44] present an optimal CREW PRAM algorithm for computing the kernel of a simple polygon, in \( O(\log n) \) time using \( O(n/\log n) \) processors.

The problem of computing in parallel the weakly visible region inside a simple polygon \( P \) from an edge is solved efficiently by Goodrich et al. [72]; this algorithm
runs in $O(\log n)$ time using $O(n)$ CREW PRAM processors. Goodrich et al. [72] can also build, in $O(\log n)$ time using $O(n)$ CREW PRAM processors, a data structure for ray-shooting queries inside $P$ (given a point $p$ in $P$ and a direction $d$, a ray-shooting query is that of reporting, in $O(\log n)$ time using one processor, the first point on the boundary of $P$ that is hit by the ray starting at $p$ and going along direction $d$; see [32]). The visibility graph on the vertices of $P$ is obtained in $O(\log n)$ time using $O(n\log n + k/\log n)$ CREW PRAM processors [72], where $k$ is the number of edges in the visibility graph.

The problems of computing in parallel the complete visibility polygon and the weak visibility polygon inside a simple polygon $P$ from a convex subpolygon $C$ of $P$ are recently considered in [29] (a point $p$ in $P$ is completely visible from $C$ if $p$ is visible from every point in $C$, and is weakly visible from $C$ if $p$ is visible from at least one point in $C$). The algorithms in [29] for these two problems run in $O(\log n)$ time using $O(n)$ CREW PRAM processors, and make use of the close relation between shortest paths and visibility inside a simple polygon.

An important problem that we also consider in this thesis is that of computing in parallel the convex hull of a set of points in the plane (see [54, 114]). Chow [39] presents a parallel algorithm for this problem which runs in $O(\log^2 n)$ time using $O(n)$ CREW PRAM processors. Akl [6] shows that the problem of marking the vertices of the convex hull is solvable in $O(1)$ time using $O(n^3)$ CRCW PRAM processors. Optimal CREW PRAM algorithms for computing the convex hull of points in the plane are given in [2, 17, 18, 44]; these algorithms all run in $O(\log n)$ time using $O(n)$ processors. An optimal EREW PRAM algorithm for this problem is obtained by Miller and Stout [103]; their algorithm runs in $O(\log n)$ time using $O(n)$ processors. When the points are given sorted by the $x$-coordinates, the convex hull problem can be solved optimally on the CREW PRAM in $O(\log n)$ time using $O(n/\log n)$ processors [65, 127, 20], and on the CRCW PRAM in $O(\log \log n)$ time using $O(n/\log \log n)$ processors [20]. The problem of computing in parallel the convex hull of a simple
polygon is solved optimally in [127], in $O(\log n)$ time using $O(n/\log n)$ CREW PRAM processors.

Another important related problem is that of triangulating a polygon in parallel (see [54, 114]). This problem is solved in [2], in $O(\log^2 n)$ time using $O(n)$ CREW PRAM processors. The time complexity for this problem is reduced to $O(\log n)$ (still using $O(n)$ CREW PRAM processors) in [67, 135]. For a monotone polygon, the triangulation can be done optimally in $O(\log n)$ time using $O(n/\log n)$ CREW PRAM processors [20, 67, 86, 128]. For the problem of triangulating a set of points in the plane in parallel, optimal algorithms are available [101, 131]; these algorithms run in $O(\log n)$ time using $O(n)$ CREW PRAM processors. In fact, when the planar points are given sorted by the $x$-coordinates, the problem of triangulating the points can be solved by reducing it to that of triangulating $O(1)$ monotone polygons (and hence can be solved, by using the algorithms for the monotone case, in $O(\log n)$ time using $O(n/\log n)$ CREW PRAM processors).

1.4 Our Thesis

This thesis presents several new parallel techniques for solving shortest paths, visibility, and other related geometric problems. It is organized as follows.

Chapter 2 deals with the following shortest paths problem: Given a rectilinear convex polygon $P$ which has $O(n)$ vertices and which contains $n$ pairwise disjoint rectangular rectilinear obstacles, compute, in parallel, a data structure that supports queries about rectilinear shortest obstacle-avoiding paths in $P$. That is, a query specifies a source and a destination, and the data structure enables efficient processing of the query. We give a CREW PRAM algorithm for building the data structure in $O(\log^2 n)$ time. The number of processors used is $O(n^2/\log^2 n)$ if all queries are such that the source and the destination are both on the boundary of $P$, $O(n^2/\log n)$ if the source is an obstacle vertex and the destination is on the boundary of $P$, and $O(n^2)$ if both the source and destination are arbitrary points in $P$. The data structure we compute enables one processor to obtain the path length for any pair of query
vertices (of obstacles or of \(P\)) in constant time, or \(O([k/\log n])\) processors to retrieve the shortest path itself in logarithmic time, where \(k\) is the number of segments of that path. If the two query points are arbitrary rather than vertices, then one processor takes \(O(\log n)\) time (instead of constant time) for finding the path length, while the complexity bounds for reporting an actual shortest path remain unchanged. A number of other related shortest paths problems are solved. The techniques we use involve a fast computation of staircase separators, and a scheme for partitioning the boundaries of the obstacles in a way which ensures that the resulting path length matrices have a monotonicity property (such a property is apparently absent before applying our partitioning scheme). We also show that the data structure can be easily built sequentially in \(O(n^2)\) time, by using a different approach.

In Chapter 3, we consider the following visibility problem: computing the visible portions of an \(n\)-vertex simple polygonal chain from a point in the plane. In his book on art gallery problems and algorithms, O'Rourke argues that this problem is perhaps the most fundamental problem in visibility [110]. The parallel algorithm we design for this problem runs in \(O(\log n)\) time using \(O(n/\log n)\) processors in the EREW PRAM model, and hence is asymptotically optimal. The main difficulty in solving this problem is to detect quickly the intersections between visibility chains. We present new geometric insights that enable us to avoid the well-known linear work lower bound for detecting intersections between two general polygonal chains [31]. These geometric insights together with a combination of the many-way divide-and-conquer and the two-way divide-and-conquer strategies provide an approach to settling this important visibility problem optimally in parallel.

Chapter 4 further elaborates the parallel technique that is used in Chapter 3. This generalized technique enables us to solve optimally a number of important geometric problems in \(O(\log n)\) time using \(O(n/\log n)\) EREW PRAM processors, where \(n\) is the problem size. These problems include: computing the convex hull of a set of planar points sorted by the \(x\)-coordinates, computing the convex hull of a simple polygon, finding the kernel of a simple polygon, triangulating a set of planar points
sorted by the x-coordinates, triangulating monotone polygons and star-shaped polygons, solving the all dominating neighbors problem, etc. PRAM algorithms for these problems were previously known to be optimal (i.e., in $O(\log n)$ time and $O(n/\log n)$ processors) only on the CREW PRAM, which is a stronger model than the EREW PRAM.

The problem of detecting the weak visibility of an $n$-vertex simple polygon $P$ is that of finding whether $P$ is weakly visible from one of its edges and (if it is) identifying every edge from which $P$ is weakly visible. In Chapter 5, we present an optimal parallel algorithm for solving this problem. Our algorithm runs in $O(\log n)$ time using $O(n/\log n)$ processors in the CREW PRAM model. In order to solve this problem optimally in parallel, we reduce it to several subproblems; the parallel solutions to these subproblems are interesting in their own right. This algorithm enables us to solve optimally, in parallel, a number of problems on weakly visible polygons, such as computing shortest paths in a weakly visible polygon, triangulating a weakly visible polygon, checking the weak external visibility, solving the one-cruising-guard problem, etc. The previously known parallel solutions to these problems were not optimal.
2. RECTILINEAR SHORTEST PATHS WITH RECTANGULAR OBSTACLES

The problem of computing shortest paths that avoid obstacles is fundamental in computational geometry and has many applications. It has been studied in both sequential [37, 41, 50, 76, 77, 90, 91, 97, 96, 99, 105, 104, 107, 106, 108, 111, 119, 132, 134] and parallel [57, 58, 72, 75] settings, using various distance metrics. The rectilinear version of the problem, which assumes that each path's constituent segments are parallel to the coordinate axes, is motivated by applications in areas such as wire layout, circuit design, plant and facility layout, urban transportation, and robot motion. There are many efficient sequential algorithms that compute rectilinear shortest paths avoiding different classes of polygonal obstacle sets [41, 50, 90, 97, 105, 106, 132, 134]. In this chapter, we will present parallel techniques for solving several rectilinear shortest paths problems in the presence of rectangular obstacles in the plane. The parallel computational model we use in this chapter is the CREW PRAM.

Given a set of obstacles in the plane, a rectilinear shortest obstacle-avoiding path between two specified points is a rectilinear path connecting the two points such that the path does not intersect the interior of any obstacle, and the total length of the path is minimized. Figure 2.1 gives an example of a rectilinear shortest path between points \( v \) and \( w \) in the presence of rectangular obstacles in the plane.

This chapter considers the following problem. Let \( P \) be a rectilinear convex polygon having \( O(n) \) vertices and inside which lie \( n \) pairwise disjoint rectangular obstacles whose edges are parallel to the coordinate axes. We are interested in computing, in parallel, a data structure that supports queries about rectilinear shortest obstacle-avoiding paths in \( P \). That is, a query specifies a source and a destination, and the data structure enables efficient processing of the query.
We establish the following complexity bounds for solving this problem. We build the data structure in $O(\log^2 n)$ time, with $O(n^2/\log^2 n)$ processors if all queries are such that the source and the destination are both on the boundary of $P$, with $O(n^2/\log n)$ processors if the source is an obstacle vertex and the destination is on the boundary of $P$, and with $O(n^2)$ processors if both the source and destination are arbitrary points in $P$. The data structure we compute enables one processor to obtain the path length for any pair of query vertices (of obstacles or of $P$) in constant time, or $O([k/\log n])$ processors to retrieve the shortest path itself in logarithmic time, where $k$ is the number of segments of that path. If the two query points are arbitrary rather than vertices, then one processor takes $O(\log n)$ time (instead of constant time) for finding the path length, while the complexity bounds for reporting an actual shortest path remain unchanged. We also solve the case when $P$ is a convex $N$-gon with $n = o(N)$, in which case we are able to get an $O(N)$ rather than an $O(N^2)$ term in the work complexity by implicitly representing the $O(N^2)$ paths of interest, and the data structure for this implicit representation supports queries on
lengths and paths within the same time and processor bounds as the data structure for the explicit representation. A number of other related shortest paths problems are solved. We also show that the data structure can be easily built sequentially in $O(n^2)$ time, by using a different approach.

The techniques we develop involve a fast computation of staircase separators and a scheme for partitioning the obstacles of the boundaries in a way which ensures that the resulting path length matrices have a monotonicity property (such a property is apparently absent before the application of our partitioning scheme). These techniques could be useful for other related problems. The most general version of our algorithm uses a novel pipelining of the computation up and down the recursion tree, with $O(n)$ computational "flows" that originate from all the nodes and proceed only to the nodes whose associated problem size is larger than that of the flow's origin.

De Rezende et al. [50] gave a sequential algorithm for computing rectilinear shortest paths avoiding a set of $n$ rectangles between a fixed point $s$ (the source) and arbitrary destination points in the plane. That is, the algorithm in [50] solves the single source case of the shortest path problem. In $O(n \log n)$ time, this algorithm builds a data structure that can, in $O(\log n)$ time, answer a query that asks for the length of a rectilinear shortest path between the fixed source point $s$ and an arbitrary destination point $a$. The data structure also enables the reporting of an actual rectilinear shortest path between $s$ and $a$, in time proportional to the number of segments on the reported path. The method used in building the data structure of [50] is plane sweeping [114]. The queries we consider in this chapter are more general than the ones in [50], because the data structure we build is for all pairs shortest paths between arbitrary points in the plane. Our algorithm is not a parallelized version of the algorithm in [50], and it indeed takes a very different approach to solve the problem. Recently, Guha and Stout [75] and, independently, ElGindy and Mitra [58] have given an $O(\log^3 n)$ time and $O(n^{1.5}/\log^2 n)$ processor algorithm for the special case where both the source and destination are fixed. Note that answering our queries using this approach would be inefficient, both in terms of the time and of the processor complexity.
The rest of the chapter is organized as follows. Section 2.1 introduces some terminology and preliminary results. Section 2.2 gives one of the main ingredients we shall be using (the Staircase Separator Theorem). Section 2.3 proves some technical results that will be needed later in the "conquer" stages of our algorithms. Section 2.4 presents an algorithm which computes a data structure for an explicit representation for the lengths of the rectilinear shortest paths between the vertices of $P$ for the case $|P| = O(n)$. Section 2.5 generalizes our solution to paths between arbitrary pairs of points (Subsection 2.5.3 is the most difficult part of the chapter). Section 2.6 deals with the case $n = o(|P|)$. Section 2.7 extends the algorithms to computing the actual paths (rather than just their lengths). Section 2.8 sketches a sequential algorithm for building the data structure in $O(n^2)$ time.

Throughout this chapter, all geometric objects (segments, polygons, paths, rectangles, etc.) are implicitly assumed to be rectilinear; that is, each of their constituent segments is parallel to one of the two coordinate axes. In the rest of this chapter, all paths (shortest or otherwise) are assumed to be obstacle-avoiding.

### 2.1 Preliminaries

A rectilinear convex polygon is a rectilinear simple polygon such that every line segment which joins two points of the polygon and is parallel to a coordinate axis is contained in the polygon.

The input polygon $P$ is a convex polygon of $N$ vertices. We use $\text{Bound}(P)$ to denote the boundary of $P$. Polygon $P$ is specified by a circular sequence of vertices $v_1, v_2, \ldots, v_N$, as encountered by a counterclockwise walk along $\text{Bound}(P)$ starting at $v_1$. A circular ordering of the points on $\text{Bound}(P)$ is defined by the order in which they are encountered in the walk along $\text{Bound}(P)$ that follows the circular sequence of vertices of $P$. The boundary of $P$ is said to be clear since it does not intersect the interior of any obstacle.
The set of rectangular obstacles is denoted by $R$. $R$ is contained in $P$. The vertex set of $R$ is denoted by $V_R$ (hence $|V_R| = 4n$). We assume that $V_R$ has already been sorted in $O(\log n)$ time using $O(n)$ processors [43].

To avoid cluttering the exposition, in the rest of this chapter, we assume that no two distinct edges from $P$ or $R$ are collinear (the general case can be taken care of without much difficulty).

We use $x(p)$ and $y(p)$ to denote the two coordinates of a point $p$. In the $L_1$ metric, the distance between two points $p$ and $q$ is $d(p, q) = |x(p) - x(q)| + |y(p) - y(q)|$. A segment with endpoints $v$ and $w$ is denoted by $vw$ ($= wv$). The length of a path $C$ connecting two points is the sum of the lengths of its constituent segments. On the other hand, we use $|C|$ to denote the size of $C$, which is the number of segments of $C$ (not its length).

A path is said to be monotone with respect to the $x$-axis (resp., $y$-axis) iff its intersection with every vertical (resp., horizontal) line is a contiguous portion of that line. A path is convex if it is monotone with respect to both the $x$-axis and the $y$-axis. A convex path has the shape of a staircase, and in fact we shall henceforth use the word “staircase” as a shorthand for “convex path.” Note that a staircase from a point $p$ to a point $q$ is a shortest path between $p$ and $q$ since its length equals $d(p, q)$. Staircases can be increasing or decreasing, depending on whether they go up or down as we move along them from left to right. A staircase is unbounded if it starts and ends with a semi-infinite segment, i.e., a segment that extends to infinity on one side. A staircase is said to be clear if it does not intersect the interior of any obstacle.

A point $p$ is strictly below (resp., to the left of) a point $q$ iff $x(p) = x(q)$ and $y(p) < y(q)$ (resp., $y(p) = y(q)$ and $x(p) < x(q)$); we can equivalently say that $q$ is strictly above (resp., to the right of) $p$. A rectangle $r$ is below (resp., to the left of) an unbounded staircase $S$ if no point of $r$ is strictly above (resp., to the right of) a point of $S$; we can equivalently say that $S$ is above (resp., to the right of) $r$.

For a subset $R'$ of $R$, let $S$ be a decreasing unbounded staircase that is above all the rectangles in $R'$. Among all such staircases $S$, choose the lowest-leftmost one; that
is, if $S''$ is the chosen one, then there is no unbounded decreasing staircase $S'$ above $R'$ with a point of $S'$ strictly below or to the left of a point of $S''$. Denote such an $S''$ by $MAX_{NE}(R')$, where "N" is mnemonic for "North," and "E" is mnemonic for "East". Note that $MAX_{NE}(R')$ goes through all the maximal elements of $V_{R'}$ (see [114] for the definition of maximal elements of a point set). Using "S" and "W" as mnemonics respectively for "South" and "West," one can similarly define $MAX_{NW}(R')$, $MAX_{SE}(R')$, and $MAX_{SW}(R')$: $MAX_{NW}(R')$ is the lowest-rightmost increasing unbounded staircase above $R'$, $MAX_{SE}(R')$ is the highest-leftmost increasing unbounded staircase below $R'$, and $MAX_{SW}(R')$ is the highest-rightmost decreasing unbounded staircase below $R'$. See Figure 2.2.

The rectilinear convex hull of a set of objects in the plane, if it exists, is a (rectilinear) convex polygon that contains the set of objects and has minimum area [109]. In this chapter, all convex hulls are rectilinear.

Given a subset $R'$ of $R$, it is possible that the convex hull of $R'$ does not exist (see [109] for example). This can happen in exactly one of two ways (but not both): (i) $MAX_{NE}(R')$ and $MAX_{SW}(R')$ intersect, or (ii) $MAX_{NW}(R')$ and $MAX_{SE}(R')$ intersect. In case (i) (resp., (ii)) we define the convex connected region $Env(R')$ that contains
$R'$, called the envelope of $R'$, as follows: consider the disconnected convex region of the plane that is below $\text{MAX}_{\text{NE}}(R')$ and $\text{MAX}_{\text{NW}}(R')$, and above $\text{MAX}_{\text{SE}}(R')$ and $\text{MAX}_{\text{SW}}(R')$, and let $\text{Env}(R')$ be the union of that region with the finite segments of $\text{MAX}_{\text{NE}}(R')$ (resp., $\text{MAX}_{\text{NW}}(R')$). Figure 2.3 (a) illustrates case (i), and Figure 2.3 (b) illustrates case (ii). Although the definition of $\text{Env}(R')$ does not rule out that $\text{Env}(R')$ intersects the interior of an obstacle in $R - R'$, this will not happen because of the way $R'$ will be chosen ($\text{Env}(R')$ will not intersect the interior of any obstacle in $R - R'$). Note that if the convex hull of $R'$ exists then it coincides with $\text{Env}(R')$ (see Figure 2.3 (c)). It is trivial to construct $\text{Env}(R')$ in $O(\log |R'|)$ time using $O(|R'|/\log |R'|)$ processors when $V_{R'}$ is already sorted, by using parallel prefix [88, 89] and parallel merging [122].

Let $R'$ be a subset of $R$ such that $\text{Env}(R')$ does not intersect the interior of any obstacle in $R - R'$. We now extend the circular ordering on the points of $\text{Bound}(Q)$ we defined earlier (where $Q$ was a polygon) to the case when $Q = \text{Env}(R')$. We need to be able to say, for any three points $p$, $p'$, $p''$ on $\text{Bound}(Q)$ (cf. Figure 2.3 (a)), that (for example) $p'$ is between $p$ and $p''$ in the (extended) circular ordering (i.e., starting at $p$ and moving along the circular ordering we encounter $p'$ before $p''$). For each $X \in \{\text{NE}, \text{NW}, \text{SE}, \text{SW}\}$, we define $\text{MAX}_X(Q)$ similarly to the way we defined $\text{MAX}_X(R')$. Observe that there is an obvious total ordering that one can define for the points of $\text{MAX}_X(Q)$ that are on the boundary of $\text{Env}(R')$ (i.e., $\text{MAX}_X(Q) \cap \text{Bound}(Q)$). The circular ordering we seek can then be viewed as the concatenation of these four total orderings. The concatenation may result in some points (from $\text{MAX}_{\text{NE}}(R')$ in case (i), and from $\text{MAX}_{\text{NW}}(R')$ in case (ii)) appearing more than once in the ordering, and we duplicate those points and treat them as different points on $\text{Bound}(Q)$. More formally, the circular ordering is the circular version of the total order obtained as follows: start with the (totally ordered) points of $\text{Bound}(Q) \cap \text{MAX}_{\text{NE}}(Q)$, followed by those on $(\text{Bound}(Q) \cap \text{MAX}_{\text{NW}}(Q)) - \text{MAX}_{\text{NE}}(Q)$, followed by those on $(\text{Bound}(Q) \cap \text{MAX}_{\text{SE}}(Q)) - \text{MAX}_{\text{NW}}(Q)$, and followed by those on $(\text{Bound}(Q) \cap \text{MAX}_{\text{SW}}(Q)) - (\text{MAX}_{\text{SW}}(Q) \cup \text{MAX}_{\text{NE}}(Q))$. 
Figure 2.3 Illustrating $Env(R')$ and the circular ordering on $Bound(Env(R'))$. 
Figure 2.4 Illustrating $B(Q)$.

Let $Q$ be a convex connected region containing $R'$, for a subset $R'$ of $R$, such that $Q$ does not intersect the interior of any obstacle in $R - R'$ (hence $Bound(Q)$ is clear). In particular, $Q$ can be either $Env(R')$ or a convex polygon. In what follows, when we talk about "visibility", it is assumed that the obstacles as well as $Bound(Q)$ are opaque.

Definition 2.1 Let $B(Q)$ be the set of points $p$ on $Bound(Q)$ such that either (i) $p$ is a vertex of $Q$, or (ii) $p$ is horizontally or vertically visible from a vertex in $V_{R'}$ or from a vertex of $Q$ (see Figure 2.4).

That is, point $p \in Bound(Q)$ is in $B(Q)$ iff there is a vertex $v$ of $Q$ or of an obstacle contained in $Q$, such that segment $pv$ is horizontal or vertical, and the interior of $pv$ does not intersect $Bound(Q)$ or any obstacle. Obviously, $|B(Q)| = O(|Q| + |R'|)$. Using [16] and parallel merging [122], $B(Q)$ can be computed in $O(\log |Q| + \log |R'|)$ time and $O(|Q| + |R'| \log |R'|)$ work. We assume that $B(Q)$ is sorted according to the order in which its points are visited by a counterclockwise walk around $Bound(Q)$, starting at some vertex.

One result that we shall repeatedly make use of is Brent's theorem [28] (see Section 1.2, Chapter 1). Note that both qualifications to Brent's theorem will be easily satisfied in our algorithms.
Another result we shall be using deals with multiplying special kinds of matrices. All matrix multiplications are henceforth assumed to be in the \((\min, +)\) closed semi-ring, i.e., \((M' \ast M'')(i, j) = \min_k \{M'(i, k) + M''(k, j)\}\). If \(X, Y,\) and \(Z\) are finite sets of points in the plane, and if \(M_{XZ}\) (resp., \(M_{ZY}\)) denotes the matrix containing the lengths of the shortest paths from \(X\) to \(Z\) (resp., \(Z\) to \(Y\)), then it is not hard to see that the matrix \(M_{XZ} \ast M_{ZY}\) contains the lengths of the shortest \(X\)-to-\(Y\) paths that are \textit{constrained to go through} \(Z\) (i.e., they might not be best in absolute terms). Of course if for every path \(P\) from \(p \in X\) to \(q \in Y\) there exists a \(p\)-to-\(q\) path \(P'\) that goes through \(Z\) and is not longer than \(P\), then \((M_{XZ} \ast M_{ZY})(p, q)\) does contain the length of a shortest (unconstrained) \(p\)-to-\(q\) path.

A matrix \(M\) is said to be \textit{Monge} \([3]\) iff for any two successive rows \(i, i + 1\) and columns \(j, j + 1\) we have \(M(i, j) + M(i + 1, j + 1) \leq M(i, j + 1) + M(i + 1, j)\). Now, consider two finite point sets \(X\) and \(Y\), each totally ordered in some way (so we can talk about the predecessor and successor of a point in \(X\) or in \(Y\)), and such that the rows (resp., columns) of the path lengths matrix \(M_{XY}\) are as in the ordering for \(X\) (resp., \(Y\)). Matrix \(M_{XY}\) is \textit{Monge} iff for any two successive points \(p, p'\) in \(X\) and two successive points \(q, q'\) in \(Y\) we have \(M_{XY}(p, q) + M_{XY}(p', q') \leq M_{XY}(p, q') + M_{XY}(p', q)\). Figure 2.5 gives examples for \(M_{XY}\). Suppose that \(Q\) is a connected region whose
boundary is clear and that \(X\) and \(Y\) are two finite point sets that are on two disjoint portions of the boundary of \(Q\). In Figure 2.5 (a), \(Q\) is convex, and hence \(M_{XY}\) is Monge (assuming the points in \(X\) (resp., \(Y\)) are ordered as shown by the arrow). Figure 2.5 (b) shows an \(X\) and a \(Y\) for which \(M_{XY}\) is non-Monge (this figure also illustrates how length matrices that are non-Monge can arise in our problem). We shall later frequently make statements like "\(M_{XY}\) is Monge (or non-Monge)" without explicitly specifying what ordering we are assuming for the points in \(X\) and \(Y\), when such an ordering is obvious from the context; for example, if \(X\) and \(Y\) are each a contiguous subset of the vertices of a convex polygon \(Q\) and are on two disjoint portions of \(\text{Bound}(Q)\) (as in Figure 2.5 (a)), then the implicit ordering assumed for \(X\) and \(Y\) is the obvious one for which \(M_{XY}\) is Monge (\(X\) in clockwise order along \(Q\)'s boundary and \(Y\) in counterclockwise order, or \(X\) in counterclockwise order and \(Y\) in clockwise order). The following lemma summarizes these easy observations.

**Lemma 2.1** Let \(CR\) be a convex connected region whose boundary is clear. Let \(X\) and \(Y\) be finite sets of points on the boundary of \(CR\), such that the portion of that boundary spanned by \(X\) is disjoint from that spanned by \(Y\) (as in Figure 2.5 (a)). The matrix \(M_{XY}\) of path lengths between \(X\) and \(Y\) is Monge.

The next lemma is frequently used later.

**Lemma 2.2** Let \(X\) and \(Y\) be two finite point sets that belong to two unbounded staircases \(S_X\) and (respectively) \(S_Y\). Assume that \(S_X\) and \(S_Y\) are both clear. If \(X\) is completely on one side of \(S_Y\), and \(Y\) is completely on one side of \(S_X\), then \(M_{XY}\) is Monge.

**Proof.** It is easy to see that the lemma's hypotheses imply the existence of a convex connected region \(CR\) having the properties stated in Lemma 2.1.

The following lemma is well known [3, 12].

**Lemma 2.3** Assume \(M_{XZ}\) and \(M_{ZY}\) are Monge, with \(|X| = c_1|Z| \leq c_2|Y|\) for positive constants \(c_1\) and \(c_2\). Then \(M_{XZ} \ast M_{ZY}\), which is also Monge, can be computed in \(O(\log |Z|)\) time and \(O(|X||Y|)\) work on the CREW PRAM.
The next two lemmas are easy consequences of the previous one.

Lemma 2.4 Let $M_{XZ}$ and $M_{ZY}$ be Monge, where $|X| \leq \alpha$, $|Y| \leq \beta$, and $|Z| \leq \gamma$, such that $\alpha = c_1 \gamma \leq c_2 \beta$ for positive constants $c_1$ and $c_2$. Then $M_{XZ} \ast M_{ZY}$ (which is also Monge) can be computed in $O(\log \gamma)$ time and $O(\alpha \beta)$ work on the CREW PRAM.

Proof. "Pad" the matrices $M_{XZ}$ and $M_{ZY}$ with $+\infty$ entries so that they become $M'_{XZ}$ and $M'_{ZY}$, where $|X'| = \alpha$ and $|Y'| = \beta$. Apply Lemma 2.3 to multiply these padded matrices. The $M_{XZ} \ast M_{ZY}$ product is readily available from the $M'_{XZ} \ast M'_{ZY}$ product.

Lemma 2.5 Let $X$, $Y$, and $Z$ be finite point sets such that for any $p \in X$ and $q \in Y$, a shortest $p$-to-$q$ path can be chosen to go through $Z$, where $|X| \leq \alpha$, $|Y| \leq \beta$, and $|Z| \leq \gamma$, such that $\alpha = c_1 \gamma \leq c_2 \beta$ for positive constants $c_1$ and $c_2$. Assume that $X$ (resp., $Y, Z$) can be partitioned into a constant number of subsets $X_i, 1 \leq i \leq l_X$ (resp., $Y_j, Z_k, 1 \leq j \leq l_Y, 1 \leq k \leq l_Z$) such that every $M_{X_iZ_k}$ and $M_{Z_kY_j}$ is Monge. Given $M_{XZ}$ and $M_{ZY}$, the matrix $M_{XY}$ can be computed in $O(\log \gamma)$ time and $O(\alpha \beta)$ work on the CREW PRAM.

Proof. Trivial.

2.2 Computing a Staircase Separator

This section establishes the following theorem:

Theorem 2.1 (Staircase Separator) In $O(\log n)$ time and using $O(n)$ processors, it is possible to find an unbounded staircase, $Sep$, which partitions $R$ into two subsets $R_1$, $R_2$ such that the following properties hold:

1. $Sep$ does not intersect the interior of any obstacle in $R$.
2. Each of $R_1$ and $R_2$ contains no more than $7n/8$ rectangular obstacles.
3. $Sep$ consists of $O(n)$ segments.
Figure 2.6 Illustrating $NE(p)$ and $WS(p)$. 

Note: It is trivial to prove the existence of a $Sep^*$ for which $|R_1| = |R_2| = n/2$. The main contribution of this theorem is the parallel algorithm.

The rest of this section proves the Staircase Separator Theorem. We first introduce some terminology. For any point $p$, the NorthWest path of $p$ (denoted by the shorthand $NW(p)$) is the path to infinity obtained by starting at $p$ and going north until reaching an obstacle, at which point we go west along the obstacle’s boundary until we clear the obstacle and are able to resume our trip north. One can in this way define an $XY(p)$ path and a $YX(p)$ path for any combination of $X \in \{N, S\}$ and $Y \in \{E, W\}$. An $XY(p)$ path starts at $p$ and goes in the $X$ direction whenever it can, and uses a “go in the $Y$ direction” policy for getting around obstacles. A $YX(p)$ path is defined similarly. See Figure 2.6 for example.

To prove the theorem, it clearly suffices to find an unbounded staircase of size $O(n)$ that does not properly intersect any obstacle in $R$ (it may run along an obstacle’s boundary, however) and that has no less than $n/8$ obstacles on either side of it. The following lemma is one of the ingredients that will be used in computing such a staircase.
Lemma 2.6 Given a point $p$ not in the interior of any obstacle, an $XY(p)$ or a $YX(p)$ path can be computed in $O(\log n)$ time using $O(n)$ processors, where $X \in \{N, S\}$ and $Y \in \{E, W\}$.

Proof. Without loss of generality (WLOG), we just show how to compute $NW(p)$ (the other $XY(p)$ and $YX(p)$ paths can be obtained similarly). The ingredients we need for this computation are the parallel trapezoidal decomposition method [16] and the Euler Tour technique for tree computation [124]. Let the bottom edge of each obstacle have a “parent” pointer to the left edge of the obstacle. Using the algorithm in [16] we obtain, for the upper-left vertex $v$ of each obstacle, the trapezoidal segment above $v$ (the trapezoidal segment is thus above the left edge containing $v$). The trapezoidal segment for point $p$ is easy to find. These trapezoidal segments are the bottom edges of obstacles. (In the case where a trapezoidal segment does not exist, we assume that it is the “segment at infinity”.) Then let $p$ and the left edges of the obstacles each have a “parent” pointer to their respective trapezoidal segments. In this way, we create a forest whose nodes are left edges and bottom edges of obstacles, and point $p$. The roots of the trees in the forest are the nodes whose trapezoidal segment is at infinity. Using the Euler Tour technique for tree computation [124], we find the path from $p$ to the root of the tree to which $p$ belongs. The path so found is $NW(p)$.

The algorithm for computing the desired staircase separator $\mathcal{S}ep$ is as follows: we first find a vertical line $V$ such that there are as many vertices of $R$ to its left as to its right. Let $v$ be the number of obstacles in $R$ that are properly intersected by $V$. If $v \geq n/4$ then we are essentially done: we find a point $p$ on $V$ such that half of the obstacles properly intersected by $V$ are above it, and half of them below it. Assume that $p$ is not in any obstacle (the algorithm can be easily modified for the case when $p$ lies inside an obstacle). Then we take $\mathcal{S}ep$ to be the union of $NE(p)$ and $SW(p)$. So suppose, in what follows, that $v < n/4$. Find a horizontal line $H$ such that there are as many vertices of $R$ above it as below it. Let $h$ be the number of obstacles in $R$ properly intersected by $H$. If $h \geq n/4$ then we are done for the same reason as in
the case where \( v \geq n/4 \). So suppose, in what follows, that \( h < n/4 \). Let \( p \) be the intersection of \( V \) and \( H \), and assume that \( p \) is not in any obstacle (the algorithm can be easily modified for the case when \( p \) lies inside an obstacle).

Lines \( V \) and \( H \) together partition the plane into four quadrants which we call \( NE \) (NorthEast), \( NW \), \( SE \), and \( SW \). Let \( R_{NW} \) be the subset of \( R \) that lies only in the \( NW \) quadrant (hence no obstacle in \( R_{NW} \) properly intersects either \( V \) or \( H \)). Let \( R_{NE}, R_{SE}, \) and \( R_{SW} \) be defined analogously. Note that

\[
|R_{NE}| + |R_{NW}| + |R_{SE}| + |R_{SW}| = n - v - h.
\]

WLOG, assume that

\[
|R_{NW}| = \max\{ |R_{NE}|, |R_{NW}|, |R_{SE}|, |R_{SW}| \}.
\]

We now show that \( Sep \) can be taken to be the union of \( NE(p) \) and \( WS(p) \). Since such a \( Sep \) is obviously a staircase that consists of no more than \( 2n + 2 \) segments, does not properly intersect any obstacle, and separates \( R \) into two subsets, it suffices to prove that there are (i) at least \( n/8 \) obstacles above \( Sep \) and (ii) at least \( n/8 \) obstacles below \( Sep \). Now, (i) is trivially true because, since each of \( h \) and \( v \) is less than \( n/4 \),
we must have $|R_{NE}| + |R_{NW}| + |R_{SE}| + |R_{SW}| > n/2$, which implies $|R_{NW}| > n/8$. The proof of (ii) requires some work. Suppose to the contrary that there are fewer than $n/8$ obstacles below $Sep$. The staircase $Sep$ partitions $R_{NE}$ into two subsets: call them $R'_{NE}$ and $R''_{NE}$ (see Figure 2.7). Similarly, $Sep$ partitions $R_{SW}$ into two subsets: call them $R'_{SW}$ and $R''_{SW}$ (see Figure 2.7). WLOG, assume that $|R'_{NE}| \geq |R'_{SW}|$ (the other case is symmetrical). We obtain a contradiction to the definition of $H$, as follows. The number of vertices of $R$ above $H$ is $\geq 4|R_{NW}| + 2h + 4|R'_{NE}| + 4|R''_{NE}| \geq 4|R_{NW}| + 2h + 4|R'_{NE}|$. The number of vertices of $R$ below $H$ is $\leq 4(n/8) + 2h + 4|R'_{SW}|$ (where we used the assumption that there are fewer than $n/8$ obstacles below $Sep$ and the fact that the number of obstacles that are simultaneously below both $Sep$ and $H$ is no more than the number of obstacles that are below $Sep$). Now, let us compare $4|R_{NW}| + 2h + 4|R'_{NE}|$ (which is less than or equal to the number of vertices of $R$ above $H$) with $4(n/8) + 2h + 4|R'_{SW}|$ (which is strictly larger than the number of vertices of $R$ below $H$). Since $|R_{NW}| > n/8$ and $|R'_{NE}| \geq |R'_{SW}|$, we have

$$4|R_{NW}| + 2h + 4|R'_{NE}| > 4(n/8) + 2h + 4|R'_{SW}|.$$  

It follows that the number of vertices of $R$ below $H$ is smaller than the number of vertices of $R$ above $H$. This contradicts the definition of $H$, and completes the proof of the Staircase Separator Theorem.

2.3 Other Building Blocks

This section introduces further technical results that will later be used. In what follows, $Q$ is a convex connected region containing a subset $R'$ of $R$ such that either (i) $Q$ is a convex polygon with $O(|R'|)$ vertices, or (ii) $Q = Env(R')$. The lemmas in this section assume that $Q$ does not intersect the interior of any obstacle in $R - R'$. Note that the boundary of $Q$ is clear. For such a $Q$, we define arrays $Horiz$ and $Vert$ (of size $|B(Q)|$ each) as follows. Let $p, q$ be a pair of adjacent points in $B(Q)$; that is, $pq$ is on $Bound(Q)$ and $p, q$ are the only points of $B(Q)$ that are on $pq$. Then $Horiz(pq)$ (resp., $Vert(pq)$) is the portion of $Bound(Q) - pq$ that is horizontally
Figure 2.8 Illustrating array Vert.

Figure 2.8 Illustrating array Vert.

(resp., vertically) visible from \( pq \); that is, either \( \text{Horiz}(pq) \) (resp., \( \text{Vert}(pq) \)) is empty, or for each point \( a \in \text{Horiz}(pq) \) (resp., \( a \in \text{Vert}(pq) \)) there is a point \( b \in pq \) such that \( a \) is horizontally (resp., vertically) visible from \( b \). In Figure 2.8, \( \text{Vert}(pq) = p'q' \), and \( \text{Vert}(q'r) \) is empty. The procedures that later use these lemmas will always make sure that the \( \text{Horiz} \) and \( \text{Vert} \) arrays are available (it is in fact quite easy to compute these arrays, by using parallel prefix [88, 89]).

When computing the shortest paths between pairs of vertices of \( Q \), we shall also concern ourselves with the nonvertex points in \( B(Q) \). The reason we do this is that (as will become apparent later) it is easier to solve the more general problem of computing the \( B(Q) \)-to-\( B(Q) \) paths.

Notation 2.1 We use \( D_Q \) to denote the \( |B(Q)| \times |B(Q)| \) matrix containing the lengths of shortest paths between all the pairs of points in \( B(Q) \).

Lemma 2.7 Given the matrix \( D_Q \) and arrays \( \text{Horiz} \) and \( \text{Vert} \), the length of a shortest path between any pair of points on \( \text{Bound}(Q) \) can be found in \( O(\log |B(Q)|) \) time using one processor.

Proof. Let \( b_1 \) and \( b_2 \) be two points on \( \text{Bound}(Q) \). Let \( v \) (resp., \( w \)) be the first point of \( B(Q) \) encountered by a clockwise (resp., counterclockwise) walk from \( b_1 \) along
Bound(Q). If \( b_1 \in B(Q) \), then \( b_1 = v = w \). Let points \( v' \) and \( w' \) be similarly defined for \( b_2 \). WLOG, assume that both \( b_1 \) and \( b_2 \) are not in \( B(Q) \). The \( O(\log |B(Q)|) \) time is needed only for finding \( \overline{vw} \) and \( \overline{v'w'} \). If \( \overline{vw} \) is contained in \( \text{Horiz}(\overline{v'w'}) \) or in \( \text{Vert}(\overline{v'w'}) \), or if \( \overline{v'w'} \) is contained in \( \text{Horiz}(\overline{vw}) \) or in \( \text{Vert}(\overline{vw}) \), then the \( b_1 \)-to-\( b_2 \) path length is simply \( d(b_1, b_2) \). Otherwise the path length we seek is one of the following four quantities: (i) \( d(b_1, v) + D_Q(v, v') + d(v', b_2) \), (ii) \( d(b_1, v) + D_Q(v, w') + d(w', b_2) \), (iii) \( d(b_1, w) + D_Q(w, v') + d(v', b_2) \), and (iv) \( d(b_1, w) + D_Q(w, w') + d(w', b_2) \). This can be proved by contradiction: assuming that none of (i)-(iv) is the length we seek leads to a contradiction with the definition of one of \( \{v, w\} \) or \( \{v', w'\} \).

To avoid introducing new notation, we shall from now on use \( \text{Env}(X) \) even when \( X \) consists of arbitrary objects (not just rectangular obstacles). The definition we gave earlier for the case \( X = R \) extends to other objects in a natural way. In particular, \( X \) can now be a collection of polygons, staircases, etc.

Lemma 2.8 Let \( C \) be a bounded staircase originating on \( \text{Bound}(Q) \) such that (i) \( C \) does not intersect \( Q \) except at one of its endpoints, (ii) \( C \) is a contiguous portion of the boundary of \( Q' = \text{Env}(Q \cup C) \), and (iii) \( Q' \) intersects the interior of an obstacle only if the obstacle is contained in \( Q \). Let \( C' \) (resp., \( B' \) ) be \( B(Q') \cap C \) (resp., \( B(Q') \cap \text{Bound}(Q) \) ). Then given the matrix \( D_Q \), we can obtain the matrix of the \( B' \)-to-\( C' \) path lengths in \( O(\log m) \) time and \( O(m^2) \) work, where \( m = |C| + |B(Q)| \).

Proof. WLOG, we assume that \( C \) starts at the highest edge of \( Q \) and is decreasing (Figure 2.9). Let \( \text{Cross} \) be the set of points on \( \text{Bound}(Q) - \text{Bound}(Q') \) that either are in \( B(Q) \) or are horizontal or vertical projections of the vertices of \( C \). We partition \( \text{Cross} \) into two subsets: \( \text{Cross}_1 \) which contains those points of \( \text{Cross} \) on \( \text{MAXNE}(Q) \), and \( \text{Cross}_2 = \text{Cross} - \text{Cross}_1 \) (see Figure 2.9). The matrix \( M \) of the \( B' \)-to-\( \text{Cross} \) path lengths can be obtained from \( D_Q \) within the desired complexity bounds, by using Lemma 2.7, and similarly for the matrix \( M' \) of the \( \text{Cross}_2 \)-to-\( \text{Cross}_1 \) path lengths. The matrix \( M_1 \) of the \( \text{Cross}_1 \)-to-\( \text{C'} \) path lengths is trivially available (each \( v \)-to-\( w \) path length in it is simply \( d(v, w) \)). The lengths of shortest paths between \( \text{Cross}_2 \) and the portion of \( C' \) that is above \( \text{Cross}_1 \) can be obtained by multiplying \( M' \) with
Figure 2.9 Illustrating Lemma 2.8.

$M_1$; since both $M'$ and $M_1$ are Monge (by Lemma 2.1), they can be multiplied within the desired complexity bounds (by using Lemma 2.5). The lengths of shortest paths between $Cross_2$ and the portion of $C'$ that is not above $Cross_1$ are trivial to obtain (they are described by the function $d(\cdot, \cdot)$). Hence we now have the matrix $M^*$ of the lengths of the $Cross$-to-$C'$ paths. To obtain the lengths of the $B'$-to-$C'$ paths, we use Lemma 2.5 on length matrices $M$ and $M^*$, with $B'$ playing the role of $X$, $C'$ playing the role of $Y$, and $Cross$ playing the role of $Z$. \[\square\]

Lemma 2.9 Let Sep' be the staircase obtained by applying the Staircase Separator Theorem (Theorem 2.1) to $R'$, and let $R_1'$ and $R_2'$ be the two subsets of $R'$ on either side of Sep'. Then both $Bound(Env(R_1'))$ and $Bound(Env(R_2'))$ are clear.

Proof. This follows from the facts that Sep' is a staircase that does not properly intersect the obstacles in $R'$, that $Env(R_1')$ and $Env(R_2')$ are both contained in $Q$, and that $Q$ does not intersect the interior of any obstacle in $R - R'$.

Lemma 2.10 Let points $q_1$ and $q_2$ belong to $Q$ and let $P$ be a path between $q_1$ and $q_2$. Then there exists a path $P'$ between $q_1$ and $q_2$ which does not go outside $Q$ and is not longer than $P$.

Proof. Since $Q$ is a convex connected region whose boundary is clear, any portion of $P$ that goes outside $Q$ can be replaced by going along the boundary of $Q$. The length
of the path $P'$ obtained from the replacement is not longer than that of $P$ because of the convexity of $Q$.

Lemma 2.11 If a shortest path between points $p$ and $q$ intersects a clear staircase $S'$, then there exists a shortest path between $p$ and $q$ whose intersection with $S'$ is one connected component.

Proof. This is an immediate consequence of the fact that for any two points $s_1$ and $s_2$ of $S'$, a shortest path between them is the path along $S'$.

2.4 Computing the Lengths Matrix $D_P$ When $|P| = O(|R|)$

Recall that the input polygon $P$ is convex and contains all the obstacles in $R$, and that $D_P$ is the matrix of the $B(P)$-to-$B(P)$ shortest path lengths. In this section, we assume that $|P| = N \leq c|R|$ for some positive constant $c$, and we only concern ourselves with computing $D_P$. It suffices to give an algorithm for the case where the input consists of only $R$ and where we wish to compute the lengths of paths between pairs of points in $B(Q)$ with $Q = Env(R)$. This is enough because if the input includes both $P$ and $R$, then we first compute $D_Q$ and then easily obtain $D_P$ from it with a constant number of applications of Lemma 2.8.

The algorithm takes as input the set $R$ of $n$ rectangular obstacles, and computes the $|B(Q)| \times |B(Q)|$ matrix $D_Q$, where $Q = Env(R)$. It does so by first finding a staircase separator $Sep$ that partitions $R$ into two subsets $R_1$ and $R_2$. Then it recursively solves, in parallel, the subproblems for $R_1$ and $R_2$, respectively, obtaining two matrices $D_{Q_1}$ and $D_{Q_2}$, where $Q_1 = Env(R_1)$ and $Q_2 = Env(R_2)$. Finally it obtains matrix $D_Q$ from matrices $D_{Q_1}$ and $D_{Q_2}$.

We use the Staircase Separator Theorem (Theorem 2.1) to find $Sep$. Computing $Q_1$ and $Q_2$ is trivial. Because of Lemma 2.9 and Lemma 2.10, the two matrices returned by the two recursive calls contain, respectively, the lengths of the $B(Q_1)$-to-$B(Q_1)$ paths and the $B(Q_2)$-to-$B(Q_2)$ paths (i.e., they are indeed $D_{Q_1}$ and $D_{Q_2}$). Thus the main difficulty is how to obtain efficiently $D_Q$ from $D_{Q_1}$ and $D_{Q_2}$.
Figure 2.10 Illustrating the proof of Theorem 2.2.

Let \( T(n) \) and \( W(n) \) respectively denote the time and work complexities of the algorithm. Then to show that \( T(n) = O(\log^2 n) \) and \( W(n) = O(n^2) \), it suffices to prove Theorem 2.2 below. This would be enough because we would then have:

\[
T(n) \leq T(7n/8) + c_1 (\log n)
\]
\[
W(n) \leq W(|R_1|) + W(|R_2|) + c_2 (n^2)
\]

with the boundary conditions \( T(1) = c_3 \) and \( W(1) = c_4 \), where the \( c_i \)'s are positive constants, \(|R_1| + |R_2| = n, n/8 \leq |R_1|, |R_2| \leq 7n/8\). Brent's theorem [28] would then imply a processor complexity of \( O(n^2 / \log^2 n) \).

Theorem 2.2 The matrix \( D_Q \) can be computed from \( D_{Q_1} \) and \( D_{Q_2} \) in \( O(\log n) \) time and \( O(n^2) \) work.

Proof. Let \( Q_{left} \) (resp., \( Q_{right} \)) be the portion of \( Q \) on the left (resp., right) side of \( Sep \) (see Figure 2.10). (Note that \( Q_{left} \) and \( Q_{right} \) both include the portion of \( Sep \) that is in \( Q \).) Since \( Q_1 \) is contained in \( Q_{left} \) and the matrix \( D_{Q_1} \) is known, we can apply Lemma 2.7 and Lemma 2.8 a constant number of times to obtain the matrix \( D_{Q_{left}} \).

The matrix \( D_{Q_{right}} \) is obtained similarly. Let \( Left \) (resp., \( Right \)) be the subset of \( B(Q) \) that is in \( Q_{left} \) (resp., \( Q_{right} \)), and let \( Middle \) be the subset of \( B(Q_{left}) \cup B(Q_{right}) \) that
lies on \(Sep\). From matrix \(D_{Q_{left}}\) (resp., \(D_{Q_{right}}\)), using Lemma 2.7, we can obtain the matrix \(M_{left}\) (resp., \(M_{right}\)) of the lengths of shortest paths between \(Left\) (resp., \(Right\)) and \(Middle\). Lemma 2.11 implies that the problem of computing \(D_Q\) is essentially that of multiplying \(M_{left}\) and \(M_{right}\). By Lemma 2.1, these two matrices are Monge. Hence by using Lemma 2.5, these two matrices can be multiplied within the desired bounds. The correctness of the computation of \(D_Q\) easily follows from the fact that for any points \(p, q\), where \(p \in Left\) and \(q \in Right\), there exists a \(p\)-to-\(q\) shortest path that goes through a point in \(Middle\).

\[\square\]

2.5 Path Lengths between Arbitrary Points

We extend the techniques of the previous sections to computing the lengths of shortest paths between arbitrary query points. The query time is logarithmic using one processor. We first consider the structure for the \(B(P)\)-to-\(V_R\) paths and construct it using an \(O(\log^2 n)\) time algorithm with \(O(n^2 / \log n)\) processors. We then consider the structure for the \(V_R\)-to-\(V_R\) paths and construct it using an \(O(\log^2 n)\) time algorithm with \(O(n^2)\) processors. Finally, we show that even with arbitrary query points we can use essentially the same structure as in the \(V_R\)-to-\(V_R\) case. The first subsection gives some observations that are crucial in all the above cases.

2.5.1 Some Useful Observations

Let \(T\) be the recursion tree for the algorithm in Section 2.4; that is, the root of \(T\) corresponds to the “top-level” recursive call (the one associated with \(R\)), the children of the root correspond to the recursive calls for \(R_1\) and \(R_2\), and so on. It is easy to modify that algorithm so that the information (path length matrices, separators, etc.) produced by each recursive call remains stored in \(T\) even after that call returns. We assume that this modification has already been done, so that each node \(v\) of \(T\) stores the obstacle set \(R_v \subseteq R\) associated with \(v\), as well as \(Q_v = Env(R_v)\), the staircase \(Sep_v\) partitioning \(R_v\) (WLOG, assume \(Sep_v\) is increasing), and the following matrices in addition to matrix \(D_{Q_v}\). Let \(Left_{R_v}\) (resp., \(Right_{R_v}\)) be the
The storage space taken by $T$ and all the information associated with its nodes obeys the same recurrence as for the work complexity, and hence is $O(n^2)$.

For convenience, we now introduce a notation $\text{Chain}(\cdot)$ such that, if $X$ is a finite set of points that were obtained from some contiguous portion of a staircase, then $\text{Chain}(X)$ is that contiguous portion of the staircase; usually the context
makes it clear which contiguous portion of the staircase is meant—we shall typically use \( \text{Chain}(X) \) for \( X \in \{U_v, U'_v, W_v, W'_v\} \). For example, \( \text{Chain}(U_v) = \text{Left-Sep}_v \), and \( \text{Chain}(U'_v) = \) the portion of \( \text{Bound}(\text{Env}(\text{Left}R_v)) \) that is in the interior of \( Q_v \) and is not on \( \text{Left-Sep}_v \). Observe that staircases \( \text{Chain}(U_v) \) and \( \text{Chain}(W_v) \) both divide \( Q_v \) into two halves, each of which is a convex connected region, whereas staircases \( \text{Chain}(U'_v) \) and \( \text{Chain}(W'_v) \) respectively cut \( \text{Env}(\text{Left}R_v \cup \text{Left-Sep}_v) \) and \( \text{Env}(\text{Right}R_v \cup \text{Right-Sep}_v) \) into two halves, each of which is also a convex connected region.

Each obstacle vertex \( p \in V_R \) occurs on at least one of the \( U_v, U'_v, W_v, W'_v \) lists, for some \( v \in T \). Therefore to compute the \( V_R \)-to-\( B(P) \) path lengths, it suffices to compute, for all \( v \in T \) and \( X \in \{U, U', W, W'\} \), the \( X_v \)-to-\( B(P) \) path lengths. The reader may wonder why we have partitioned the points in \( B(\text{Env}(\text{Left}R_v)) - \text{Bound}(Q_v) \) into two subsets \( U_v \) and \( U'_v \): the reason is that it will enable the use of Lemma 2.5, by making the path length matrices Monge, something which would not have been true otherwise (this will become clearer in the proofs of the lemmas below).

We henceforth assume that a pre-processing stage has explicitly computed, for each \( p \in V_R \), the eight paths \( X(p) \) for all \( X \in \{NE, NW, SE, SW, EN, ES, WN, WS\} \) (the definitions of these paths were given in Section 2.2; see Figure 2.6 for example). This is done by first computing the forest that implicitly describes all the \( \text{NE}(p) \)’s (call it the “\( \text{NE forest} \)”) in \( O(\log n) \) time with \( O(n) \) processors, as in the proof of Lemma 2.6. Then we extract from that \( \text{NE} \) forest an explicit description of \( \text{NE}(p) \), for each \( p \in V_R \). This extraction is easily done in \( O(\log n) \) time and \( O(n^2) \) work, by making a copy of the tree that contains \( p \) for each \( p \in V_R \) and obtaining \( \text{NE}(p) \) from that copy using standard parallel tree computation methods [124]. Given points \( p \) and \( q \), where \( p \in V_R \) and \( q \) is arbitrary, determining whether \( \text{NE}(p) \) goes above or below \( q \) can be done in logarithmic time using one processor (by a binary search on \( \text{NE}(p) \)). The same holds for the other 7 forests that describe the other 7 kinds of paths. We can speak of the segments associated with a forest (say, the \( \text{NE} \) forest): these are the segments that lie on \( \text{NE}(p) \) for some \( p \in V_R \). There are clearly \( O(n) \) such
segments associated with each of the 8 forests. In fact, all the chains associated with the recursion tree’s nodes (i.e., the chains for \{U_v, U'_v, W_v, W'_v\}) use only segments associated with the eight forests. We pre-process the segments associated with these 8 forests in the following way: for each such forest (say, the NE one), we compute an *indicator* matrix \(I_{NE}\) of size \(O(n) \times O(n)\) which is defined as follows. For each \(p \in V_R\) and each segment \(s\) associated with the 8 forests, \(I_{NE}(p, s) = s'\), where \(s'\) is the segment of \(NE(p)\) that intersects the infinite line \(l_s\) containing \(s\). These eight indicator matrices are easily computed in \(O(\log n)\) time and using a quadratic amount of work.

It is easily seen that these indicator matrices enable us to determine, for any point \(p \in V_R\) and any staircase \(C\) which uses only segments associated with the 8 forests, whether, for example, \(NE(p)\) intersects \(C\), and to find a point on that intersection, in \(O(\log |C|)\) time and \(O(|C|)\) work. This last observation is used implicitly in the proof of Lemma 2.14. The next two lemmas are also needed for proving Lemma 2.14.

**Definition 2.2** Two staircases \(\mathcal{P}\) and \(\mathcal{P}'\) are said to **cross once** iff (i) their intersection is not empty, (ii) each staircase has at least one point that is strictly to the left of the other staircase and one point that is strictly to its right, and (iii) for either staircase, the portion of that staircase that is on or to the left (resp., right) of the other staircase consists of one connected component. We adopt the convention that the **crossing point** between two such staircases is one that belongs to their intersection and partitions them into pieces that do not satisfy (ii) (if many such points can be so chosen, we choose the one with, say, the smallest \(x\) coordinate).

Intuitively, “crossing once” means switching from being strictly on one side of the other staircase to being strictly on the other side of it, exactly one time. For example, two unbounded increasing staircases \(\mathcal{P}\) and \(\mathcal{P}'\) such that no point of \(\mathcal{P}\) is strictly above \(\mathcal{P}'\) cannot be said to cross once *even if their intersection is non-empty*.

**Lemma 2.12** Let \(R'\) be a subset of \(R\) such that \(Env(R')\) does not intersect the interior of any obstacle in \(R - R'\). Let \(C\) be a staircase on the boundary of \(Env(R')\). For any
$p \in V_R$ and any $X \in \{NE, NW, SE, SW, EN, ES, WN, WS\}$, $X(p)$ crosses $C$ at most once.

Proof. Note that $\text{Bound}(\text{Env}(R'))$ is clear. If one of $C$ and $X(p)$ is increasing and the other decreasing, then the lemma trivially holds. So suppose that both $C$ and $X(p)$ are increasing (the proof is similar if they are both decreasing). To prove that $C$ and $X(p)$ cross at most once, first observe that one of the two classes of segments of $X(p)$ (horizontal or vertical) consists of segments that coincide with obstacle boundaries. WLOG, assume the horizontal segments of $X(p)$ all coincide with obstacle boundaries. In order for $C$ and $X(p)$ to cross more than once, at least one vertical segment of $C$ would have to properly intersect one of the horizontal obstacle edges along which runs one of $X(p)$'s horizontal segments. This would imply that $C$ penetrates the interior of an obstacle, contradicting the hypothesis that $C$ is clear.

Lemma 2.13 Let $v$ be a node of $T$ and $X$ be any of $\{U, U', W, W'\}$. For a point $p \in X_v$ and a point $q$ not in the interior of $Q_v$, there exists a shortest $p$-to-$q$ path that goes through a point of $B(Q_v)$.

Proof. Let $\mathcal{P}$ be a shortest $p$-to-$q$ path. Since $q$ is not in the interior of $Q_v$, $\mathcal{P}$ must intersect $\text{Bound}(Q_v)$ before reaching $p$. By Lemma 2.10, $\mathcal{P}$ can be chosen so that it enters $Q_v$ only once, say, $\mathcal{P}$ intersects $\text{Bound}(Q_v)$ in between two adjacent points $b_1, b_2 \in B(Q_v)$. (Note that $\overline{b_1b_2}$ is on $\text{Bound}(Q_v)$ and no other point of $B(Q_v)$ is on $\overline{b_1b_2}$.) WLOG, assume $\overline{b_1b_2}$ is vertical and the interior of $Q_v$ is to its left. Imagine shooting leftward horizontal rays from all the points of $\overline{b_1b_2}$, and let $\text{Region}$ be the region illuminated by these rays, assuming that obstacles as well as $\text{Bound}(Q_v)$ are opaque. Point $p$ cannot lie in the interior of $\text{Region}$, since otherwise $b_1$ and $b_2$ would not be adjacent in $B(Q_v)$ and would be separated in $B(Q_v)$ by the horizontal projection of $p$ on $\overline{b_1b_2}$. This means that $\mathcal{P}$ has to intersect one of the two rays from $b_1$ and (respectively) $b_2$, and hence can be deformed so that it goes through either $b_1$ (if it intersects the ray of $b_1$) or $b_2$. \qed
Lemma 2.14 Let $X$ and $Y$ be any of $\{U, U', W, W'\}$. Let $v$ and $w$ be two nodes of $T$ such that $|R_v| \leq c|R_w|$ for some positive constant $c$ and $\text{Chain}(Y_w)$ does not intersect the interior of $Q_v$. If, in addition to the information stored in $T$, we are given the lengths of the $Y_w$-to-$B(Q_v)$ paths, then we can compute, in $O(\log(|R_v|))$ time and $O(|R_v||R_w|)$ work, the lengths matrix of the shortest $X_v$-to-$Y_w$ paths.

Proof. We begin with the case $X_v = U_v$ or $W_v$; WLOG, assume $X_v = U_v$. Note that $\text{Chain}(X_v)$ partitions $Q_v$ into two halves such that each half of $Q_v$ is convex and connected.

Let $p, p'$ be the endpoints of $\text{Chain}(X_v)$, and $q, q'$ be the endpoints of $\text{Chain}(Y_w)$. WLOG, assume that $\text{Chain}(Y_w)$ is increasing, that $q'$ is the lower-left endpoint of $\text{Chain}(Y_w)$, and that $q$ is the upper-right endpoint of $\text{Chain}(Y_w)$. Now, augment $\text{Chain}(Y_w)$ by adding to it $\text{NE}(q)$ and $\text{SW}(q')$, thus obtaining an unbounded staircase $\text{Chain}'(Y_w)$. We distinguish two cases, depending on whether $\text{Chain}'(Y_w)$ intersects the interior of $Q_v$ or not. Testing whether such an intersection occurs is easy to do, by using the indicator matrices.

The first case, when $\text{Chain}'(Y_w)$ does not intersect the interior of $Q_v$, is handled as follows. WLOG, assume that $Q_v$ is below $\text{Chain}'(Y_w)$. Let $l, r, t,$ and $b$ be respectively a leftmost, rightmost, top, and bottom vertex of $Q_v$ (there are at most two candidates for each, and we choose one of these two arbitrarily). The idea is to use Lemma 2.5, with $B(Q_v)$ playing the role of $Z$ in that lemma, $X_v$ playing the role of $X$ in that lemma, and $Y_w$ playing the role of $Y$ in that lemma. (Note that by Lemma 2.13, the $X_v$-to-$Y_w$ paths can be chosen to go through $B(Q_v)$.) But in order to be able to use that lemma, we need to judiciously partition each of $B(Q_v)$ and $Y_w$ into a constant number of pieces ($X_v$ will not need to be partitioned). The partitioning of $B(Q_v)$ is quite simple: the points determining the partition are $l, r, t, b, p,$ and $p'$ (see Figure 2.12); hence $B(Q_v)$ gets partitioned into at most six pieces—fewer if the six points determining the partition are not distinct. Note that the path lengths matrix between $X_v$ and any of these six pieces is Monge (by Lemma 2.1), thus satisfying one of the requirements for Lemma 2.5. To satisfy the other requirement, however, we must
Figure 2.12 Illustrating the proof of Lemma 2.14.
partition $Y_w$ with great care, in such a way that the path lengths matrix between each piece of $Y_w$ and each piece of $B(Q_v)$ is indeed Monge. This partitioning of $Y_w$ is induced by a partitioning of $\text{Chain}(Y_w)$ into at most seven pieces, according to the following (at most six) points: the points at which $\text{Chain}(Y_w)$ crosses each of $\text{NE}(r)$, $\text{NE}(t)$, $\text{NW}(t)$, $\text{NW}(l)$, $\text{SW}(l)$, and $\text{SW}(b)$ (see Figure 2.12). (Note that $\text{Chain}(Y_w)$ can cross each of $\text{NE}(r)$, $\text{NE}(t)$, $\text{NW}(t)$, $\text{NW}(l)$, $\text{SW}(l)$, and $\text{SW}(b)$ at most once, by Lemma 2.12.) Finding these six points is easy to do by using the indicator matrices. It is not hard to see that this is a suitable partition of $Y_w$, by Lemma 2.2.

The second case, when $\text{Chain}'(Y_w)$ intersects the interior of $Q_v$, is handled as follows. By Lemma 2.12, $\text{Chain}'(Y_w)$ can cross $\text{Chain}(X_v)$ at most once and $\text{Bound}(Q_v)$ at most twice. The crossing point between $\text{Chain}(X_v)$ and $\text{Chain}'(Y_w)$ (if one exists), as well as the (at most) two crossing points of $\text{Chain}'(Y_w)$ with the boundary of $Q_v$, can easily be computed by using the indicator matrices. $\text{Chain}'(Y_w)$ defines two independent subproblems, one on each side of it; they are independent because of Lemma 2.10. We solve each of these two subproblems separately, similarly to the way we solved the first case.

We now turn our attention to the case $X_v = U_v'$ or $W_v'$; WLOG, assume $X_v = U_v'$. Suppose that we have computed the lengths of the $U_v$-to-$Y_w$ paths using the algorithm in the previous paragraphs (hence the lengths of the $Y_w$-to-$B(\text{Env}($Left$R_v \cup $Left$-$Sep_v$))$ paths are known). Then essentially the same algorithm as for the case $X_v = U_v$ works except that $\text{Env}(\text{Left}R_v \cup \text{Left} -$Sep_v$)$ now plays the role of $Q_v$ and $U_v'$ plays the role of $U_v$ ($Y_w$ being the same).

Lemma 2.15 Let $w$ be an ancestor of $v$ in $T$. Let $X$ be any of $\{U, U', W, W'\}$. If, in addition to the information stored in $T$, we are given the lengths of the $B(Q_v)$-to-$B(Q_w)$ paths, then we can compute, in $O(\log(|R_v|))$ time and $O(|R_v||R_w|)$ work, the lengths matrix of the shortest $X_v$-to-$B(Q_w)$ paths.

Proof. If $w = v$, then the computation is trivial. Otherwise, $Q_w$ properly contains $Q_v$ (see Figure 2.13). Hence $\text{Bound}(Q_w)$ does not intersect the interior of $Q_v$. Partition
Figure 2.13 Illustrating Lemma 2.15.

Bound(Q_w) into four staircases, in the obvious way, and for each such staircase C use the same proof as in Lemma 2.14, with B(Q_w) \cap C playing the role of Y_w. \hfill \Box

Lemma 2.16 For each v in T and all X, Y \in \{U, U', W, W'\}, the lengths matrix of the X_v-to-Y_v paths can be computed in O(\log |R_v|) time and O(|R_v|^2) work (see Figure 2.11).

Proof. Similar to that of Lemma 2.14 and omitted. \hfill \Box

The observations presented in this subsection will be used in what follows.

2.5.2 The B(P)-to-V_R Path Lengths

We begin with the case P = Env(R). First, we build the recursion tree T and all its associated information, as explained in the previous subsection. Let root be the root of T (hence Q_root = Env(R)). We would like to compute, for each node v \in T, the four matrices containing the X_v-to-B(Q_root) path lengths, for each X \in \{U, U', W, W'\}. We do this from the root down, one level at a time. At root, we use Lemma 2.15 to do this in O(|R_root|^2) work (the condition for the lemma is trivially satisfied there, since we are using it with root = v = w). Having done this for root makes the application of Lemma 2.15 at each child v of root possible (with w = root), which takes O(|R_root||R_v|) work for each such v. This in turn makes
the application of the lemma at each grandchild \( v \) of the root possible, etc. We proceed in this way from the root down, one level at a time, until we reach the leaf level. Let the height of \( T \) be \( \text{height}(T) \). The time for this is clearly \( O(\log |R_{\text{root}}| \times \text{height}(T)) \) and the work is \( O(|R_{\text{root}}| \sum_{v \in T} |R_v|) \). This implies an \( O(\log^2 n) \) time and \( O(n^2 \log n) \) work complexities (where the fact that \( \sum_{v \in T} |R_v| = O(n \log n) \) was used). By Brent’s theorem, the processor complexity is \( O(n^2 / \log n) \). The case where \( P \) properly contains \( \text{Env}(R) \) is easily handled by the method for the above case, in conjunction with that of Section 2.4.

2.5.3 The \( V_R \)-to-\( V_R \) Path Lengths

First we do the following pre-processing. In parallel for each \( w \in T \), we compute the lengths of the \( X_v \)-to-\( B(Q_w) \) paths and the \( X_v \)-to-\( Y_w \) paths for all descendants \( v \) of \( w \), and all \( X, Y \in \{U, U', W, W'\} \). These two computations are trivial to do if \( v = w \) (in the first case the information is already stored in \( T \), in the second case we can use Lemma 2.16). So suppose \( v \neq w \), i.e., \( v \) is a proper descendant of \( w \). Then the computation of the \( X_v \)-to-\( B(Q_w) \) path lengths is done exactly as in the previous subsection (with \( w \) now playing the role of \( \text{root} \)), resulting in \( O(\log^2 n) \) time and \( O(|R_w|^2 \log |R_w|) \) work for this particular \( w \). This also gives us some but not all the desired \( X_v \)-to-\( Y_w \) path lengths; for example, if \( u \) is the child of \( w \) whose \( Q_u \) contains \( X_v \), and if \( U'_w \) is on \( \text{Bound}(Q_u) \), then we already know the \( X_v \)-to-\( U'_w \) path lengths but not the \( X_v \)-to-\( W'_w \) path lengths—these must still be computed. We compute the remaining \( X_v \)-to-\( Y_w \) path lengths also in a top-down manner, in parallel for all \( w \), from \( w \) down, by using repeatedly Lemma 2.14 at each level of the downward trip from \( w \); the lemma’s hypothesis is satisfied, i.e., we do know the \( Y_w \)-to-\( B(Q_v) \) path lengths, because they would already have been computed earlier by \( w \)’s top-down computation. This too takes \( O(\log^2 n) \) time and \( O(|R_w|^2 \log |R_w|) \) work. Summed over all such \( w \), the total work for the pre-processing is \( O(\log n \sum_{w \in T} |R_w|^2) = O(n^2 \log n) \).

Since we already computed, in the previous subsection, the lengths of the paths having an endpoint in \( B(\text{Env}(R)) \), it suffices to compute the lengths of paths having
both endpoints in $V_R - B(Env(R))$. Each vertex in $V_R - B(Env(R))$ appears on some $X_v, v \in T, X \in \{U, U', W, W'\}$. Therefore it suffices to compute the lengths of the $X_v$-to-$Y_w$ paths for all $v, w \in T$ and $X, Y \in \{U, U', W, W'\}$. This is done in the rest of this subsection.

Before going into the details, we point out the main reason behind the elaborate constructions that are about to follow: unless great caution is exercised, when computing the $X_v$-to-$Y_w$ path lengths for a particular $v, w$ pair, the associated Monge matrix multiplication might not satisfy the size requirements of Lemma 2.5; that is, the required relations between $\alpha, \beta, \gamma$ of that lemma might be violated. This is the main reason for the condition "$|R_v| \leq |R_w|$" that is about to play such an important role in the concept of "flow" that is given next.

For nodes $v, w \in T$, let the tree distance between $v$ and $w$, denoted by $l(v, w)$, be the number of edges on the $v$-to-$w$ path in the undirected version of $T$. Clearly, $l(v, v) = 0$. The computation for the $V_R$-to-$V_R$ path lengths proceeds in $2 \times \text{height}(T)$ stages, each of which takes $O(\log n)$ time. Whereas the approach in the previous subsection was a "top-down flow" from the root of $T$, repeatedly making use of Lemma 2.15, here the flow is from each $v$ to the $w$'s that have $|R_w| \geq |R_v|$, in the order of their tree distance from $v$. The flows for all $v$'s start at the same time. Thus, if $|R_v| \leq |R_w|$, then the flow for $v$ reaches $w$ at stage $l(v, w)$ (which is at most $2 \times \text{height}(T)$). When the flow for $v$ reaches $w$, it computes the desired information between $v$ and $w$, possibly using Lemma 2.5 and Lemma 2.14 (this information consists of more than the $X_v$-to-$Y_w$ path lengths—more on this later). Observe that for any pair $v, w \in T$, the flow of one of these two nodes eventually reaches the other, so that all the $X_v$-to-$Y_w$ path lengths eventually get computed. In what follows, $X, Y \in \{U, U', W, W'\}$.

Before describing the detailed computation done when the flow for $v$ reaches $w$, let us look at the subset of $T$ visited by the flow for $v$ (call it $\text{Region}(v)$). The flow for $v$ obviously does not visit the proper subtree of $v$ in $T$, and it obviously does visit every $w$ on the $v$-to-root path in $T$. For every such $w$, it may also visit a portion of the subtree of the child of $w$ (call it $u$) which is not an ancestor of $v$; the portion
so visited induces a subtree of $T$ rooted at $u$. If $v'$ is the parent of $v$ then clearly $\text{Region}(v') \subseteq \text{Region}(v)$ and, if the flow for $v'$ reaches $w$ at (say) stage $k$, then the flow for $v$ will reach that same $w$ at stage $k+1$.

When the flow for $v$ enters $w$, $w \neq v$, we obtain the $X_v$-to-$Y_w$ path lengths. These path lengths are available from the pre-processing stage if $w$ is an ancestor of $v$, but otherwise they must be computed—we compute them using Lemma 2.14. The details of this computation are tricky. When $v$'s flow enters $w$ from $w$'s parent, it can do so under one of two possible modes of operation (call them mode 1 and mode 2): mode 1 when $|R_{\text{parent}(v)}| \leq |R_w|$, and mode 2 when $|R_w| < |R_{\text{parent}(v)}|$. Observe that, as a result of these definitions of modes 1 and 2, we have the following:

- If the flow for $v$ is at $w$ in a mode, then at the next stage the flows of $v$'s children will enter $w$ in mode 1.
- If the flow for $v$ is at $w$ in mode 1, then at the next stage it can go to a child of $w$ in mode 1 or mode 2.
- If the flow for $v$ is at $w$ in mode 2, then at the next stage it can go to a child of $w$ in mode 2 only.
- If the flow for $v$ is at $w$ in mode 2, then $|R_w| = O(|R_v|)$ and, furthermore, that flow will finish visiting $w$'s subtree in $O(1)$ stages.

Obviously, if at stage $k$, the flow for $v$ is simultaneously at $w$ and $w'$, then its mode at $w$ might be different from its mode at $w'$.

In order to compute the desired $X_v$-to-$Y_w$ path lengths, the flow for $v$ gets help from a piece of preparatory information that enables it to use Lemma 2.14; this preparatory information consists of either (i) the $B(Q_v)$-to-$Y_w$ path lengths (if $v$'s flow enters $w$ in mode 1), or (ii) the $X_v$-to-$B(Q_w)$ path lengths (if $v$'s flow enters $w$ in mode 2). In case (i), this preparatory information is either obtained from $\text{parent}(v)$ (if $v$'s flow enters $w$ in mode 1), or is available from the pre-processing (if $v$'s flow enters $w$ from a child of $w$). In case (ii), the preparatory information comes from
\(v\) itself (it would have obtained that information at the previous stage). Of course, the assumption that the preparatory information is already available to \(v\) as its flow enters \(w\) places an extra burden on \(v\): that of computing the preparatory information that it will be required to supply at the next stage; it will supply the information to each of its children \(u'\) (because the flow for \(u'\) may enter \(w\) in mode 1 at the next stage), or it will supply the information to itself (if its own flow will enter a child of \(w\) in mode 2 at the next stage). Below we give the details of the computations performed in each of these two modes.

In what follows, suppose the flow for \(v\) has just entered \(w\), at stage \(k = l(v, w)\). We must prove that we can compute the \(X_v\)-to-\(Y_w\) path lengths and that we can compute the preparatory information to help perform the next stage \(k + 1\). The proof is by induction on \(k\), the basis \((k = 1)\) being straightforward (since \(w = \text{parent}(v)\) in that case, and hence all the needed information is trivially available). The details for the induction step follow. We distinguish two cases, based on the mode in which \(v\)’s flow has entered \(w\).

**Mode 1.** \(|R_{\text{parent}(v)}| \leq |R_w|\): then it must have been the case where, at stage \(k - 1\), the flow for \(\text{parent}(v)\) had already reached \(w\) and (by the induction hypothesis) had computed (for its children’s future benefit) the \(B(Q_v)\)-to-\(Y_w\) path lengths information. It should be clear that this information (available after stage \(k - 1\) at \(\text{parent}(v)\)) enables us to use Lemma 2.14 for computing the \(X_v\)-to-\(Y_w\) path lengths (see Figure 2.14 (a)), in \(O(\log |R_v|)\) time and \(O(|R_v||R_w|)\) work.

Now \(v\) must compute, for the benefit of each of its own children, say \(u'\), the preparatory information that \(u'\) will need at the next stage \(k + 1\), namely, the \(B(Q_{u'})\)-to-\(Y_w\) path lengths information (note that the flow for \(u'\) will enter \(w\) in mode 1). But this information is readily available, from the knowledge of the \(B(Q_v)\)-to-\(Y_w\) and the \(X_v\)-to-\(Y_w\) path lengths information.

Finally, \(v\) checks whether its flow will next enter a child \(u\) of \(w\) in mode 2 and, if so, it collects the preparatory information that it will then need at the next stage \(k + 1\), namely, the \(B(Q_u)\)-to-\(X_v\) path lengths. We say “collect” rather than compute,
Figure 2.14 Illustrating the computation of (a) mode 1, and (b) mode 2.
because this information is already available, by the following argument. WLOG, assume \( Q_u = \text{Env}(\text{Left}R_w) \). The portion of \( B(Q_u) \) that is interior to \( Q_w \) consists of \( U'_w \) and a portion of \( U_w \), and the path lengths between these and \( X_v \) have just been computed. We claim that the path lengths between \( X_v \) and \( B' = B(Q_u) - U'_w - U_w \) had been computed earlier. To see this, first observe that every point \( p \in B' \) is either (i) in \( B(Q_{\text{lca}(u,v)}) \) where \( \text{lca}(u,v) \) is the lowest common ancestor of \( u \) and \( v \) in \( T \), or (ii) in \( Y_z \) for some \( z \) on the \( w \)-to-\( \text{lca}(u,v) \) path in \( T \). In case (i) we already know the \( p \)-to-\( X_v \) path lengths because of the pre-processing. In case (ii), we also know the \( X_v \)-to-\( Y_z \) path lengths information, because the flow for \( v \) has already reached \( w \), and hence had earlier reached \( z \).

Mode 2. \(|R_w| < |R_{\text{parent}(v)}|\): we claim that \( v \) already knows the \( X_v \)-to-\( B(Q_w) \) path lengths information. To see this, first observe that, if \( v \) and \( w \) are siblings, then that information is already available from the pre-processing. If on the other hand \( v \) and \( w \) are not siblings, then it must be the case where \( v \)'s flow entered \( \text{parent}(w) \) at the previous stage \( k - 1 \): by the induction hypothesis it must have prepared that information for its own use at stage \( k \). The availability of this information implies that we can use Lemma 2.14 to compute the \( X_v \)-to-\( Y_w \) path lengths information, where our \( v \) (resp., \( w \)) plays the role of the lemma's \( w \) (resp., \( v \)) (see Figure 2.14 (b)). Note that as a by-product of this computation, we now know the \( X_v \)-to-\( B(Q_u) \) path lengths information for each child \( u \) of \( w \), and this is precisely the preparatory information that may be needed by \( v \)'s flow for the next stage, in case \( v \)'s flow enters \( u \) (as already noted, it would do so in mode 2).

We now claim that we can also easily collect, for every child \( u' \) of \( v \), the \( B(Q_{u'}) \)-to-\( Y_w \) path lengths, which is precisely the preparatory information that is needed by the flow of \( u' \) for the next stage, when that flow enters \( w \) in mode 1. To prove the claim, assume WLOG that \( Q_{u'} = \text{Env}(\text{Left}R_{u'}) \). The portion of \( B(Q_{u'}) \) that is interior to \( Q_v \) consists of \( U'_{u'} \) and a portion of \( U_v \), and the path lengths between these and \( Y_w \) have just been computed. We claim that the path lengths between \( Y_w \) and \( B' = B(Q_{u'}) - U'_{u'} - U_v \) had been computed earlier. To see this, first observe that every point \( p \)
$\in B'$ is either (i) in $B(Q_{lca(v,w)})$ where $lca(v,w)$ is the lowest common ancestor of $v$ and $w$ in $T$, or (ii) in $X_z$ for some $z$ on the $parent(v)$-to-$lca(v,w)$ path in $T$. In case (i) we already know the $p$-to-$Y_w$ path lengths because of the pre-processing. In case (ii), we also know the $Y_w$-to-$X_z$ path lengths information, because the flow for $w$ has already reached $parent(v)$ and hence had earlier reached $z$.

To analyze the work complexity of the above scheme, observe that the work done, when $w$ is visited by the flow for $v$, is $O(|R_v||R_w|)$. Hence the total work is $O(\sum_{v \in T} \sum_{w \in T} |R_v||R_w|) = O(\sum_{v \in T} |R_v|(n \log n)) = O(n^2 \log^2 n)$ (where we made use of the fact that $\sum_{w \in T} |R_w| = O(n \log n)$).

Of course, we can collect the lengths of the paths between the points in $V_R \cup B(P)$, which we just computed, into a single $O(n) \times O(n)$ lengths matrix.

### 2.5.4 Path Lengths with Arbitrary Query Points

We point out that, given the lengths matrix computed for the case of the $V_R$-to-$V_R$ paths, we can augment this structure with two planar subdivisions so that we are able to handle a path length query between two arbitrary endpoints in $O(\log n)$ time using one processor. We begin with the case of queries with only one arbitrary endpoint, the other endpoint being in $V_R$, and then we later extend it to the case of two arbitrary endpoints.

Recall that one of the by-products of the previous $V_R$-to-$V_R$ length matrix computation is the $X(p)$ paths for all $p \in V_R$ and all $X = NE, NW, \ldots, etc$. Given such an $X(p)$ path for a $p \in V_R$, we can use one processor to do a logarithmic time binary search on the path. However, we shall need to do binary search on such paths originating from an arbitrary point $p$ (not in $V_R$). For such a $p$, the (e.g.) $NE(p)$ path is not explicitly available, but it could easily be obtained if we knew which obstacle is first encountered by an upward ray-shooting from $p$. We can easily perform such a ray-shooting query in logarithmic time and one processor, provided we do the following pre-processing. The horizontal (resp., vertical) trapezoidal edges of $V_R$, together with the obstacles' boundaries, define an $O(n)$-vertex planar subdivision $H_1$ (resp.,
We pre-process $H_1$ (resp., $H_2$) as in [16], in $O(\log n)$ time and $O(n)$ processors, so that it can support a point location query in $O(\log n)$ time with one processor. This enables one processor to determine, in $O(\log n)$ time, which obstacle is first encountered by a horizontal (resp., vertical) ray-shooting from an arbitrary query point $p$ by using $H_1$ (resp., $H_2$).

Assume the path length query is between points $p$ and $q$ where, WLOG, $x(q) \leq x(p)$ and $y(q) \leq y(p)$. If $p$ is arbitrary and $q \in V_R$, then we first check whether $p$ lies above or below $NE(q)$; assume it lies below (the other case is symmetrical). We then perform a leftward ray-shooting query from $p$. If the ray intersects $NE(q)$ before it hits an obstacle, then we are done because the path length from $p$ to $q$ is simply $d(p, q)$ (since there is a $q$-to-$p$ staircase). Otherwise let $e = q_1q_2$ be the (vertical) obstacle edge encountered by the ray-shooting. The length of a shortest $q$-to-$p$ path is the smaller of the following: (i) $d(p, q_1) +$ the $q_1$-to-$q$ path length, and (ii) $d(p, q_2) +$ the $q_2$-to-$q$ path length (recall that the $q_1$-to-$q$ and $q_2$-to-$q$ path lengths are readily available, since $q, q_1, q_2 \in V_R$). That the length we seek is the smaller of (i) or (ii) is easy to establish and was in fact proved in [50].

If both $p$ and $q$ are arbitrary, then we first obtain $NE(q)$ in $O(\log n)$ time using one processor, by doing an upward ray-shooting from $q$, etc. We then proceed exactly as in the previous case, except that we need to use the method of the previous paragraph to compute the lengths of the shortest $q_1$-to-$q$ and $q_2$-to-$q$ paths.

2.6 Path Lengths When $|P| \gg |R|$ 

In this section we consider the case when the polygon $P$ containing the $n$ obstacles has many more vertices than $n$, that is, $|P| = N \gg |R| = n$. So suppose that $|R| = o(|P|)$. We can avoid a term quadratic in $N$ in the work complexity by building a data structure for an implicit representation of the path lengths. The method we show here works for any of the versions of the problem we considered earlier, and results in $O(\log N + \log^2 n)$ time and $O(N + n^2 f(n))$ work complexities where $f(n) = 1$ in the $B(P)$-to-$B(P)$ case, and $f(n) = \log n$ in the $B(P)$-to-$V_R$ case. This implicit
representation allows us to still use one processor to achieve constant time for a length query whose endpoints are in $B(P) \cup V_R$.

The idea is to partition $Bound(P)$ into eight chunks, each of which is a contiguous portion of $Bound(P)$. Each of the eight chunks has associated with it an $O(n)$-vertex unbounded staircase which separates that chunk from the interior of $Env(R)$, and that is used to answer queries relevant to that chunk. Since each such staircase has $O(n)$ vertices, we can use the algorithms of the previous sections to process it, that is, to compute length information about paths that have an endpoint on that staircase.

The way we partition $Bound(P)$ is by drawing an infinite horizontal (resp., vertical) line from each of the highest and lowest (resp., leftmost and rightmost) edges of $Env(R)$. These four lines induce a partition of $Bound(P)$ into at most eight connected components, each of which is one of the above-mentioned chunks. We call these the top, north-east, etc. chunks (in clockwise order), respectively (see Figure 2.15). It is easy to find, for each point in $B(P)$, to which chunk it belongs. We explain how to process the top chunk and the north-east one, since the others are obviously analogous. We only consider the shortest paths that are nontrivial in the sense that
they link two endpoints that are on segments that do not horizontally or vertically “see each other.” The trivial shortest paths are easily handled as explained earlier in Section 2.3, specifically, in Lemma 2.7.

For the top chunk, we let $K$ be the set of vertical projections of the points of $B(Env(R))$ on the horizontal line $H$ defining that chunk. It is obvious that for any vertex $p$ of $P$ in the top chunk, a nontrivial shortest path from $p$ to anywhere below $H$ can be “deformed”, without any increase in its length, so that it goes through a point of $K$, and hence the lengths of paths to the points in $K$ implicitly represent the lengths of all the paths to the top chunk.

For the north-east chunk, we project horizontally as well as vertically on that chunk the points of $B(Env(R))$; let $K$ be the set of these $O(n)$ projection points. Let $C$ be $\text{MAX}_\text{NE}(K)$. We must prove that any nontrivial path from a vertex $p$ of $P$ on the north-east chunk which crosses $C$ can be deformed, without any increase in its length, so that it goes through a vertex of $C$. Let $p$ be any vertex on the north-east chunk, and let $q$ (resp., $q'$) be the point of $K$ that is immediately after (resp., before) $p$ in the linear ordering of that chunk’s points. Note that $q$ and $q'$ are not adjacent vertices on $C$, since there is a vertex $q''$ of $C$ between them (by definition of the $\text{MAX}_\text{NE}(K)$). Now, consider any nontrivial path to $p$. Since there is no point of $B(Env(R))$ whose horizontal or vertical projection on the north-east chunk falls in between $q$ and $q'$ in $K$, it follows that any such path must go below one of $\{q, q'\}$, in which case we can deform it to go through one of $\{q, q'\}$ (say, $q$) or through $q''$. Hence the lengths of paths to the vertices of $C$ implicitly represent the lengths of all the paths to the north-east chunk.

To achieve constant query time, we must have associated, in a pre-processing stage, each such $p$ with $q$ and $q'$, something that is easily done by a parallel merging [122] and a parallel prefix [88, 89].
2.7 Computing the Actual Paths

In this section we present a parallel algorithm for building a data structure that enables us to report an actual shortest path (rather than just its length) between the query points, within $O(\log n)$ time and $O(\log n + k)$ work, where $k$ is the number of segments on that path. Assuming that the structure for querying path lengths is available (computed as in Section 2.5), the algorithm builds the data structure for the actual path queries in an additional $O(\log n)$ time and $O(n^2)$ work. We use the same terminology as in Section 2.5.

The data structure for the path queries consists of: (i) $|V_R|$ shortest path trees, each of them rooted at one of the vertices in $V_R$, (ii) the two planar subdivisions $H_1$ and $H_2$ of Subsection 2.5.4, and (iii) the $X(v)$ paths for each $v \in V_R$ and $X = NE, NW, \ldots$, etc.

We already discussed the computation of the $X(v)$ paths (in Subsection 2.5.1), and that of the two planar subdivisions $H_1$ and $H_2$ (Subsection 2.5.4). Hence we need only show how to compute a shortest path tree for every vertex in $V_R$, and how to use these shortest path trees to process a path query in parallel.

The shortest path trees are computed using the following information: (1) the $V_R$-to-$V_R$ lengths matrix, containing the lengths of paths between the vertices in $V_R$ (computed in Subsection 2.5.3), (2) the two planar subdivisions $H_1$ and $H_2$, (3) the $X(v)$ paths for each vertex $v \in V_R$, (4) two copies of $V_R$, one sorted by the $x$-coordinates and the other by the $y$-coordinates, (5) for every $w \in V_R$, the obstacle (if there exists one) that is hit by a horizontal leftward (resp., rightward) ray-shooting from $w$, and the obstacle (if there exists one) that is hit by a vertical upward (resp., downward) ray-shooting from $w$ (note that using $H_1$ and $H_2$, all these obstacles for a vertex $w$ can be found in $O(\log n)$ time and one processor), and (6) for each edge $e$ of the obstacles, the set of the vertices in $V_R$ whose ray-shootings hit $e$, denoted as $Hit(e)$, sorted according to where their rays hit $e$ (for example, if $e$ is the right edge of an obstacle, then $Hit(e)$ is the set of vertices in $V_R$ whose horizontal leftward
ray-shootings hit e, and Hit(e) is sorted by the y-coordinates) (note that all the Hit(e) sets can be obtained in \(O(\log n)\) time and \(O(n \log n)\) work).

We now show how to use the above information in (1)-(6) to compute, in an additional \(O(\log n)\) time and linear work, a shortest path tree rooted at a vertex \(v \in V_R\). For every \(w \in V_R - \{v\}\), we associate a “parent” pointer with \(w\) as follows. WLOG, assume that \(w \in V_R - \{v\}\) such that \(x(v) \leq x(w), y(v) \leq y(w)\), and \(w\) is below \(NE(v)\); note that in this case, the shortest path between \(v\) and \(w\) is monotone with respect to the \(x\)-axis (see [50] for a proof). If the horizontal leftward ray-shooting from \(w\) crosses \(NE(v)\) before reaching an obstacle, then a shortest path from \(w\) to \(v\) is via \(NE(v)\); we then let \(w\) have an associated pointer to the segment on \(NE(v)\) at which the ray from \(w\) crosses \(NE(v)\). If the ray from \(w\) does not cross \(NE(v)\), then let \(u_1\) and \(u_2\) be the two vertices of the right edge of the obstacle hit by the ray; using the \(V_{wto-V_R}\) lengths matrix, we can easily decide whether a shortest path from \(w\) to \(v\) is via \(u_1\) or via \(u_2\) (say it is via \(u_1\)), and we then let \(w\) have an associated pointer to \(u_1\). Also we let the segments of each \(X(v)\) path be directed toward \(v\).

This computation for vertex \(v\) results in a directed graph of \(O(n)\) edges and vertices, whose vertices are the union of the vertices in \(V_R\) and the vertices of the \(X(v)\) paths. This graph is a tree rooted at \(v\) because every vertex in the graph except \(v\) has exactly one out-going edge (the pointer to its parent) and no cycle can occur in this directed graph because of the monotonicity property of the shortest paths [50] (recall that this monotonicity states that the only shortest paths we need to consider are those that are monotone with respect to one of the two coordinate axes). Therefore, we have obtained a shortest path tree rooted at \(v\).

It follows that the computation of all the \(O(n)\) shortest path trees whose roots are the vertices in \(V_R\) can be done in an additional \(O(\log n)\) time and \(O(n^2)\) work.

Next we discuss how to pre-process the shortest path trees, so that each tree can support a shortest path query between the vertex of \(V_R\) stored in the root of the tree and any vertex in \(V_R\). We restrict our attention to the case where both query points are vertices in \(V_R\), because the case of arbitrary query points can be reduced to it in a
way similar to the one we used for computing path lengths of arbitrary query points (see Subsection 2.5.4).

We pre-process each shortest path tree so that the following type of queries can be quickly answered: given a vertex \( v \) in the tree and a positive integer \( i \), find the \( i \)-th vertex on the path from \( v \) to the root of the tree. Such queries are called *level-ancestor queries* by Berkman and Vishkin [21], who gave efficient parallel algorithms for pre-processing rooted trees so that the level-ancestor queries can be answered quickly. The work of Berkman and Vishkin [21, 22] shows (implicitly) that a level-ancestor query can be handled sequentially in constant time, after a logarithmic time and linear work pre-processing on the CREW PRAM. The pre-processing of the shortest path trees is done by simply applying the result of Berkman and Vishkin to each of the \( O(n) \) trees, in totally \( O(\log n) \) time and \( O(n^2) \) work.

For the sake of processor assignment in reporting paths, we also need to compute the number of segments on the actual shortest path which is to be reported. Suppose a shortest path between vertices \( v \) and \( w \) in \( V_R \) is to be reported. The number of segments on such a \( v \)-to-\( w \) path can be obtained from the depth of \( w \) in the shortest path tree rooted at \( v \); it is known that the depths can be computed within the required complexity bounds by using the Euler Tour technique [124].

To report an actual shortest path between vertices \( v \) and \( w \) in \( V_R \), we do the following. First, we go to the shortest path tree rooted at (say) \( v \), and find the number of segments on the path in the tree from node \( w \) to the root \( v \). Let that number be \( k \). The \( w \)-to-\( v \) path in the tree corresponds to a geometric shortest path between \( v \) and \( w \), which we must report. We do so by performing, in parallel, \( \lceil k / \log n \rceil - 1 \) level-ancestor queries, using node \( w \) and integers \( \lceil \log n \rceil, 2\lceil \log n \rceil, \ldots, (\lceil k / \log n \rceil - 1)\lceil \log n \rceil \). Each query is handled by one processor in \( O(1) \) time. These queries cut the \( w \)-to-\( v \) path into \( \lceil k / \log n \rceil \) pieces of \( O(\log n) \) segments each. Finally, we report the \( \lceil k / \log n \rceil \) pieces of the path in parallel by assigning one processor to output each piece of the path sequentially.
2.8 A Note on the Sequential Time Complexity

In this section we make a fairly straightforward observation about the sequential time complexity of the problem we considered (but one that, to the best of our knowledge, has not yet been documented). We describe an \(O(n^2)\) time sequential algorithm for building the data structure that supports the fast processing of the length and path queries (i.e., \(O(\log n)\) time for a length query, and \(O(\log n + k)\) time for a path query, where \(k\) is the number of segments on the path reported). In this sequential algorithm, we take a topological sort [4] approach, which is very different from the divide-and-conquer approach used in our parallel algorithms.

We only discuss how to compute the \(V_R\)-to-\(V_R\) matrix of path lengths, because we have shown (in Sections 2.5 and 2.7) that the other components of the data structure can be computed in \(O(n^2)\) work (hence \(O(n^2)\) sequential time). Recall that these components are the two planar subdivisions \(H_1\) and \(H_2\), the \(X(v)\) paths for every \(v \in V_R\), and the shortest path trees rooted at the vertices in \(V_R\), where \(X = NE, NW, \ldots\), etc.

Note that there is a sequential algorithm in [50] that optimally solves the single source case of the problem for computing rectilinear shortest paths avoiding rectangular obstacles. The algorithm in [50] uses the plane sweeping technique. This algorithm can be used to compute, in \(O(n \log n)\) time, the lengths of the shortest paths between a chosen vertex \(v\) in \(V_R\) (designated as the fixed source point) and the vertices in \(V_R - \{v\}\). Hence the \(V_R\)-to-\(V_R\) lengths matrix can be obtained by simply applying the algorithm of [50] \(O(n)\) times (each time a different vertex in \(V_R\) is designated as the fixed source point), in totally \(O(n^2 \log n)\) time.

The \(O(n^2)\) time algorithm is based on the geometric observations given in [50]. The only thing we do differently is that, when computing the path lengths between a fixed vertex \(v\) and the vertices in \(V_R - \{v\}\), we do not use plane sweeping. Rather, we do topological sorts [4] on \(O(n)\) directed acyclic graphs of size \(O(n)\) each. These
directed graphs will be built using trapezoidal decomposition [114] and the $X(v)$
paths for all $v \in V_R$.

First, we show how to build the $O(n)$ directed graphs. For a vertex $v \in V_R$, there are four directed acyclic graphs associated with it. Consider the shortest paths between $v$ and the vertices in $V_R - \{v\}$. The four graphs of $v$ correspond to the following four cases of the shortest paths: (i) those monotone with respect to the $x$-axis and with $v$ as their left endpoints, (ii) those monotone with respect to the $x$-axis and with $v$ as their right endpoints, (iii) those monotone with respect to the $y$-axis and with $v$ as their upper endpoints, and (iv) those monotone with respect to the $y$-axis and with $v$ as their lower endpoints. We only show how to compute for case (i) (the other cases are handled similarly). Let $V_R$ be given sorted by the $y$-coordinates.

Suppose that we already know the following information: for the right edge $e$ of each obstacle, the vertex set $Hit(e)$ (recall that this is the set of vertices in $V_R$ whose horizontal leftward ray-shootings hit $e$). (Computing these sets is done during the pre-processing, by using trapezoidal decomposition [114].) Let $u_1$ and $u_2$ be the two vertices of $e$. For each $w \in Hit(e)$, the path length between $w$ and $u_1$ (resp., $w$ and $u_2$) is simply $d(u_1, w)$ (resp., $d(u_2, w)$) and can be trivially computed in $O(1)$ time.

It has been shown in [50] that a shortest path between $v$ and a point $p$ is of case
(i) if $p$ is on or is to the right of $NE(v) \cup SE(v)$. We do the following. (1) Find all
the vertices in $V_R$ that are on or to the right of $NE(v) \cup SE(v)$; this can be easily
done in $O(n)$ time by merging $V_R$ (sorted by the $y$-coordinates) and $NE(v) \cup SE(v)$.
Let the set of the vertices that are on or to the right of $NE(v) \cup SE(v)$ be $Right(v)$.
$Right(v)$ is the vertex set of the graph. (2) For every vertex $u \in Right(v)$ whose
horizontal leftward ray-shooting crosses $NE(v) \cup SE(v)$ before reaching an obstacle,
compute the length of its path to $v$, which is simply $d(v, u)$ (note: there will be no
incoming edge for such a vertex $u$ in the graph). (3) For every vertex $w \in Right(v)$
whose horizontal leftward ray-shooting does not cross $NE(v) \cup SE(v)$, let $e$ be the
right edge of an obstacle such that $w \in Hit(e)$, and let $u_1$ and $u_2$ be the two vertices
of $e$ ($u_1, u_2 \in Right(v)$); associate with $u_1$ (resp., $u_2$) a pointer to $w$ and assign the
pointer a weight equal to \(d(u_1, w)\) (resp., \(d(u_2, w)\)) (note: \(w\) has exactly two incoming edges in the graph, one from \(u_1\) and the other from \(u_2\)). The construction of this graph for vertex \(v\) clearly requires \(O(n)\) time.

The directed graph for vertex \(v \in V_R\) so constructed is acyclic because of the monotonicity property of the shortest paths in case (i), and it obviously has \(O(n)\) vertices and directed edges. The undirected version of the graph may have more than one connected component. A shortest \(v\)-to-\(w\) path in it, when \(w \in \text{Right}(v)\), corresponds to a shortest geometric path between \(v\) and \(w\). The single-source shortest paths problem in such a graph can easily be solved in linear time, since it is acyclic. Therefore the \(V_R\)-to-\(V_R\) path lengths matrix can be computed in \(O(n^2)\) time.
3. VISIBILITY OF A SIMPLE POLYGONAL CHAIN FROM A POINT

Given a set $S$ of geometric objects, two distinct points $p_1$ and $p_2$ are said to be visible to each other iff the interior of the line segment having $p_1$ and $p_2$ as its two endpoints does not intersect any object in $S$. Intuitively, the objects in $S$ can be viewed as being “opaque.”

Visibility is one of the most fundamental topics in computational geometry. Visibility problems find applications in many areas, such as computer graphics, VLSI design, and robotics. Also, visibility problems often appear as subproblems of many other problems (like finding shortest obstacle-avoiding paths and computing intersections of geometric figures). In this chapter, we consider the following visibility problem: given a point $q$ and an $n$-vertex simple polygonal chain $P$ in the plane, find all the points on $P$ that are visible from $q$ if $P$ is opaque (see Figure 3.1 for example). In his book on art gallery problems and algorithms, O’Rourke argues that this problem is perhaps the most fundamental problem in visibility [110]. Our goal here is to provide an efficient parallel algorithm for this problem on the EREW PRAM.

Using the “cascading divide-and-conquer” technique, Atallah et al. [16, 43] provided a long list of optimal algorithms for geometric problems, in particular an EREW PRAM algorithm for the visibility problem when the opaque objects are $n$ non-intersecting planar line segments. The algorithm for this visibility problem in [16] runs in $O(\log n)$ time using $O(n)$ processors, which is optimal for $n$ arbitrary non-intersecting line segments. Very recently, Bertolazzi et al. [23] considered this problem (for $n$ non-intersecting planar line segments) and gave an optimal algorithm that runs in $O(\log n)$ time using $O(n)$ processors on the CREW PRAM. The technique used in [23] is the many-way divide-and-conquer strategy. Both the algorithms in [16, 23] are optimal for the case of $n$ non-intersecting planar line segments, but are suboptimal
Figure 3.1 An example of the visibility chain $VIS(P)$.

when the line segments form a simple (possibly closed) polygonal chain. No modification of [16] and [23] seems to yield an optimal EREW PRAM algorithm for the case where the line segments form a simple polygonal chain.

Indeed, in order to obtain an optimal EREW PRAM algorithm for the simple polygonal chain case, we follow a very different approach, and present an algorithm that takes $O(\log n)$ time and uses $O(n/\log n)$ EREW PRAM processors. The contribution of this chapter is actually twofold: first, it provides the first optimal parallel algorithm on the EREW PRAM for the problem of visibility of a simple polygonal chain from a point, which also gives efficient parallel algorithms for other geometric problems on a simple polygonal chain (some of them are given in Section 3.5 of this chapter and in Chapter 4); second, it presents geometric insights that allow efficient detection of intersections between the visibility chains of different portions of the polygonal chain. These insights are likely to be useful in solving other problems about simple polygonal chains.

This algorithm is optimal to within a constant factor because (i) there is an obvious $\Omega(n)$ sequential lower bound for the problem, and (ii) an $\Omega(\log n)$ lower bound in its EREW PRAM time complexity can be obtained by reducing to it the problem of computing the maximum of $n$ entries (the reduction is easy and is omitted). Several
sequential algorithms [51, 56, 83, 93] have solved the problem within a linear time bound.

In the next section, we give the notation and definitions used in this chapter, and some preliminary results. An overview of the algorithm is sketched in Section 3.2. Section 3.3 presents the crucial geometric insights and the algorithm based on them. Section 3.4 addresses the EREW PRAM implementation of the algorithm. Section 3.5 gives some applications of the algorithm.

3.1 Preliminaries

The input consists of a point $q$ and a simple polygonal chain $P = (v_1, v_2, \ldots, v_n)$ in the plane (possibly $v_1 = v_n$), where the given sequence of vertices is such that when we visit them in the order $v_1, v_2, \ldots, v_n$, we are traveling along chain $P$ and encounter each point on $P$ exactly once (except at the starting point $v_1$ if $v_1 = v_n$). Let $s_i$ denote the segment of $P$ joining $v_i$ to $v_{i+1}$. The order in which a walk along $P$ from $v_1$ to $v_n$ encounters the $v_i$'s is called the chain order and is denoted by $<_P$. We say $v_i$ has rank $i$ in the chain order, and denote it by $rank(v_i)$. For example, $v_3 <_P v_9$ since $rank(v_3) = 3 < 9 = rank(v_9)$. We extend the notion of rank to all the points on $P$ as follows: if $p$ is a point in the interior of segment $s_i$, then $rank(p) = rank(v_i) = i$.

If $u$ and $w$ are two points in the plane, then $uw (= uw)$ denotes the line segment joining them. We assume that every chain we consider in this chapter is simple, that is, no two segments in it intersect each other (except possibly at their endpoints), and $v_i \neq v_j$ for every $i \neq j$ except that possibly $v_1 = v_n$ (if $v_1 = v_n$, then $P$ is closed, otherwise it is open). From now on, all chains are assumed to be open because the closed case is reduced to the open case by first "opening" it by removing a segment $s$ from it (any $s$ will do), then solving the visibility problem for $P - s$ using the algorithm for the open case, and finally including the effect of $s$ in $O(\log n)$ additional time using the $n/\log n$ processors available. Each chain $C$ has a length, denoted by $|C|$, which is the number of line segments in it. Given a chain $C$, let $Q$ be the star-shaped
polygon consisting of the portion of the plane visible from \( q \) when \( C \) is the only opaque object. Then \( VIS(C) \), the visibility chain of \( C \) from \( q \), equals the boundary of \( Q \) minus the (at most two) edges on the boundary of \( Q \) that are incident to the point at infinity (see Figure 3.1 for example). Once we have \( VIS(C) \), it is easy to extract from it the portions of \( C \) that are visible from \( q \) (i.e., \( VIS(C) \cap C \)) by a parallel prefix computation [88, 89] which removes the segments of \( VIS(C) \) that are not the segments of \( C \). Hence, our goal from now on is to compute \( VIS(P) \) for the input polygonal chain \( P \).

A point \( p \) is represented by its \( x \)-coordinate and \( y \)-coordinate, denoted by \( x(p) \) and \( y(p) \), respectively. Without loss of generality, we assume that \( q \) is at the origin of coordinates, i.e., \( x(q) = 0 \) and \( y(q) = 0 \). We often refer to the polar angle of a point \( p \), denoted by \( \theta(p) \), which is the angle of vector \( \overrightarrow{qp} \) with respect to the positive \( x \)-axis (measured counterclockwise, with \( 0 \leq \theta(p) < 2\pi \)). Although we refer to polar angles frequently, we do not need to explicitly compute them (in fact we can use \( x(p) \) and \( y(p) \) as an implicit representation of \( \theta(p) \) and thus avoid the computation of inverse trigonometric functions).

Note that \( VIS(C) \) is "monotone" with respect to \( q \), in the sense that if a half-line originating at \( q \) intersects \( VIS(C) \), then it cuts \( VIS(C) \) either at a single point, or along a segment collinear with \( q \) and connecting two visible portions of \( C \) (see Figure 3.2). Except for their endpoints, these line segments of \( VIS(C) \) that are collinear with \( q \) do not belong to \( C \), and we therefore call them the extra segments of \( VIS(C) \).

In Figure 3.2, \( C \) is the chain from \( v_1 \) to \( v_m \), and the segments of \( VIS(C) \) are, in counterclockwise order, \( \overline{lu}, \overline{uv}, \overline{vw}, \overline{wf}, \overline{fw'}, \overline{w'v'}, \overline{v'u'}, \) and \( \overline{u'r} \) (\( \overline{vw} \) and \( \overline{w'v'} \) are the extra segments). The angular interval of \( VIS(C) \), denoted as \( I(C) \), is defined as follows. It is \([0,2\pi]\) if \( VIS(C) \) is closed (i.e., it has no beginning and end), otherwise it is the interval of polar angles \([\theta(l), \theta(r)]\) (counterclockwise) where \( l \) and \( r \) are, respectively, the first and last points of \( VIS(C) \) encountered by a counterclockwise angular scan of \( VIS(C) \) (see Figure 3.2). Note that \( ||\theta(l), \theta(r)|| + ||\theta(r), \theta(l)|| = 2\pi \).

In the second case (when \( VIS(C) \) is not closed), we refer to \( l \) and \( r \) as the endpoints
of \(\text{VIS}(C)\) (note that they need not coincide with the endpoints of \(C\)). In Figure 3.2, the endpoints of \(C\) are \(v_1\) and \(v_m\), while the endpoints of \(\text{VIS}(C)\) are \(l\) and \(r\).

The monotonicity of \(\text{VIS}(C)\) enables us to store it in the leaves of a binary search tree structure that allows a processor to search in the tree, in time proportional to its height, by polar angle (i.e., “find the point \(p\) in \(\text{VIS}(C)\) whose polar angle is \(\theta(p)\)”) or, alternatively, by leaf order (i.e., “find the \(t\)-th vertex of \(\text{VIS}(C)\) starting from vertex \(v\) and moving counterclockwise”). The tree structure also supports “split” operations in time proportional to its height (i.e., “remove from the tree all the leaves whose polar angles lie in \([\theta_1, \theta_2]\) and put them in a tree of their own”). Even if these splits are done very naively (i.e., if each of the two trees resulting from a split has the same height as the original one), we shall later (in Subsection 3.3.1) show that the height of this binary search tree for \(\text{VIS}(C)\) remains logarithmic in \(|C|\). To avoid introducing new terminology, we also use the same symbol (i.e., \(\text{VIS}(C)\)) to denote both the visibility chain of \(C\) and the balanced tree data structure describing it.

We say that point \(v\) is \textit{behind} point \(w\) (with respect to \(q\)) if \(\theta(v) = \theta(w)\) and \(w\) is on the segment joining \(q\) to \(v\) (equivalently, we can say that \(w\) is \textit{in front of} \(v\)). Suppose that chain \(C\) is partitioned into \(k\) subchains and the visibility chain of each subchain is available. Then when we talk about \textit{combining} the \(k\) visibility chains, we mean computing \(\text{VIS}(C)\) from these \(k\) visibility chains.
To simplify the exposition, we assume that no segment of \( P \) is collinear with \( q \), and that no two consecutive segments of \( P \) are collinear (the general case can be included in our solution without much difficulty).

### 3.2 An Overview of the Algorithm

We call \( \text{VisChain} \) the recursive procedure for computing the visibility chain of a simple polygonal chain. The procedure is outlined below. The initial call to the procedure is \( \text{VisChain}(P, n, \log n) \), where \( P \) is a simple polygonal chain and \( n = |P| \).

\[ \text{VisChain}(C, m, d) \]

**Input.** A subchain \( C \) of \( P \), \( m = |C| \), and a positive integer \( d \) of our choice.

**Output.** The visibility chain of \( C \), \( \text{VIS}(C) \), from the point \( q \).

**Case A.** If \( m \leq d \), then compute \( \text{VIS}(C) \) with one processor in \( O(m) \) time, using any of the known sequential linear-time algorithms.

**Case B.** If \( d < m \leq d^2 \), then divide \( C \) into two subchains \( C_1 \) and \( C_2 \) of equal length and recursively call \( \text{VisChain}(C_1, |C_1|, d) \) and \( \text{VisChain}(C_2, |C_2|, d) \) in parallel. Then compute \( \text{VIS}(C) \) from \( \text{VIS}(C_1) \) and \( \text{VIS}(C_2) \), in \( O((\log m)^2) \) time and using one processor.

**Case C.** If \( m > d^2 \), then partition \( C \) into \( g = (m/d)^{1/4} \) subchains \( C_1, C_2, \ldots, C_g \) of length \( m^{3/4}d^{1/4} \) each. Call \( \text{VisChain}(C_1, |C_1|, d) \), \( \text{VisChain}(C_2, |C_2|, d) \), \( \ldots \), \( \text{VisChain}(C_g, |C_g|, d) \), in parallel. Then compute \( \text{VIS}(C) \) from \( \text{VIS}(C_1) \), \( \text{VIS}(C_2) \), \( \ldots \), \( \text{VIS}(C_g) \), in \( O(\log m) \) time and using \( m/d = g^4 \) processors.

The main difficulty lies in the "conquer" stages: two visibility chains \( \text{VIS}(C_i) \) and \( \text{VIS}(C_j) \), \( i \neq j \), can have two intersections, and we have (cf. Case C above) only \( g^2 = (m/d)^{1/2} \) processors to compute these two intersections between each pair \( \langle \text{VIS}(C_i), \text{VIS}(C_j) \rangle \). Doing this in \( O(\log m) \) time may appear impossible at first sight: the length of each of \( \text{VIS}(C_i) \) and \( \text{VIS}(C_j) \) can be \( m^{3/4}d^{1/4} \), and there is a well-known
linear lower bound [31] on the work needed for computing the two intersections of two arbitrary polygonal chains that intersect twice (even if both chains are convex). This seems to imply that, since we have only \((m/d)^{1/2}\) processors assigned to the task, it will take \(m^{3/4}d^{1/4}(m/d)^{-1/2} = m^{1/4}d^{3/4}\) time rather than the claimed \(O(\log m)\). What enables us to achieve \(O(\log m)\) time is the fact that both \(C_i\) and \(C_j\) are subchains of a simple polygonal chain. How to exploit this fact is one of the main contributions of this chapter.

Observe that, if we could perform the various cases of the above algorithm within the claimed bounds, then it would indeed run in \(O(d + \log m)\) time with \(O(m/d)\) processors since its time and processor complexities would satisfy the following recurrences:

\[
\begin{align*}
t(m, d) &= \begin{cases} 
c_1 m & \text{if } m \leq d \\
t(m/2, d) + c_2 (\log m)^2 & \text{if } d < m \leq d^2 \\
t(m^{3/4}d^{1/4}, d) + c_3 \log m & \text{if } m > d^2
\end{cases} \\
p(m, d) &= \begin{cases} 
1 & \text{if } m \leq d \\
\max\{2p(m/2, d), m/d\} & \text{if } d < m \leq d^2 \\
\max\{(m/d)^{1/4}p(m^{3/4}d^{1/4}, d), m/d\} & \text{if } m > d^2
\end{cases}
\end{align*}
\]

where \(c_1, c_2, c_3\) are constants. From the above recurrences, the following bounds for \(t(m, d)\) and \(p(m, d)\) are easy to prove by induction:

\[
\begin{align*}
t(m, d) &\leq \begin{cases} 
\alpha_1 d & \text{if } m \leq d \\
\alpha_2 d + \beta_2 (\log m)^2 \log d & \text{if } d < m \leq d^2 \\
\alpha_3 d + \beta_3 \log m & \text{if } m > d^2
\end{cases} \\
p(m, d) &= \begin{cases} 
1 & \text{if } m \leq d \\
m/d & \text{if } m > d
\end{cases}
\end{align*}
\]

where \(\alpha_1, \alpha_2, \beta_2, \alpha_3, \beta_3\) are constants.
Choosing \( d = \log n \), the above implies that 
\[ t(n, \log n) = O(\log n) \] 
and 
\[ p(n, \log n) = O(n / \log n) \]. Hence the call \( \text{VisChain}(P, n, \log n) \) would compute \( \text{VIS}(P) \) in \( O(\log n) \) time using \( O(n / \log n) \) processors.

Thus, in the rest of this chapter, it suffices for us to show how, with \( m/d \) processors, to do the “combine” part of Case B in \( O((\log m)^2) \) time, and more importantly, how to implement the “combine” part of Case C in \( O(\log m) \) time.

We use the terminology of the above outline in the rest of this chapter, so that a \( C_i \) is one of the subchains from the partition of \( C \), and \( \text{VIS}(C_i) \) is already available from the recursive call that computed it (i.e., we are focusing on the “combine” part of the algorithm). We define \( B_i \) to be the subchain of \( C \) which is \textit{before} \( C_i \) along the chain order, that is, \( B_i \) consists of the concatenation of \( C_1, C_2, \ldots, C_{i-1} \). The subchain \( A_i \) of \( C \) which is \textit{after} \( C_i \) along the chain order is defined similarly, that is, \( A_i \) consists of the concatenation of \( C_{i+1}, C_{i+2}, \ldots, C_g \). Note that if points \( b_i, c_i, \) and \( a_i \) belong to \( B_i, C_i, \) and \( A_i \), respectively, then \( b_i \leq_P c_i \leq_P a_i \).

### 3.3 Visibility Chains and Their Intersections

This section presents the geometric insights together with their algorithmic implications. The most crucial insights are Lemma 3.4 and Lemma 3.5.

#### 3.3.1 Simple Geometric Facts

Let \( s = \overline{ab} \) be any straight line segment in \( \text{VIS}(C) \cap C \), where \( C \) is a subchain of \( P \) and \( a \) is encountered before \( b \) by a \( v_1 \)-to-\( v_n \) walk along \( P \). Then \( s \) is \textit{clockwise} if \( s \) is traversed in the clockwise direction (with respect to \( q \)) by such a walk, and is \textit{counterclockwise} otherwise. For example, in Figure 3.2, segments \( \overline{uv} \) and \( \overline{uw'} \) are clockwise, while segments \( \overline{wf} \) and \( \overline{fw'} \) are counterclockwise. Let \( p \) be any point of \( \text{VIS}(C) \cap C \). If \( p \) is not a vertex of \( \text{VIS}(C) \), then \( p \) is said to have a \textit{clockwise} (resp., \textit{counterclockwise}) \textit{arrow tag} iff the segment of \( \text{VIS}(C) \) to which \( p \) belongs is clockwise (resp., counterclockwise). If \( p \) is a vertex of \( \text{VIS}(C) \), then let \( s \) and \( s' \) be the two segments of \( \text{VIS}(C) \) having \( p \) as their common endpoint (it is possible that one of
s or s' does not exist). Observe that, if none of s or s' is an extra segment (i.e., neither s nor s' is collinear with q), then either both of these segments are clockwise, or both are counterclockwise. If both are clockwise (resp., counterclockwise), then p is said to have a clockwise (resp., counterclockwise) arrow tag. If one of \{s, s'\} is an extra segment or does not exist (in case p is an endpoint of VIS(C)), say it is s', then p has a clockwise (resp., counterclockwise) arrow tag iff s is clockwise (resp., counterclockwise). In Figure 3.2, the arrow tags of \( v, u, l, r, u', \) and \( v' \) are clockwise, while those of \( w, f, \) and \( w' \) are counterclockwise.

Lemma 3.1 Let \( C \) be a simple chain and \( C' \) be a subchain of \( C \). Then \( VIS(C) \cap VIS(C') \) has at most three connected components (i.e., at most three separate portions of \( VIS(C') \) appear in \( VIS(C) \)). If \( C - C' \) has a single connected component (i.e., \( C' \) is at the beginning or the end of \( C \)), then \( VIS(C) \cap VIS(C') \) has at most two connected components.

Proof. We begin with the proof of the part when \( C' \) is at the beginning or the end of \( C \). By contradiction, suppose that \( VIS(C) \cap VIS(C') \) has three connected components. Let \( a, b, c \) be arbitrary points on each of these three components, respectively, with \( \theta(a) < \theta(b) < \theta(c) \). Let \( u \) and \( w \) be points of \( VIS(C) - VIS(C') \) such that \( \theta(a) < \theta(u) < \theta(b) < \theta(w) < \theta(c) \). Now, the path \( Q \) in \( C - C' \) joining \( u \) to \( w \) cannot pass in front of any of \( \{a, b, c\} \), and therefore \( Q \) must join \( u \) to \( w \) in a way that isolates \( b \) from both \( a \) and \( c \) (i.e., \( Q \) either passes behind \( b \) or passes behind both \( a \) and \( c \)), making it impossible for \( C' \) to go through \( b \) without crossing \( Q \) (see Figure 3.3 (a)). This contradicts the fact that \( C \) is simple. We now prove the part when \( C' \) is neither at the beginning nor at the end of \( C \). By contradiction, assume that \( VIS(C) \cap VIS(C') \) has four connected components, and let \( a, b, c, e \) be points on each of those four components, respectively, with \( \theta(a) < \theta(b) < \theta(c) < \theta(e) \). Let \( u, v, w \) be points of \( VIS(C) - VIS(C') \) such that \( \theta(a) < \theta(u) < \theta(b) < \theta(v) < \theta(c) < \theta(w) < \theta(e) \) (see Figure 3.3 (b)). Let \( A \) and \( B \) be the two connected components of \( C - C' \). By the pigeonhole principle, at least two points in the set \( \{u, v, w\} \) are both in \( A \) or both in
Figure 3.3 Illustrating the proof of Lemma 3.1.

$B$ (say, both in $A$). But then, $\text{VIS}(A \cup C') \cap \text{VIS}(C')$ has three connected components, contradicting the already proven part of the lemma.

The above lemma implies that, in order to obtain from the tree for $\text{VIS}(C_i)$ the portions of $\text{VIS}(C_i)$ that are visible in $\text{VIS}(C)$, we need to perform only a constant number of "split" operations on the tree representing $\text{VIS}(C_i)$ (which can be done by one processor in $O(\log m)$ time). Over all $i, 1 \leq i \leq g$, the total of $O(g)$ such splits on $\text{VIS}(C_i)$'s results in $g' \leq 3g$ trees that are then used to create the tree for $\text{VIS}(C)$ by simply building a complete binary tree "on top" of the roots of the $g'$ trees, resulting in the height of $\text{VIS}(C)$ being higher by $\log(3g)$ ($= O(\log m)$) than the highest of the $\text{VIS}(C_i)$ trees. Given the angular order of the leaves of the "on top" complete binary tree, the building of the complete binary tree can be easily done in $O(\log m)$ time using $O(g)$ EREW PRAM processors. We shall explain later (in this subsection) how to obtain the correct ordering of the $g'$ trees used to build $\text{VIS}(C)$. For now, we simply observe that this method of building $\text{VIS}(C)$ from the $\text{VIS}(C_i)$'s results in the height of $\text{VIS}(C)$ being logarithmic in $|C|$ (the proof of this is by an easy induction).

Let $C'$ and $C''$ be two subchains of $C$ such that $C'$ and $C''$ are disjoint except possibly at a common endpoint. Since both $\text{VIS}(C')$ and $\text{VIS}(C'')$ can contain points
of VIS(C), and since they can intersect each other, we would like to compute exactly where the intersections occur in order to find which portions of VIS(C') are hidden by VIS(C'') (and vice versa). The next lemma ensures that the number of intersections between the two visibility chains VIS(C') and VIS(C'') is no more than two. Recall that I(C) denotes the angular interval of C (defined in Section 3.1).

Lemma 3.2 If C' and C'' are two subchains of C that are disjoint except that they may share one endpoint, then there are at most two intersections between VIS(C') and VIS(C''). Furthermore, if VIS(C') and VIS(C'') have two intersections and I(C') \cap I(C'') consists of two disjoint intervals, then there is exactly one intersection in each such interval. If VIS(C') and VIS(C'') have two intersections and I(C') \cap I(C'') consists of one interval, then one of I(C') or I(C'') contains the other.

Proof. Clearly, I(C') \cap I(C'') consists of no more than two disjoint intervals. The lemma would follow if we could show that there do not exist four points a, b, c, e such that \( \theta(a) < \theta(b) < \theta(c) < \theta(e) \), a and c are in VIS(C') and are not hidden from q by C'', while b and e are in VIS(C'') and are not hidden from q by C'. Suppose to the contrary that four such points exist. The only way C' and C'' can link a to c and (respectively) b to e without hiding any of the four points \{a, b, c, e\} would require an intersection between C' and C'', contradicting the fact that C is simple. \( \square \)

Figure 3.4 gives examples for the two possible cases where VIS(C') and VIS(C'') have two intersections. When VIS(C') intersects VIS(C''), I(C') \cap I(C'') must be nonempty. The two cases are such that either one of I(C') or I(C'') contains the other (Figure 3.4 (a)), or none of I(C') and I(C'') contains the other (Figure 3.4 (b)). The more difficult case of the two is that of Figure 3.4 (a). In fact, for the case of Figure 3.4 (b), it is easy to compute the two intersections by using a procedure for solving the one-intersection cases (a reduction of the two-intersection case of Figure 3.4 (b) to the one-intersection cases follows from Lemma 3.2). However, Lemma 3.2 does not imply a reduction of the two-intersection case for Figure 3.4 (a) to the one-intersection cases.
Figure 3.4 Illustrating the two possible two-intersection cases.
Although Lemma 3.2 limits to two the number of possible intersections between the visibility chains of two subchains of C that are disjoint (except possibly at a common endpoint), the linear-work lower bound for detecting intersections between polygonal chains proven by Chazelle and Dobkin [31] holds even for two chains that intersect each other no more than twice. We shall exploit the fact that the two chains are subchains of a simple chain in order to get around the lower bound when we solve the case of Figure 3.4 (a). Specifically, the rest of this section shows how to compute, for each $C_i$, the (by Lemma 3.1, at most two) portions of $VIS(C_i)$ that are hidden by $A_i$ (computing the portions of $VIS(C_i)$ hidden by $B_i$ is done in a symmetrical way and is therefore omitted). Note that there can be two intersections between $VIS(A_i)$ and $VIS(C_i)$, and we must compute these intersections in order to compute the (by Lemma 3.1, at most three) portions of $VIS(C_i)$ not hidden by $A_i \cup B_i$. The computation of the portions of $VIS(C_i)$ hidden by $A_i \cup B_i$ immediately gives us the (at most three) portions of $VIS(C_i)$ that belong to $VIS(C)$. Once we have done this (in parallel) for every $i \in \{1, \ldots, g\}$, it is easy to "stitch" the resulting $g' \leq 3g$ pieces of $VIS(C)$ and create $VIS(C)$: first split the trees representing $VIS(C_1)$, $VIS(C_2)$, $VIS(C_g)$, in order to discard all the portions of the $VIS(C_i)'s$ that are invisible in $VIS(C)$; then the problem essentially becomes that of sorting (by the polar angles) the $O(g)$ endpoints of those portions of the $VIS(C_i)'s$ that are visible in $VIS(C)$, which can be done in time $O(\log m)$ using $O(g)$ EREW PRAM processors [43]. We have $g^4$ processors available, more than enough to do this sorting. Thus we are justified in focusing, for the rest of this section, on the problem of determining the portions of $VIS(C_i)$ that are hidden by $A_i$.

3.3.2 Simple Computational Observations

We next observe that, although $VIS(A_i)$ is not available after the recursive calls of Case C return the $VIS(C_j)'s$, we can still use the $g^3$ processors assigned to each $C_i$ in order to answer meaningful queries about $VIS(A_i)$. 
Lemma 3.3 Let the \( \text{VIS}(C_i) \)'s be given. For a fixed \( i \in \{1, 2, \ldots, g\} \), let \( l \) be a half-line originating at \( q \), and let \( w \) be the intersection point between \( l \) and \( \text{VIS}(A_i) \) that is closest to \( q \). Then \( g \) processors suffice for computing, in \( O(\log m) \) time, the point \( w \) and the arrow tag of \( w \) on \( \text{VIS}(A_i) \).

Proof. Although we do not have \( \text{VIS}(A_i) \) itself, we know that \( w \) is one of the \( g - i \) points determined by the \( g-i \) intersections of \( l \) with each of \( \text{VIS}(C_{i+1}), \text{VIS}(C_{i+2}), \ldots, \text{VIS}(C_g) \). Thus \( w \) can be obtained in \( O(\log m) \) time by (i) computing the intersection between \( l \) and each of \( \text{VIS}(C_{i+1}), \text{VIS}(C_{i+2}), \ldots, \text{VIS}(C_g) \), then (ii) choosing the intersection point that is closest to \( q \). As for the arrow tag computation, it too is done in \( O(\log m) \) time by computing the immediate predecessor and successor of \( w \) on \( \text{VIS}(A_i) \); these are easy to obtain, since they are determined by the set of \( 2(g-i) \) vertices that are adjacent to \( \theta(l) \) on each of \( \text{VIS}(C_{i+1}), \text{VIS}(C_{i+2}), \ldots, \text{VIS}(C_g) \). \( \Box \)

Corollary 3.1 Let the \( \text{VIS}(C_i) \)'s be given. For every \( i \in \{1, 2, \ldots, g\} \), given \( k \) half-lines \( \{l_{i1}, \ldots, l_{ik}\} \) originating at \( q \), sorted by their polar angles in counterclockwise order, let \( w_{ij}, 1 \leq j \leq k, \) be the intersection point between \( l_{ij} \) and \( \text{VIS}(A_i) \) that is closest to \( q \). With \( gk \) processors assigned to every \( C_i \), each \( w_{ij} \) and its arrow tag on \( \text{VIS}(A_i) \) can be computed in \( O(\log m) \) time.

Proof. For every \( \text{VIS}(A_i) \), to compute the intersection point \( (w_{ij}) \) between \( \text{VIS}(A_i) \) and half-line \( l_{ij} \), assign \( g \) of the \( gk \) available processors of \( C_i \) to half-line \( l_{ij} \), and use Lemma 3.3. While this computation apparently requires "common read" capability (i.e., CREW rather than EREW), we shall show in Section 3.4 how to implement it in the EREW model (recall that the CREW PRAM is a stronger parallel model than the EREW PRAM since it allows concurrent read accesses to the same memory address by more than one processor). \( \Box \)

3.3.3 The Relative Positions of \( A_i \) and \( \text{VIS}(C_i) \)

This subsection gives a classification of the various possible relative positions of \( A_i \) and \( \text{VIS}(C_i) \), based on Lemma 3.2. We do not yet compute the actual intersections.
of \(\text{VIS}(A_i)\) and \(\text{VIS}(C_i)\) (if any): this is postponed until the next subsection, when we will have developed more machinery for the computation of intersections (the most difficult cases will turn out to be those where two intersections might occur). Each of the cases and subcases below can easily be seen to be identifiable in \(O(\log m)\) time by using Corollary 3.1, where by "identifying a subcase", we mean just ascertaining that the subcase holds, not actually computing the portions of \(\text{VIS}(C_i)\) hidden by \(A_i\) in that subcase.

Note that \(I(A_i) \cap I(C_i)\) can be easily computed from \(I(C_i), I(C_{i+1}), \ldots, I(C_g)\) in \(O(\log m)\) time using \(O(g)\) processors. Suppose for the time being that \(I(A_i) \cap I(C_i) \neq [0, 2\pi)\) (i.e., there is a half-line originating at \(q\) that goes to infinity without intersecting either \(\text{VIS}(A_i)\) or \(\text{VIS}(C_i)\)). Notice that \(I(A_i) \cap I(C_i)\) can consist of up to two disjoint intervals (see Figure 3.4 (b) for an example). Let \([\theta_1, \theta_2]\) be one such interval; the case analysis below will hold for the other interval as well (if there are two of them). Let \(a_1\) and \(a_2\) denote the two points closest to \(q\) on \(\text{VIS}(A_i)\) such that \(\theta(a_1) = \theta_1\) and \(\theta(a_2) = \theta_2\), and let \(c_1\) and \(c_2\) denote the two points closest to \(q\) on \(\text{VIS}(C_i)\) such that \(\theta(c_1) = \theta_1\) and \(\theta(c_2) = \theta_2\). The points \(a_1\) and \(a_2\) are obtained in \(O(\log m)\) time using Corollary 3.1, while the points \(c_1\) and \(c_2\) are easily obtained from \(\text{VIS}(C_i)\) in \(O(\log m)\) time by a simple one-processor search.

Case 1. \(\theta_1 = \theta_2\). In this case, it is clear that no portion of \(\text{VIS}(C_i)\) in interval \([\theta_1, \theta_2]\) is hidden by \(A_i\).

Case 2. \(\theta_1 \neq \theta_2\) and neither \(I(A_i)\) nor \(I(C_i)\) contains the other. There are three subcases.

Subcase 2.1. Both \(c_1\) and \(c_2\) are in front of \(\text{VIS}(A_i)\). Then there is no intersection between the portions of \(\text{VIS}(A_i)\) and \(\text{VIS}(C_i)\) in interval \([\theta_1, \theta_2]\), and the portion of \(\text{VIS}(C_i)\) in \([\theta_1, \theta_2]\) is not hidden by \(A_i\).

Subcase 2.2. Both \(c_1\) and \(c_2\) are behind \(\text{VIS}(A_i)\). Then there is no intersection between the portions of \(\text{VIS}(A_i)\) and \(\text{VIS}(C_i)\) in \([\theta_1, \theta_2]\), and the portion of \(\text{VIS}(C_i)\) in \([\theta_1, \theta_2]\) (i.e., the portion of \(\text{VIS}(C_i)\) from \(c_1\) to \(c_2\) counterclockwise) is hidden by \(A_i\) (one of \(c_1\) or \(c_2\) is also hidden).
Subcase 2.3. One of $c_{i1}$ or $c_{i2}$ is in front of $VIS(A_i)$ and the other is behind $VIS(A_i)$. Then the portions of $VIS(A_i)$ and $VIS(C_i)$ in $[\theta_l, \theta_r]$ have exactly one intersection. To know which part of $VIS(C_i)$ in $[\theta_l, \theta_r]$ is hidden by $A_i$, we must later on compute that intersection (how to do this will be explained in the next subsection).

Case 3. $\theta_l \neq \theta_r$ and $I(C_i)$ is contained in $I(A_i)$. (Note that in this case, $I(A_i) \cap I(C_i)$ consists of one rather than two intervals.) There are three subcases.

Subcase 3.1. Both $c_{i1}$ and $c_{i2}$ are in front of $VIS(A_i)$. Then $VIS(A_i)$ and $VIS(C_i)$ do not intersect, and no portion of $VIS(C_i)$ is hidden by $A_i$.

Subcase 3.2. Both $c_{i1}$ and $c_{i2}$ are behind $VIS(A_i)$. Then either $VIS(C_i)$ is completely hidden by $A_i$, or $VIS(A_i)$ and $VIS(C_i)$ have two intersections (cf. Lemma 3.2). Lemma 3.4, to be given in the next subsection, will provide a method for distinguishing these two situations, and for computing the two intersections (if any) and hence the portions of $VIS(C_i)$ that are hidden by $A_i$.

Subcase 3.3. One of $c_{i1}$ or $c_{i2}$ is in front of $VIS(A_i)$ and the other is behind $VIS(A_i)$. Then there is exactly one intersection between $VIS(A_i)$ and $VIS(C_i)$ and there can be only one contiguous portion of $VIS(C_i)$ in $[\theta_l, \theta_r]$ that is hidden by $A_i$. The portion of $VIS(C_i)$ hidden by $A_i$ is not known until the intersection between $VIS(A_i)$ and $VIS(C_i)$ is found (how to find it will be explained in the next subsection).

Case 4. $\theta_l \neq \theta_r$ and $I(A_i)$ is contained in $I(C_i)$. (Note that in this case, $I(A_i) \cap I(C_i)$ consists of one rather than two intervals.) There are three subcases 4.1, 4.2, and 4.3. These three subcases are respectively analogous to those for Case 3, with the roles of $A_i$ and $C_i$ being interchanged, and the relevant computational lemma being Lemma 3.5 rather than Lemma 3.4.

All Cases 1 – 4 assumed that $I(A_i) \cap I(C_i) \neq [0, 2\pi)$. Now we turn our attention to the case where $I(A_i) \cap I(C_i) = [0, 2\pi)$:

Case 5. $I(A_i) = I(C_i) = [0, 2\pi)$. In this case there are no beginning and end for $I(A_i) \cap I(C_i)$, hence no $\theta_l$ and $\theta_r$ as in Cases 1 – 4. We distinguish three subcases, based on the outcome of the following. Let $L$ be any line through $q$, let $L^+$ and $L^-$ be the two half-lines on $L$ originating at $q$, and let $\theta^+$ be the polar angle of $L^+$, $\theta^-$
that of $L^-$ (of course each of $[\theta^+, \theta^-]$ and $[\theta^-, \theta^+]$ is of size $\pi$). Let $a_{i1}$ and $a_{i2}$ denote the two points closest to $q$ on $VIS(A_i)$ such that $\theta(a_{i1}) = \theta^+$ and $\theta(a_{i2}) = \theta^-$, and let $c_{i1}$ and $c_{i2}$ denote the two points closest to $q$ on $VIS(C_i)$ such that $\theta(c_{i1}) = \theta^+$ and $\theta(c_{i2}) = \theta^-$. The points $a_{i1}$ and $a_{i2}$ are obtained in $O(\log m)$ time using Corollary 3.1, while the points $c_{i1}$ and $c_{i2}$ are easily obtained from $VIS(C_i)$ in $O(\log m)$ time by a simple one-processor search. We can still use the notation used in Cases 1 - 4 by letting $[\theta_l, \theta_r]$ be one of $[\theta^+, \theta^-]$ or $[\theta^-, \theta^+]$ (the comments below hold in either interval — of course we process each of them separately). The subcases below depend on the relative positions of $a_{i1}, a_{i2}, c_{i1}, c_{i2}$. Each subcase really describes two subproblems, one on each side of line $L$, although our discussion is focusing on only one of these two; so when we refer to “two intersections” in this case, we mean two intersections occurring on the same side of $L$.

Subcase 5.1. Both $c_{i1}$ and $c_{i2}$ are behind $VIS(A_i)$. Then there is either no intersection or two intersections between the portions of $VIS(A_i)$ and $VIS(C_i)$ on the same side of $L$. On each side of $L$, this subcase is handled in the same way as Subcase 3.2 (i.e., using Lemma 3.4, to be given in the next subsection).

Subcase 5.2. Both $c_{i1}$ and $c_{i2}$ are in front of $VIS(A_i)$. Then there is either no intersection or two intersections between the portions of $VIS(A_i)$ and $VIS(C_i)$ on the same side of $L$. On each side of $L$, this subcase is handled in the same way as Subcase 4.2 (i.e., using Lemma 3.5, to be given in the next subsection).

Subcase 5.3. One of $c_{i1}$ or $c_{i2}$ is in front of $VIS(A_i)$ and the other is behind $VIS(A_i)$. Then $VIS(A_i)$ and $VIS(C_i)$ have exactly one intersection on each side of $L$. Each of these is handled exactly like Subcase 3.3.

The above discussion considered the five possible cases and their subcases, and pointed out that each of them can easily be identified. We now turn our attention to the actual computation of the intersections and hidden portions of $VIS(C_i)$ for each of these above subcases.
3.3.4 Computing the Portions of $VIS(C_i)$ Hidden by $A_i$

Let interval $[\theta_i, \theta_r]$ be defined as in the previous subsection. We focus on the portions of $VIS(A_i)$ and $VIS(C_i)$ in $[\theta_i, \theta_r]$ only. Therefore, in this subsection, when we talk about the portions of $VIS(A_i)$ and $VIS(C_i)$, we implicitly mean the portions of $VIS(A_i)$ and $VIS(C_i)$ in $[\theta_i, \theta_r]$, unless otherwise specified.

Now that we have identified which case and subcase hold for the portions of $VIS(A_i)$ and $VIS(C_i)$ in $[\theta_i, \theta_r]$, we turn our attention to the problem of actually computing, for each subcase, the portions of $VIS(C_i)$ that are hidden by $A_i$. Note that, from Lemma 3.1, there are at most two such portions. Doing this for Case 1, Subcase 2.1, and Subcase 3.1 is trivial, since then no portion of $VIS(C_i)$ in $[\theta_i, \theta_r]$ is hidden by $A_i$.

For Subcases 2.2 and 4.1, in which there is no intersection between the portions of $VIS(A_i)$ and $VIS(C_i)$ in $[\theta_i, \theta_r]$, a simple one-processor binary search in $VIS(C_i)$ computes, in $O(\log m)$ time, the portion of $VIS(C_i)$ hidden by $A_i$.

For Subcases 2.3, 3.3, 4.3, and 5.3, in which there is exactly one intersection between the portions of $VIS(A_i)$ and $VIS(C_i)$ in $[\theta_i, \theta_r]$, we must locate that intersection in order to find the portion of $VIS(C_i)$ hidden by $A_i$. In the computation for that intersection, the arrow tags are not needed (however, they will play a crucial role in solving the two-intersection cases discussed later).

In Case B of the algorithm, the problem is much easier (than in Case C), since in this case we know explicitly $VIS(A_i)$ (because $i \in \{1,2\}$ and $A_i = C_2$ if $i = 1$, and is empty if $i = 2$). The (one) intersection is then found by applying a one-processor binary search procedure. This results in the intersection being computed in $O((\log m)^2)$ time (because there are $O(\log m)$ queries in the binary search procedure and each such query requires $O(\log m)$ time). Such an $O((\log m)^2)$ time one-processor search procedure is fine in Case B, since our goal is to perform the "combine" part of the case within this time bound anyway.

In Case C, however, we need to find the intersection in $O(\log m)$ time, and thus we cannot afford to use the one-processor search procedure. However, since $g^3 =$
processors are available for each \( C_i \), we can use Corollary 3.1 (with \( k = O(g^2) \)) to perform a search for the intersection, as follows. Notice that for each \( a \in \{i, \ldots, g\} \), there are at most two connected components of \( VIS(C_a) \) in interval \([\theta_i, \theta_r]\) (i.e., \( I(C_a) \cap [\theta_i, \theta_r] \) can consist of at most two disjoint intervals). A query of this search procedure consists of the following:

(i) in parallel for every \( C_a, a \in \{i, \ldots, g\} \), for each connected component of \( VIS(C_a) \) in \([\theta_i, \theta_r]\), find \( g+1 \) half-lines originating at \( q \) that would partition that connected component of \( VIS(C_a) \) into \( g \) equal pieces (the first and last such half-lines for that component go through its endpoints),

(ii) sort the \( g' = O(g^2) \) half-lines of (i) by their polar angles in counterclockwise order,

(iii) compute the (counterclockwise) sequence of points \( w_1, w_2, \ldots, w_{g'} \) where, for the \( t \)-th half-line, \( w_t \) is the intersection point of that half-line with \( VIS(C_i) \cup VIS(A_i) \) that is closest to \( q \) (and thus may come from either \( VIS(C_i) \) or \( VIS(A_i) \)), and

(iv) check whether the intersection we seek is on one of those \( g' \) half-lines by comparing the intersections of the \( g' \) half-lines with \( VIS(A_i) \) and \( VIS(C_i) \) (if so we stop, if not we proceed recursively to the next query).

The query either finds the intersection, or allows us to restrict the next query of the search to one interval \([\theta(w_j), \theta(w_{j+1})]\), where one of \( \{w_j, w_{j+1}\} \) is on \( VIS(C_i) \) while the other is on \( VIS(A_i) \). Such a query is done in \( O(\log m) \) time using Corollary 3.1 (with \( k = O(g^2) \)). Notice that even though \( VIS(C_a) \) can have two connected components in \([\theta_i, \theta_r]\), after the first query is applied, there can be at most one connected component of \( VIS(C_a) \) left in \([\theta(w_j), \theta(w_{j+1})]\). Our choice of the half-lines for partitioning each connected component of \( VIS(C_a) \) in \([\theta_i, \theta_r]\) guarantees that \([\theta(w_j), \theta(w_{j+1})]\) \( \cap I(C_a) \) is either one contiguous interval or is empty. Clearly, a query either finds the intersection, or restricts the next query of the search for it to one
unique interval $[\theta(w_j), \theta(w_{j+1})]$. The search terminates within $O(\log g m) (= O(1))$ such queries, because for each $a \in \{i, \ldots, g\}$, the portion of $VIS(C_a)$ that lies in $[\theta(w_j), \theta(w_{j+1})]$, if any, has a size that is smaller by a factor of $g$ than the size of the portions of $VIS(C_a)$ in $[\theta_l, \theta_r]$. At the “bottom” of the recursion, either the intersection has been found, or the portion of each $VIS(C_a)$ in $[\theta(w_j), \theta(w_{j+1})]$ has been reduced to a size of $O(1)$ and hence it is a trivial matter to find the intersection among the $O(g)$ surviving segments. Therefore, with $g^3 = (m/d)^{3/4}$ processors, the one-intersection case can be solved in $O(\log m)$ time.

For Subcases 3.2, 4.2, 5.1, and 5.2, in which there are either zero or two intersections between the portions of $VIS(A_i)$ and $VIS(C_i)$ in $[\theta_l, \theta_r]$, we can no longer directly apply the above one-intersection search, because we do not yet know how to use a query’s outcome to constrict the search range for the next query to a single interval $[\theta(w_j), \theta(w_{j+1})]$. The rest of this subsection develops the machinery that enables us to use a query’s outcome in order to constrict the search range to the right interval.

First observe that, in Subcases 3.2 and 5.1, if the portion of $VIS(C_i)$ in $[\theta_l, \theta_r]$ is not completely hidden by $A_i$, then there are exactly two intersections between $VIS(A_i)$ and $VIS(C_i)$, the two intersections both occur at the portions of $VIS(A_i)$ and $VIS(C_i)$ in $[\theta_l, \theta_r]$, and the portion of $VIS(C_i)$ not hidden by $A_i$ is contiguous in $VIS(C_i)$ (followed from Lemma 3.2 and the fact that $I(C_i)$ is contained in $I(A_i)$). Furthermore, the contiguous portion of $VIS(C_i)$ not hidden by $A_i$ is delimited by the two intersections (i.e., the two endpoints of that contiguous portion of $VIS(C_i)$ are at the two intersections, respectively). Thus, any point $p$ of $VIS(C_i)$ not hidden by $A_i$ must lie in that contiguous portion of $VIS(C_i)$. If we could find such a point $p$, then the two intersections would be found by using, for each of them, the one-intersection search procedure (one search would operate on the portion of $VIS(C_i)$ on one side of $p$, i.e., the portion of $VIS(C_i)$ in $[\theta_l, \theta(p)]$, and the other search would operate on the portion of $VIS(C_i)$ on the other side of $p$, i.e., the portion of $VIS(C_i)$ in $[\theta(p), \theta_r]$). This reduces the problem of tackling Subcases 3.2 and 5.1 to that of locating such a point $p$. Lemma 3.4 (to be given below) will help us compute such a point $p$. 
Similarly, in Subcases 4.2 and 5.2, if the portion of VIS(A<sub>i</sub>) in [\theta_l, \theta_r] is not completely hidden by C<sub>i</sub>, then there are exactly two intersections between VIS(A<sub>i</sub>) and VIS(C<sub>i</sub>) in [\theta_l, \theta_r], and the portion of VIS(C<sub>i</sub>) in [\theta_l, \theta_r] hidden by A<sub>i</sub> is contiguous in VIS(C<sub>i</sub>). Thus, any point p of VIS(A<sub>i</sub>) not hidden by C<sub>i</sub> must lie in that contiguous portion of VIS(A<sub>i</sub>) which is delimited by the two intersections. If we could find such a point p, then the two intersections would be found by using, for each of them, the one-intersection search procedure. Lemma 3.5 (to be given below) will help us compute such a point p.

In the next two lemmas, the rank of a point is always with respect to its chain order in the original input chain P.

**Lemma 3.4** Suppose we have two points W<sub>1</sub> and W<sub>k</sub> on VIS(A<sub>i</sub>) ∩ A<sub>i</sub> that are both in front of VIS(C<sub>i</sub>). Suppose that each of I(A<sub>i</sub>) and I(C<sub>i</sub>) contains [\theta(w<sub>1</sub>), \theta(w<sub>k</sub>)]. Let W = (w<sub>1</sub>, w<sub>2</sub>, ..., w<sub>k</sub>) be a sequence of points (not necessarily vertices) on VIS(A<sub>i</sub>) ∩ A<sub>i</sub> that are all in front of VIS(C<sub>i</sub>) and are encountered in that order by a counterclockwise walk from w<sub>1</sub> to w<sub>k</sub> along VIS(A<sub>i</sub>). If the portion of VIS(C<sub>i</sub>) that is in the angular interval [\theta(w<sub>j</sub>), \theta(w<sub>j+1</sub>)] contains a point p that is not hidden by A<sub>i</sub> (i.e., p would be visible if C<sub>i</sub> and A<sub>i</sub> were the only opaque objects), then exactly one of w<sub>j</sub>
or \( w_{j+1} \) has the lowest chain rank among all \( w_1, w_2, \ldots, w_k \). If it is \( w_j \) then its arrow tag is clockwise, and if it is \( w_{j+1} \) then its arrow tag is counterclockwise.

Proof. Let \( u \) and \( v \) be the endpoints of \( C_i \) (Figure 3.5). Without loss of generality, assume \( u <_P v \) (i.e., \( C_i \cap A_i = v \)). First observe that, if \( w_j \) and \( w_{j+1} \) have the same chain rank, then they are on the same segment of \( A_i \) and, since (by hypothesis) \( p \) is in front of that segment, the \( u \)-to-\( v \) path in \( C_i \) cannot reach \( p \) without hiding one of \{\( w_j, w_{j+1} \} \) (recall that \( I(C_i) \) contains \([\theta(w_j), \theta(w_{j+1})]\)). Therefore \( w_j \) and \( w_{j+1} \) have distinct chain ranks, say \( w_j <_P w_{j+1} \) (the case \( w_{j+1} <_P w_j \) is symmetrical, with the roles of "clockwise" and "counterclockwise" being interchanged). The \( v \)-to-\( w_{j+1} \) walk (call it \( Q \)) along \( A_i \) goes through \( w_j \), and we now show that this implies that (i) the first point among \{\( w_1, \ldots, w_k \} \) encountered by the walk \( Q \) is point \( w_j \), and that (ii) the arrow tag of \( w_j \) is clockwise. Suppose (i) is not true, i.e., that \( Q \) encounters some \( w_t \) before encountering \( w_j \). Then the \( w_j \)-to-\( w_{j+1} \) portion of \( Q \) would hide \( w_t \), a contradiction. Suppose (ii) is not true (i.e., the arrow tag of \( w_j \) is counterclockwise). Then \( w_j \) is "isolated" from \( w_{j+1} \) in the sense that the \( w_j \)-to-\( w_{j+1} \) portion of \( Q \) would have to intersect \( C_i \) or hide \( p \) in order to reach \( w_{j+1} \), a contradiction in either case. □

We now discuss the algorithmic implications of the above lemma for handling Subcases 3.2 and 5.1. Assume that we are in Case C. The above lemma implies that in Subcases 3.2 and 5.1, the point \( p \) we seek (if it exists) lies in the unique interval \([\theta(w_j), \theta(w_{j+1})]\) such that exactly one of \( w_j \) or \( w_{j+1} \) (say, \( w_j \)) has the lowest chain rank among all \( w_1, w_2, \ldots, w_k \), and \( w_{j+1} \) is on the side of \( w_j \) which is opposite to the direction of the arrow tag of \( w_j \). Therefore the lemma implies that the point \( p \) we seek (if it exists) has its polar angle \( \theta(p) \) in one interval \([\theta(w_j), \theta(w_{j+1})]\) that is easy to identify in \( O(\log m) \) time so long as we have \( O(gk) \) processors (Corollary 3.1). This suggests using essentially the same search procedure as in the one-intersection case, except that we use Lemma 3.4 to narrow down the search range for the next query (if any) to a single interval \([\theta(w_j), \theta(w_{j+1})]\). At the "bottom" of the recursion, either \( p \) has been found, or the portion of each \( VIS(C_a) \) in \([\theta(w_j), \theta(w_{j+1})]\), \( a \in \{i, \ldots, g\} \),
has been reduced to a size of $O(1)$ and hence it is a trivial matter to find $p$ (if it exists) among the $O(g)$ surviving segments (actually in that second case we get more than the point $p$: we get the (possibly empty) portion of $VIS(C_i)$ in $[\theta_i,\theta_r]$ which is not hidden by $A_i$). If the search terminates without finding such a point $p$, then we know that in interval $[\theta_i,\theta_r]$, no intersection between $VIS(A_i)$ and $VIS(C_i)$ exists and all the portion of $VIS(C_i)$ in that interval is hidden by $A_i$. If such a point $p$ is found, then we have already explained how the problem is reduced to the one-intersection case.

In Case B, we perform a one-processor search procedure, just like the one for the one-intersection case, except that this one-processor search procedure is based on Lemma 3.4 by setting $k = 3$.

Lemma 3.5 Suppose we have two points $w_1$ and $w_k$ on $VIS(C_i) \cap C_i$ that are both in front of $VIS(A_i)$. Suppose that each of $I(C_i)$ and $I(A_i)$ contains $[\theta(w_1), \theta(w_k)]$. Let $W = (w_1, w_2, \ldots, w_k)$ be a sequence of points (not necessarily vertices) on $VIS(C_i) \cap C_i$ that are all in front of $VIS(A_i)$ and are encountered in that order by a counterclockwise walk from $w_1$ to $w_k$ along $VIS(C_i)$. If the portion of $VIS(A_i)$ that is in $[\theta(w_j), \theta(w_{j+1})]$ contains a point $p$ that is not hidden by $C_i$ (i.e., $p$ would be visible if $C_i$ and $A_i$ were the only opaque objects), then exactly one of $w_j$ or $w_{j+1}$ has the highest chain rank among all $w_1, w_2, \ldots, w_k$. If it is $w_j$ then its arrow tag is counterclockwise, and if it is $w_{j+1}$ then its arrow tag is clockwise.

Proof. Similar to that of Lemma 3.4 with the roles of $A_i$ and $C_i$ being interchanged, and hence omitted. \hfill $\Box$

The algorithmic implications of the above lemma for Subcases 4.2 and 5.2 are analogous to those that Lemma 3.4 had for Subcases 3.2 and 5.1, except that the roles of $A_i$ and $C_i$ are interchanged.
3.4 The EREW PRAM Implementation

We had, earlier in the chapter, postponed until this section some of the details of the EREW implementation, although it was at all times clear that the various cases of the algorithm would run in the stronger CREW model. It is easy to see that both Case A and Case B can be trivially implemented on the EREW PRAM since, after the recursive calls return, only one processor is used to compute $VIS(C)$ from the visibility chains of the two subchains of $C$. In Case C, however, the EREW PRAM implementation is complicated by the fact that we use $g^3 = (m/d)^{3/4}$ processors assigned to every $C_i$ in the “combine” part to compute $VIS(C) \cap VIS(C_i)$. Hence there can be many processors searching simultaneously in the tree representing $VIS(C_i)$, which may result in concurrent read accesses by many processors to the same data. How to prevent such read conflicts from happening is the main issue addressed in this section.

Note that, if for all $i$ we were given the portions of $VIS(C_i)$ not hidden by $A_i \cup B_i$, then we already know (from Subsection 3.3.1) how to build the tree representing $VIS(C)$ on the EREW PRAM, within the claimed bounds. Hence, we need only concern ourselves with computing the portions of $VIS(C_i)$ not hidden by $A_i$, without having read conflicts (the computation for $B_i$ is similar).

In computing the portions of $VIS(C_i)$ not hidden by $A_i$, the places where read conflicts could occur are (i) in identifying which case and subcase hold between $VIS(A_i)$ and $VIS(C_i)$ (Subsection 3.3.3), (ii) in reducing the two-intersection cases to the one-intersection cases (Subsection 3.3.4), and (iii) in solving the zero- or one-intersection cases (Subsection 3.3.4). The only nontrivial issue in resolving read conflicts is how to handle the cases where many processors want to simultaneously search in the tree representing $VIS(C_i)$. To avoid read conflicts during such concurrent searching, we use the scheme of [113], which is reviewed in the following lemma.

Lemma 3.6 (Paul, Vishkin, and Wagener [113]) Let $T$ be a 2-3 tree with $m$ leaves, let $a_1, a_2, \ldots, a_k$ be data items that may or may not be stored in (the leaves of) $T$,
and suppose \( k \) processors want to simultaneously access \( T \) to search for the \( a_j \)'s (i.e., processor \( P_j \) wants to search in \( T \) for \( a_j, j = 1, 2, \ldots, k \)). Then in \( O(\log m + \log k) \) time, the \( k \) processors can perform their respective searches without any read conflict.

Proof. See [113]. \( \square \)

Corollary 3.2 Let \( h_i \) be the height of the tree representing VIS(\( C_i \)). Suppose each of \( k \) processors wants to perform a search in the tree for VIS(\( C_i \)). Two types of searches are allowed: the first type is a search for a particular point using its polar angle, and the second is of the type “find the \( t \)-th vertex of VIS(\( C_i \)) starting from vertex \( v \) and moving counterclockwise.” Then the \( k \) processors can perform the \( k \) searches in \( O(\log k + h_i) \) time, without any read conflict.

Proof. The requirements for the concurrent searching scheme of [113] to be applicable are that (i) each internal node of the tree has \( O(1) \) children, and (ii) the \( k \) searches should be “sortable” according to the sorted order of the relative positions of the leaves in the tree. (The scheme of [113] has other requirements, but they are needed only for the concurrent insertions and deletions, not for searching.) Requirement (i) is clearly satisfied in the tree for VIS(\( C_i \)). Requirement (ii) is also satisfied for the searches of type one (simply sort the searches by the polar angles of the \( k \) points). It is satisfied for the searches of the second type, so long as we sort the \( k \) searches using the \( k \) integers \( \alpha(v) + t \), where \( \alpha(v) \) is the rank of the leaf in VIS(\( C_i \)) that stores vertex \( v \) (i.e., \( v \) is stored in the \( \alpha(v) \)-th leaf of the tree for VIS(\( C_i \))). So we must determine the value of the \( \alpha(v) \)'s before actually performing the concurrent searches of type two. This computation of the \( \alpha(v) \)'s is essentially a concurrent searching of the first type, and hence can be done as we process the searches of type one. We process the searches of type two separately, after we are done with the searches of type one. \( \square \)

In Case C, the total amount of processors used is \( g^4 = m/d \) and the height \( h_i \) of the tree representing VIS(\( C_i \)) is \( O(\log |C_i|) = O(\log m) \). Hence the above corollary implies that the concurrent searching in Case C takes \( O(\log m) \) time.
3.5 Applications

This visibility algorithm can be used to solve several problems optimally in parallel. Observe that if point \( q \) is at infinity (say, \( q = (0, +\infty) \)), then the points on the visibility chain of \( P \) are obtained sorted by their \( x \)-coordinates. Once the visibility chain of \( P \) with respect to \( q = (0, +\infty) \) is available, many problems on the simple polygonal chain can be solved optimally in \( O(\log n) \) time using \( O(n/\log n) \) processors. For example, we can optimally compute the convex hull of \( P \) in the above time and processor complexities on the EREW PRAM by first using our visibility algorithm, and then using the parallel algorithm for computing the convex hull of \( O(n) \) points sorted by the \( x \)-coordinates [33] (the details of this convex hull algorithm are given in the next chapter). Also, we can find all the maxima [114] of the vertices of \( P \) by using parallel prefix after the portions of \( P \) visible from \( q = (0, +\infty) \) have been computed. Another immediate consequence of our algorithm is that we can compute the visibility graph [81, 130] of \( P \) in \( O(\log n) \) time using \( O(n^2/\log n) \) EREW PRAM processors, which is worst case optimal. The visibility algorithm will be needed in Chapter 5 to solve the problem of detecting the weak visibility of a simple polygon. The visibility algorithm is likely to find applications in solving other geometric problems involving polygonal chains.
4. SOLVING GEOMETRIC PROBLEMS ON THE EREW PRAM

In this chapter, we present a technique for designing efficient parallel algorithms on the EREW PRAM. This technique is a generalization of the one used in Chapter 3. The technique, when applied to a number of geometric problems on sorted point sets and simple polygons, enables us to obtain optimal algorithms (in \(O(\log n)\) time using \(O(n/\log n)\) processors). The technique consists of a binary tree data structure (to be defined in Section 4.2), several kinds of parallel operations on the binary tree, and a combination of a two-way divide-and-conquer and a quarter-root divide-and-conquer strategies that are used in conjunction with the parallel tree operations. The parallel tree operations include parallel searching, parallel concatenation, and parallel split. These parallel tree operations are performed without read conflicts. This technique is likely to be useful for obtaining efficient EREW PRAM algorithms for problems other than those discussed in this chapter.

All our algorithms in this chapter use the EREW PRAM. We also refer to the CREW PRAM, which is obviously more powerful than the EREW PRAM. Note that the same memory location by multiple processors if the simulation of a CREW PRAM algorithm on an EREW PRAM, using the same number of processors, can increase the time complexity by a logarithmic factor.

We use the technique to solve optimally the following problems: computing the convex hull of \(n\) sorted points in the plane (and hence the dual problem of finding the common intersection of \(n\) half-planes given sorted by their slopes), computing the convex hull of an \(n\)-vertex simple polygon, finding the kernel of an \(n\)-vertex simple polygon, triangulating \(n\) sorted points in the plane, triangulating an \(n\)-vertex monotone polygon or star-shaped polygon, and computing the all dominating neighbors of
Our EREW PRAM algorithms for these problems all take $O(\log n)$ time using $O(n/\log n)$ processors.

The problems of computing the convex hulls of point sets and polygons, computing the kernel of a simple polygon, and triangulating point sets and polygons, are of fundamental importance in computational geometry and have applications in many areas. Considerable work, both in the sequential and parallel computational models, has been done on finding efficient solutions for these problems (see [96, 114] for the sequential algorithms for these problems). For the problem of computing the convex hull of $n$ arbitrary points in the plane, optimal solutions (i.e., $O(\log n)$ time and $O(n)$ processors) have been given on the CREW PRAM [2, 17, 18] and on the EREW PRAM [103]. Optimal CREW PRAM algorithms were also known for the problem of triangulating $n$ arbitrary points in the plane [101, 129] and for the problem of triangulating polygons [67, 135].

The problems we consider in this chapter all have an obvious lower bound of linear work, and sequential linear time algorithms for them have already been known. Some of the solutions can be found in [20, 59, 73, 74, 60, 86, 92, 95, 128, 131]. (Our algorithms are optimal on the EREW PRAM since they all run in $O(\log n)$ time and their $time \times processors$ products match the lower bound of these problems.)

Efficient CREW PRAM algorithms solving the problems that we consider in this chapter have also been discovered. The convex hull problem for a sorted point set can be solved in $O(\log n)$ time using $O(n/\log n)$ CREW PRAM processors [20, 65, 127], and such an algorithm implies (by duality) the same complexity bounds for computing the common intersection of $n$ half-planes whose slopes are given sorted [65]. Note that in the case where the points are already given sorted, the EREW PRAM algorithm of [103] still requires $O(n)$ processors, which is sub-optimal. Our algorithm for the convex hull of a sorted point set can be viewed as another optimal algorithm (i.e., $O(\log n)$ time and $O(n)$ processors) on the EREW PRAM for the case of unsorted input, because we can first obtain a sorted point set with $O(n)$ processors [43] and then use $O(n/\log n)$ processors for the remaining computation. For the case where
the input points are given as a list of vertices on a simple polygon, the convex hull problem can be solved optimally in $O(\log n)$ time using $O(n / \log n)$ CREW PRAM processors [127].

The problem of computing the kernel of a simple polygon has been solved optimally on the CREW PRAM by Cole and Goodrich [44]. Their algorithm is based on the interesting observations which characterize the “curvature” of the polygon boundary.

For sorted point sets and monotone polygons, the triangulation problems can be solved optimally on the CREW PRAM [20, 67, 86, 128]. In fact, Goodrich [67] showed that if the trapezoidal decomposition of a polygon (possibly with holes) has been provided, then a triangulation for that polygon can be done in $O(\log n)$ time using $O(n / \log n)$ CREW PRAM processors.

The problem of computing the all dominating neighbors of $n$ values is defined as follows: Given values $w_1, w_2, \ldots, w_n$, find for each index $i$ the largest (resp., smallest) index $j < i$ (resp., $k > i$) such that $w_j \geq w_i$ (resp., $w_k \geq w_i$). This problem is argued in [20, 86] as being fundamentally important for solving several other problems (not only in computational geometry) on the PRAM. Especially, this problem was used as the basic subproblem for triangulating point sets in the plane and monotone polygons [20, 86, 101, 128]. Optimal algorithms for this problem (in $O(\log n)$ time and $O(n / \log n)$ CREW PRAM processors) have been given in [20, 86, 128].

Most of the constituent parts of our algorithms, namely, the geometric observations, the divide and conquer strategies, the binary tree data structure, and the parallel tree operations (except for the parallel split on the EREW PRAM), have been used before (for example, see [15, 18, 44, 65, 67, 113]). Our contribution is in putting these already available “parts” together in such a way that will enable us to avoid read conflicts that occurred in the previous known CREW PRAM algorithms.

The rest of this chapter is organized as follows. Section 4.1 gives the notation we use in the chapter and outlines the general structure of the algorithms. Section 4.2
discusses the binary tree data structure and the parallel tree operations. Sections 4.3 to 4.5 show how to solve the problems we mentioned above.

4.1 Notation and Basic Algorithm Structure

Let $S$ be a set of $n$ points $p_1, p_2, \ldots, p_n$. $S$ is sorted either by the $x$-coordinates, or by the polar angles of the points with respect to a specified polar point $q \in S$. Let $P$ be a simple polygon defined by the list of vertices $v_1, v_2, \ldots, v_n$, in the order of a clockwise travel along the polygon boundary.

WLOG, we assume that in $S$ (resp., $P$), no two points (resp., vertices) have the same $x$ or $y$-coordinate and no three points (resp., vertices) are collinear. The general situations can be taken care of by slightly modifying our algorithms.

We say two point sets $S'$ and $S''$ are separable if there exists a vertical line such that $S'$ and $S''$ are on the opposite sides of the line. Furthermore, for $k \geq 2$, we say $k$ point sets are separable if there exist $k - 1$ vertical lines such that for any two point sets, at least one of the $k - 1$ vertical lines separates them.

The main procedure of our algorithms has a basic structure, which is the same as the one used in Chapter 3. We outline it as follows.

Input. A set $X$ of size $m$, which is either a sorted point set or the vertex set of a simple polygonal chain, and a positive integer $d$.

Output. The desired output $T(X)$ represented by a tree data structure.

Case A. If $m \leq d$, then compute $T(X)$ with one processor in $O(m)$ time, using a sequential linear-time algorithm.

Case B. If $d < m \leq d^2$, then divide $X$ into two subsets $X_1$ and $X_2$ of equal size and recursively solve the two subproblems in parallel. Then compute $T(X)$ from $T(X_1)$ and $T(X_2)$, in $O(\log m)$ time using one processor.

Case C. If $m > d^2$, then partition $X$ into $g = (m/d)^{1/4}$ subsets $X_1, X_2, \ldots, X_g$ of size $m^{3/4}d^{1/4}$ each. Then, in parallel, recursively solve the $g$ subproblems. Finally,
compute \( T(X) \) from \( T(X_1), T(X_2), \ldots, T(X_g) \), in \( O(\log m) \) time using \( m/d = g^4 \) processors.

Observe that, if we could perform the various cases of the above outline within the claimed bounds, then the algorithm would run in \( O(d + \log m) \) time with \( O(m/d) \) processors since the recurrences of the time and processor complexities are (almost) the same as those in Chapter 3 (the only difference is that Case B in Chapter 3 runs in \( O(\log^2 m) \) time while it takes only \( O(\log m) \) time here). Choosing \( d = \log n \), the above implies a time bound of \( O(\log n) \) and a processor bound of \( O(n/\log n) \). Therefore, a call to the algorithm with input \( (X, \log n), |X| = n \), will compute \( T(X) \) in \( O(\log n) \) time using \( O(n/\log n) \) processors. The rest of the chapter shows how to solve the problems by using algorithms like the one outlined above.

4.2 Rank Trees and Parallel Operations

The algorithms make use of a binary tree data structure. The definition of this tree structure is similar to the one for hull tree, used by Goodrich to store the information of the convex hull for a set of points [65] or the monotone funnel polygons [67]. We call such trees the rank trees because they support efficient operations based on the ranks of the leaves in the trees. A rank tree \( T \) is a binary search tree with a set of points stored at its leaves in some specified order (e.g., by the increasing \( x \)-coordinates or polar angles). WLOG, we assume that the points are sorted by the increasing \( x \)-coordinates. The leaves of \( T \) are doubly linked together. We denote the height of \( T \) (the length of the longest root-to-leaf path in \( T \)) by \( h(T) \). Let \( T_v \) be the subtree of \( T \) rooted at node \( v \) of \( T \). Each internal node \( v \) of \( T \) has four labels: the first stores the number of leaves in \( T_v \), the second stores the point \( p \) at a leaf of \( T_v \) such that \( p \) has the smallest \( x \)-coordinate among the points stored at the leaves of \( T_v \), and the other two respectively store the predecessor and the successor of \( p \) in the sorted point set stored at the leaves of \( T \). In \( O(h(T)) \) time, one processor can search in \( T \) for an \( x \)-coordinate (and hence for a point) or search for the \( i \)-th ranked point stored in \( T \).
(i.e., the \( i \)-th leaf of \( T \) in the left-to-right order) using the first or the second label stored in the internal nodes, respectively.

In the rest of this chapter, all trees are assumed to be rank trees unless otherwise specified.

Our algorithms may need many processors to simultaneously search in such a tree. The next lemma, which is based on the parallel searching scheme of [113], enables us to avoid read conflicts during such parallel search. This lemma is very similar to Lemma 3.2 in Chapter 3.

Lemma 4.1 Given a tree \( T \), suppose each of \( k \) processors wants to perform a search in \( T \). Two types of searches are allowed: the first type is a search for a particular point using its \( x \)-coordinate, and the second is of the type “find the \( t \)-th leaf of \( T \) starting from leaf \( l \) and moving to the right.” Then the \( k \) processors can perform their searches in \( O(\log k + h(T)) \) time, without any read conflict.

Proof. Same as that for Corollary 3.2 of Chapter 3, and hence omitted. \( \square \)

The next two lemmas are for the parallel concatenation and split.

Lemma 4.2 (Goodrich [65, 67]) Let \( S_1, S_2, \ldots, S_k \) be subsets of a point set \( S' \) separated by \( k - 1 \) vertical lines, and let the trees \( T(S_1), T(S_2), \ldots, T(S_k) \) for the subsets be given. Then tree \( T(S') \) for \( S' \) can be built in \( O(\log k + h) \) time using \( k \) EREW PRAM processors, where \( h \) is the maximum of the \( h(T(S_i)) \)'s. Also, \( h(T(S')) = O(h + \log k) \).

Proof. Same as the proof of Lemma 4.3 in [67] and hence omitted. \( \square \)

Lemma 4.3 Given a tree \( T \) and a list \( (x_1, x_2, \ldots, x_k) \) of the \( x \)-coordinate values, suppose \( k \) EREW PRAM processors want to split \( T \) into \( k + 1 \) trees \( T_0, T_1, \ldots, T_k \) such that all the points in \( T_i \) have their \( x \)-coordinates within interval \([x_i, x_{i+1}] \), for \( i \in \{0, 1, \ldots, k\} \) (\( x_0 = -\infty \) and \( x_{k+1} = +\infty \)). Then the parallel split can be done in \( O(\log k + h(T)) \) time.
Proof. WLOG, we assume that the list \((x_1, x_2, \ldots, x_k)\) is given sorted (otherwise, we have enough processors to do the sorting in \(O(\log k)\) time [43]). Let processor \(P_i\) have value \(x_i, i = 1, 2, \ldots, k\). Each \(P_i\) does the split operation in the same way as shown in Lemma 3.2 of [65]. That is, it searches for \(x_i\) in a tree and, when going down the tree following a root-to-leaf path, it makes two copies of the root-to-leaf path, whose nodes have the appropriately modified data from the original root-to-leaf path (actually, the original path is replaced by one copy); after it reaches the leaf of the path, it retraces the two paths which it just created, to update the labels of the nodes on the paths (see Lemma 3.2 in [65] for more details).

To avoid read conflicts, we use a procedure consisting of \(O(\log k)\) stages. Let \(T_{1,k} = T\). Before a tree \(T_{a,b}\) is split \((a \leq b)\), there is a group of processors \(P_a, P_{a+1}, \ldots, P_b\) associated with it. The following is the general step. Suppose (the root of) \(T_{a,b}\) is available in stage \(i\) but not in stage \(i - 1\). If \(a = b\), then \(P_a\) splits \(T_{a,b}\) at its root; otherwise, \(P_{\lceil(a+b)/2\rceil}\) splits \(T_{a,b}\) at its root (by making two copies) and makes the roots of \(T_{a,\lceil(a+b)/2\rceil - 1}\) and \(T_{\lceil(a+b)/2\rceil + 1,b}\) available for stage \(i + 1\) (no split is done on \(T_{\lceil(a+b)/2\rceil + 1,b}\) if \([\lceil(a+b)/2\rceil + 1 > b\)\). The processors stop when they reach the leaves on the root-to-leaf paths they follow. After all processors stop at the leaves of the \(k + 1\) trees so obtained, we let each processor retrace the leaf-to-root paths in each tree which it just created and update the labels of the nodes on the paths just as was done in [65].

The correctness of this parallel split procedure is guaranteed by the facts that \((x_1, x_2, \ldots, x_k)\) is sorted and the split is based on searching the \(x_i\)'s. No read conflict can occur in the procedure because although in the searching, different processors may follow the same root-to-leaf path in \(T\), the processors, when doing the split, actually use different copies of the path, and such copies would have been created in the previous stages of the procedure. The time complexity of the procedure is clearly \(O(\log k + h(T))\). \(\Box\)
4.3 Computing the Convex Hull of Sorted Points

This section discusses the parallel algorithm for computing the convex hull of a point set in the plane sorted by the increasing x-coordinates. Recall that the planar convex hull problem is that of computing the smallest convex region in the plane that contains a given set of points (see Figure 4.1 for example). Let \( S = \{p_1, p_2, \ldots, p_n\} \) be a set of sorted points. We denote the convex hull of \( S \) by \( CH(S) \). Points \( p_1 \) and \( p_n \) are both vertices of \( CH(S) \) because \( p_1 \) and \( p_n \), respectively, have the smallest and the largest x-coordinates among the points in \( S \). Traveling along \( CH(S) \) from \( p_1 \) to \( p_n \) clockwise, the portion of \( CH(S) \) so visited is called the upper hull of \( S \), denoted by \( UH(S) \). Similarly, the portion of \( CH(S) \) visited by traveling along \( CH(S) \) from \( p_n \) to \( p_1 \) clockwise is called the lower hull of \( S \), denoted by \( LH(S) \). Due to the similarity in the computation of \( UH(S) \) and \( LH(S) \), we only discuss the algorithm for \( UH(S) \). For two upper hulls \( UH(S') \) and \( UH(S'') \), where \( S' \) and \( S'' \) are separable point sets, the upper common tangent between \( UH(S') \) and \( UH(S'') \) is the common tangent of \( UH(S') \) and \( UH(S'') \) such that both \( UH(S') \) and \( UH(S'') \) are below it. The lower common tangent for two lower hulls is defined similarly. In the rest of this section, we just say the "common tangent" to mean the "upper common tangent."
The following two known results are useful.

Lemma 4.4 (Goodrich [65, 67]) Let two upper hulls $UH(S_1)$ and $UH(S_2)$ be stored in trees $T_1$ and $T_2$, respectively, where $S_1$ and $S_2$ are two separable point sets. Then in $O(h(T_1) + h(T_2))$ time, one processor can find the common tangent between $UH(S_1)$ and $UH(S_2)$.

Proof. See Lemma 3.1 in [65].

Lemma 4.5 (Atallah and Goodrich [18]) Let $S_1$ and $S_2$ be two separable point sets with both $|S_1|$ and $|S_2|$ being $O(m)$, and let $UH(S_1)$ and $UH(S_2)$ be their upper hulls stored in two arrays, respectively. Then the common tangent between $UH(S_1)$ and $UH(S_2)$ can be computed in $O(c^2) = O(1)$ time using $m^{1/c}$ CREW PRAM processors, where $c$ is a positive constant.

Proof. See Theorem 1 and Algorithm A in [18].

We immediately have the following corollary.

Corollary 4.1 Let $S_1$ and $S_2$ be two separable point sets with both $|S_1|$ and $|S_2|$ being $O(m)$, and let $T(S_1)$ and $T(S_2)$ be two trees storing the upper hulls $UH(S_1)$ and $UH(S_2)$, respectively. Then the common tangent between $UH(S_1)$ and $UH(S_2)$ can be computed in $O(\log m + h)$ time using $m^{1/c}$ EREW PRAM processors, where $c$ is a positive constant and $h$ is the maximum of $h(T(S_1))$ and $h(T(S_2))$.

Proof. Recall that Algorithm A in [18] partitions the two arrays for the two upper hulls $UH(S_1)$ and $UH(S_2)$ into subarrays, then it finds in which subarrays the common tangent lies, and it then recursively solves the problem in the (two) subarrays so found. We simulate Algorithm A using $m^{1/c}$ EREW PRAM processors. Note that Algorithm A is on the CREW PRAM and it requires $O(1)$ time (given $c$ a constant). Since now $UH(S_1)$ and $UH(S_2)$ are stored in trees (instead of arrays), each access to a leaf of a tree requires $O(h)$ time and one processor. Parallel searching for the points at the leaves of a tree (without read conflicts) can be done in $O(\log m + h)$ time using the available processors by Lemma 4.1. Each time an array (or a subarray) is partitioned
Algorithm A, we can achieve the same effect by partitioning the leaves of the relevant tree by using the ranks of the leaves. Such a partition can also be done in $O(\log m + h)$ time by doing parallel searching in the tree by Lemma 4.1. The other steps of Algorithm A can be easily simulated in $O(\log m)$ time on the EREW PRAM.

Now we show the algorithm for computing $UH(S)$. We refer to the cases of the outline in Section 4.1. In Case A, we call the linear time algorithm in [73] to compute the upper hull. In Case B, we use Lemma 4.4 to compute the common tangent between the two upper hulls returned from the two recursive calls, then we split the tree for each upper hull to remove the portion of that upper hull (if any) that is under the common tangent. The portions of the two upper hulls that remain form the upper hull that we seek in this case (by doing a simple concatenation). Since we use only one processor in this case, no read conflict occurs. We perform Case C as follows. Given $g$ subsets $S_1, S_2, \ldots, S_g$ of a point set $S'$, separated by $g - 1$ vertical lines, where $|S'| = m$ and $g = (m/d)^{1/4}$, and given the trees $T(S_1), T(S_2), \ldots, T(S_g)$ representing their upper hulls, respectively, we compute the common tangent $C_{ij}$ for each pair of $UH(S_i)$ and $UH(S_j)$, $1 \leq i < j \leq g$. Recall that we have $g^4 = m/d$ processors to do so, and $|S_k| = m^{3/4}d^{1/4}$ for each $k$. Every $C_{ij}$ is obtained in $O(\log m)$ time using $g^2 = (m/d)^{1/2}$ processors by Corollary 4.1 (it has been shown in Chapter 3 that $h(T(S_k))$ is $O(\log m)$ for every $k$). Note that, in this case, we do not use the procedure for Lemma 4.4 to compute the $C_{ij}$'s. This is because the procedure for Lemma 4.4 searches for the points, on two upper hulls, where the common tangent for the two upper hulls lies, and before the search starts, we do not know at all which points we are getting to. In Case C, each $T(S_k)$ is involved in the computation of $O(g)$ common tangents. If we used the procedure for Lemma 4.4, we would have read conflicts from the $O(g)$ simultaneous searches in $T(S_k)$, since we could not prearrange the $O(g)$ processors doing the searches (as was done in the scheme of [113]) in order to avoid read conflicts. From the $O(g^2)$ common tangents (the $C_{ij}$'s), we can find the
portions of the $UH(S_k)$'s that form $UH(S')$, by doing parallel prefix [88, 89] (see [65] for the details on how this is done). The tree $T(S')$ is then built using Lemma 4.2.

4.4 Triangulating a Trapezoidally Decomposed Polygon

This section deals with the parallel algorithm for triangulating an $n$-vertex polygon $P$ (possibly with holes) when given a trapezoidal decomposition of $P$. Recall that the polygon triangulation problem is that of partitioning $P$ into $n-2$ triangles by augmenting $P$ with $n-3$ non-intersecting diagonal edges (see Figure 4.2 for example). Goodrich [67] showed how to triangulate a polygon in $O(\log n)$ time using $O(n/\log n)$ CREW PRAM processors, provided that the trapezoidal decomposition of the polygon has been given (the trapezoidal decomposition of $P$ can be done in $O(\log n)$ time using $O(n)$ CREW PRAM processors [16]). Here we assume that the same input as in [67] is given. We will basically follow the same computational steps as in [67] (hence the reader is referred to [67] for more details of the algorithm). We only show how to use a quarter-root divide and conquer strategy and the parallel tree operations to perform various operations of [67] in the required time and processor complexities without having read conflicts.
There are three phases in [67]. The reader is referred to [67] for the definitions used here. There is no read conflict in Phase One, whose goal is to construct the set of one-sided monotone polygons which decomposes $P$ (by using parallel prefix [88, 89] and list ranking [46]). There is also no read conflict in Phase Three, whose goal is to triangulate the set of monotone funnel polygons resulting from Phase Two (by using parallel prefix and parallel merging [26, 79]). Hence we only need to concern ourselves with Phase Two, whose goal is to decompose every one-sided monotone polygon (from Phase One) into a set of monotone funnel polygons.

The difficult computation in Phase Two is to decompose every one-sided monotone polygon whose size is larger than $\log n$ into a set of monotone funnel polygons. Given a monotone chain $C$ (from the one-sided monotone polygon), $|C| = m$, the procedure in Phase Three for this computation first partitions chain $C$ into $\alpha$ subchains of equal size, and recursively solves the $\alpha$ subproblems in parallel. Then, from the results for the $\alpha$ subproblems, it computes the bases of the monotone funnel polygons that consist of the decomposition of the one-sided monotone polygon. Finally, it computes the left and right boundaries of the monotone funnel polygons and the lower hull of $C$. This procedure, although being quite complicated, essentially consists of the following operations: parallel prefix, sorting $O(\alpha)$ values, computing $O(\alpha^2)$ lower common tangents (among the lower hulls of the $\alpha$ subchains represented by $\alpha$ trees, as returned by the recursive calls), parallel splits on the $\alpha$ trees (into $O(\alpha)$ trees), and parallel concatenations of $O(\alpha)$ trees (to construct new trees representing the left and right boundaries of the monotone funnel polygons as well as the lower hull of chain $C$). Recall that our algorithm is based on the outline given in Section 4.1. Case A and Case B of this algorithm can be easily handled in the required complexity bounds. In Case C, we have $g = (m/d)^{1/4}$ subproblems and $g^4$ processors. The parallel prefix and sorting [43] can be done by using the available processors. The $O(g^2)$ lower common tangents are computed in a way similar to the convex hull algorithm of Section 4.3 (i.e., by Corollary 4.1). The parallel splits are done by Lemma 4.3 and the parallel concatenations are done by Lemma 4.2. None of these operations introduces read
conflicts and all of them can be performed in $O(\log m)$ time using $g^d$ EREW PRAM processors.

Note that given a simple polygon, testing whether the polygon is monotone can be done optimally in $O(\log n)$ time using $O(n/\log n)$ EREW PRAM processors [36]. Hence the triangulation of a simple polygon, if it happens to be monotone, can be done optimally in $O(\log n)$ time using $O(n/\log n)$ EREW PRAM processors.

4.5 Other Geometric Algorithms

The algorithms described in Sections 4.3 and 4.4 enable us to obtain optimal EREW PRAM algorithms for other geometric problems. All the algorithms in this section take $O(\log n)$ time using $(n/\log n)$ EREW PRAM processors.

If a set of points in the plane is given sorted by the polar angles of the points with respect to a polar point $q$, the convex hull algorithm in Section 4.3 can be slightly modified to compute the convex hull of this point set. Using the convex hull algorithm in Section 4.3, we compute the convex hull of an $n$-vertex simple polygon $P$ as follows: apply the visibility algorithm in Chapter 3 (which runs in $O(\log n)$ time using $O(n/\log n)$ EREW PRAM processors) to $P$ to obtain a point set sorted by the $x$-coordinates (the vertices of $P$ that are visible from the point $q = (0, +\infty)$), then apply the convex hull algorithm in Section 4.3 to the visible vertex set to find the upper hull of $P$. Using the geometric duality transformation [114], an immediate result from the convex hull algorithm in Section 4.3 is an optimal EREW PRAM algorithm for the problem of computing the common intersection of $n$ half-planes given sorted by their slopes. The kernel of a simple polygon can be computed optimally by using the convex hull algorithm in Section 4.3 as a subroutine in the algorithm of [44] (parallel prefix and parallel merging [26, 79] are also used in [44]).

The triangulation algorithm in Section 4.4 implies an optimal EREW PRAM solution for triangulating a monotone polygon $P$, since a parallel merging will decompose $P$ into a set of one-sided monotone polygons (triangulating one-sided monotone polygons is done in Section 4.4). Using the algorithm for computing the kernel of a simple
polygon, we can check whether a simple polygon $P$ is star-shaped or not. If it is, then the kernel of $P$ is nonempty. Let $r$ be the ray starting from vertex $v_1$ of $P$ and going through a point $q$ in the kernel of $P$. WLOG, we assume that $r$ does not contain any edge of $P$. Let $r$ intersect the boundary of $P$ at a point $p \neq v_1$. If $p$ is at some vertex of $P$, then we partition the boundary of $P$ into twopolygonal chains $C'$ and $C''$ by $r$. Otherwise, let $p$ be on edge $e$ with endpoints $v_i$ and $v_{i+1}$, and let $C'$ be the polygonal chain consisting of vertices $v_1, v_2, \ldots, v_i$, and $C''$ the polygonal chain consisting of $v_{i+1}, v_{i+2}, \ldots, v_n, v_1$. Clearly, $C'$ and $C''$ are both star-shaped (i.e., they are all visible from $q$). A triangulation for each of $C'$ and $C''$ can be done in a way similar to the algorithm in Section 4.4 for triangulating a one-sided monotone polygon, since the vertices of $P$ are sorted along the boundary of $P$ by their polar angles with respect to point $q$ (in this case, $q$ plays the role for $C'$ and $C''$ same as the distinguished edge of a one-sided monotone polygon does for the polygon). The triangulation of $C'$ and $C''$ also gives a monotone funnel polygon inside $P$ (with base $e$, the right boundary from the triangulation of $C'$, and the left boundary from the triangulation of $C''$). This monotone funnel polygon is the only portion of $P$ that has not yet been triangulated. A triangulation of $P$ can be completed by doing a parallel merging (see Phase Three in [67]). Hence we optimally solve the problem of triangulating a star-shaped polygon.

Triangulating a point set in the plane sorted by the $x$-coordinates can be reduced to that of triangulating a set of one-sided monotone polygons, as follows. We first construct a monotone chain with the sorted points being the vertices of the chain. Then we compute the convex hull of the monotone chain (by using the convex hull algorithm of Section 4.3). The convex hull and the monotone chain, together, partition the region bounded by the convex hull into a set of one-sided monotone polygons. Triangulating a point set in the plane sorted by the polar angles with respect to a polar point $q$ can be reduced to that of triangulating the interior and the exterior of a star-shaped polygon, which can be done optimally in a way similar to the triangulation algorithm for a star-shaped polygon.
Given $n$ values $w_1, w_2, \ldots, w_n$, the *all dominating neighbors problem*, in fact, can be viewed as a rectilinear version of triangulating a monotone polygonal chain. That is, we reduce the $n$ values into $n$ points $(1, w_1), (2, w_2), \ldots, (n, w_n)$, and let a rectilinear monotone polygonal chain $C$ have the $n$ points as part of its vertices. The chain $C$ consists of only vertical and horizontal line segments. All the "common tangents" we compute, and all the "diagonals" we add to the "triangulation," are horizontal line segments. Our algorithm in Section 4.4 can be modified to solve this problem.
5. DETECTING WEAK VISIBILITY OF A SIMPLE POLYGON

Weak visibility problems are often concerned with visibility with respect to “observers” that have the shape of line segments. An important class of weak visibility problems deals with the case where the opaque objects are the boundaries of simple polygons. For a point $p$ in a polygon and a line segment $s$, $p$ is weakly visible from $s$ if $p$ is visible from some point on $s$. An example of such problems is that of computing the region inside a polygon that is weakly visible from a segment. For this problem, many sequential algorithms [32, 55, 77, 80, 94, 125] and a parallel algorithm [72] have been discovered. For more examples of the weak visibility problems on simple polygons, see [19, 24, 25, 35, 38, 61, 85, 121, 126].

In this chapter, we consider the problem of detecting the weak visibility of a simple polygon. An $n$-vertex simple polygon $P$ is weakly visible if there exists an edge $e$ of $P$ such that every point in $P$ is weakly visible from $e$ ($e$ is called a weakly visible edge of $P$). The problem of detecting the weak visibility of $P$ is that of finding whether $P$ is weakly visible and (if it is) identifying all weakly visible edges of $P$. In Figure 5.1, the weakly visible edges of polygon $P$ are edges $e_1$, $e_2$, and $e_4$. Note that this problem is a natural generalization of the well-known problem of computing the kernel of a simple polygon [95]. (Recall that a point is in the kernel of a polygon iff the whole polygon is visible from that point, and that a polygon with a nonempty kernel is called a star-shaped polygon [114].)

Avis and Toussaint [19] first consider the problem of detecting the weak visibility of a simple polygon. They present a sequential linear time algorithm for the following case: check whether a polygon $P$ is weakly visible from a specified edge $e$ of $P$. Another sequential linear time algorithm for this case was recently given in [61]. Using the algorithms in [19, 61], the problem of detecting the weak visibility of $P$
Figure 5.1 The weakly visible edges of $P$ are $e_1$, $e_2$, and $e_4$.

can be trivially solved in $O(n^2)$ time (by checking the weak visibility of $P$ from each edge separately), but Sack and Suri [121] succeeded in finding a linear time algorithm for this problem. Our interest here is to solve this problem in parallel. The parallel computational model we use in this chapter is the CREW PRAM.

Based on the observations of Sack and Suri [121], a suboptimal parallel algorithm can be easily obtained by using the recent result of Goodrich et al. [72] on building a data structure that supports ray-shooting queries [32]. This algorithm first preprocesses $P$ and builds the data structure in $O(\log n)$ time using $O(n)$ CREW PRAM processors [72], and then does $O(n)$ ray-shooting queries by using the data structure. The algorithm takes in total $O(\log n)$ time and $O(n \log n)$ work (recall that the work of a parallel algorithm is the total number of operations performed by the algorithm). Obviously, the work complexity of this algorithm is a factor of $\log n$ away from optimality. The sequential algorithm in [121] manages to avoid doing ray-shooting queries, but that method seems to be inherently sequential.

Our method for obtaining an optimal parallel algorithm is very different from the above approaches. We give geometric insights and parallel techniques which enable us to use the divide-and-conquer strategies and to avoid the difficulty of doing ray-shooting queries. Our algorithm runs in $O(\log n)$ time using $O(n/\log n)$ processors,
and is thus optimal. We also use this algorithm to solve optimally, in parallel, several other problems on weakly visible polygons (such as computing shortest paths, triangulation, solving the one-cruising-guard problem [38], checking the weak external visibility, etc.); these parallel solutions all take $O(\log n)$ time using $O(n/\log n)$ processors and avoid triangulating the polygon. The known parallel algorithms for computing shortest paths inside a simple polygon usually use a preprocessing step of triangulating the polygon (for example, see [57, 72]). The best known parallel algorithms for triangulating a simple polygon require either $O(n)$ processors on the CREW PRAM [67, 135] or $O(n/\log n)$ processors on the more powerful CRCW PRAM [69], and $O(\log n)$ time. The geometric insights we present could be useful in solving other geometric problems.

There are two major subproblems solved in our weak visibility algorithm: (1) identifying the weakly visible edges for a star-shaped polygon whose kernel contains a convex vertex, and (2) checking whether a polygon is weakly visible from a specified edge (i.e., the case solved by [19, 61]). The solutions to these two subproblems could be interesting in their own right. Solving the problem of detecting the weak visibility of a simple polygon is reduced to solving these two subproblems. The reduction is based on the idea used in [121], but our reduction procedure is very different from [121].

The rest of the chapter consists of 6 sections. Section 5.1 gives some notation and preliminary results related to the weak visibility of a simple polygon. Section 5.2 discusses several geometric and computational observations needed by the algorithm. Sections 5.3 and 5.4 describe in detail the algorithms for solving the two subproblems mentioned above, respectively. Section 5.5 presents the reduction from the problem of detecting the weak visibility of a simple polygon to the two subproblems. Section 5.6 gives several applications of the weak visibility algorithm.
5.1 Preliminaries

Suppose that an n-vertex simple polygon $P$ is specified by a sequence $(v_1, v_2, \ldots, v_n)$ of its vertices, in the order in which they are visited by a counterclockwise walk along the boundary of $P$ starting from vertex $v_1$. The edge of $P$ joining $v_i$ and $v_{i+1}$ is denoted by $e_i = v_i v_{i+1} (= v_{i+1} v_i)$, with the convention that $v_{n+1} = v_1$. The boundary of $P$ is denoted by $bd(P)$.

WLOG, we assume that no edge of $P$ is vertical and no three consecutive vertices of $P$ are collinear.

Vertex $v_i$ is convex if the interior angle of $P$ at $v_i$ is $< \pi$. Edge $e_i$ is convex if both $v_i$ and $v_{i+1}$ are convex. For a vertex $v_i$, if $v_{i+1}$ (resp., $v_{i-1}$) is nonconvex, then let ray $r_i^+$ (resp., $r_i^-$) be associated with $v_i$, where $r_i^+$ starts at $v_i$ and contains $e_i$ (resp., $e_{i-1}$). The set of all such $r_i^+$'s (resp., $r_i^-$'s) is denoted by $Ray^+(P)$ (resp., $Ray^-(P)$). Let $h_i^+$ (resp., $h_i^-$) be the point at which $r_i^+$ (resp., $r_i^-$) first hits $bd(P) - e_i$ (resp., $bd(P) - e_{i-1}$) (i.e., $v_i$ is closer to $h_i^+$ (resp., $h_i^-$) than to any other point in $r_i^+ \cap (bd(P) - e_i)$ (resp., $r_i^- \cap (bd(P) - e_{i-1})$)). We henceforth call $h_i^+$ (resp., $h_i^-$) the first-hit point of $r_i^+$ (resp., $r_i^-$). Let $h_i^+$ (resp., $h_i^-$) be on $e_j - v_j$ (resp., $e_k - v_{k+1}$) for some $j$ (resp., $k$). Then we call the consecutive edges that are on the portion of
The set of counterclockwise (resp., clockwise) hidden edges of $v_i$ is denoted by $HE^+_i$ (resp., $HE^-_i$) (see Figure 5.2 for example). It is possible for $HE^+_i$ (resp., $HE^-_i$) to be empty. The union of all the $HE^+_i$'s (resp., $HE^-_i$'s) is denoted by $HE^+(P)$ (resp., $HE^-(P)$).

It is shown in [121] that the set of weakly visible edges of $P$, denoted by $WVE(P)$, is equal to $bd(P) - (HE^+(P) \cup HE^-(P))$ (see Theorem 1 of [121]). For convenience, we call the edges in $HE^+(P) \cup HE^-(P)$ the bad edges of $P$. If $HE^+(P)$ and $HE^-(P)$ were available, then $WVE(P)$ could be obtained in the desired complexity bounds (we will show how this is done in Subsection 5.3.3). The main difficulty, therefore, is in computing $HE^+(P)$ and $HE^-(P)$. WLOG, we will show the computation for $HE^+(P)$ (the computation for $HE^-(P)$ is similar). The following lemma is useful in computing $HE^+(P)$ and $HE^-(P)$.

Lemma 5.1 (Sack and Suri [121]) Suppose that polygon $P$ is weakly visible from edge $e_n$ and $e_n$ is convex. Then for every $i$, $i = 1, 2, \ldots, n$, the following holds: (i) if $h_i^+$ exists, then $h_i^+$ is the first point on $bd(P) - e_i$ collinear with $e_i$ encountered in the counterclockwise walk along $bd(P)$ starting from $v_{i+1}$, and (ii) if $h_i^-$ exists, then $h_i^-$ is the first point on $bd(P) - e_{i-1}$ collinear with $e_{i-1}$ encountered in the clockwise walk along $bd(P)$ starting from $v_{i-1}$.

Proof. See Lemma 3 of [121].

A point $p$ is represented by its $x$-coordinate and $y$-coordinate, denoted by $x(p)$ and $y(p)$, respectively. For a line segment $s$ (resp., a ray $r$), the line containing $s$ (resp., $r$) is denoted by $l(s)$ (resp., $l(r)$). The slope of a line $l$ (resp., a segment $s$, a ray $r$) is denoted by $slope(l)$ (resp., $slope(s)$, $slope(r)$).

The chain on $bd(P)$ from $v_i$ counterclockwise to $v_j$, $i \neq j$, is denoted by $bd_{ij}$. The size of a chain $C$ is the number of line segments on $C$, denoted by $|C|$. For three non-collinear points $p$, $q$, and $r$, we say that the directed chain from $p$ to $q$ to $r$ makes a left (resp., right) turn if $x(r)(y(p) - y(q)) + y(r)(x(q) - x(p)) + x(p)y(q) - x(q)y(p) > 0$ (resp., $< 0$). For a directed simple chain $C = (p_1, p_2, \ldots, p_k)$, $k \geq 3$, $C$ is said to
make only left (resp., right) turns iff every subchain of the form $(p_{i-1}, p_i, p_{i+1})$ makes a left (resp., right) turn, $1 < i < k$.

If polygon $P$ is weakly visible from edge $e_n$ and $e_n$ is convex, then for $1 \leq i \leq j \leq n$, the (directed) shortest path from $v_i$ to $v_j$ inside $P$ goes through only the vertices on $bd_{ij}$, and the shortest path makes only right turns (this fact is shown in [19, 61]). Hence, we call such a shortest path the internal convex path of $bd_{ij}$, and denote it by $ICP(bd_{ij})$.

Let $l$ be a non-vertical line. We say a point $p$ is above (resp., below) $l$ iff the vertical line passing $p$ intersects $l$ at a point $q$ such that $y(q) < y(p)$ (resp., $y(q) > y(p)$). A segment $s$ is said to be (properly) above (resp., below) $l$ iff every point of $s$ is above (resp., below) $l$. The upper (resp., lower) half-plane of $l$ is the half-plane bounded by $l$ whose interior points are all above (resp., below) $l$. We denote the upper (resp., lower) half-plane of $l$ by $up(l)$ (resp., $lp(l)$). A left (resp., right) half-plane of a ray $r$ is the half-plane whose boundary contains $r$ and which is to the left (resp., right) of $r$. For a set $L$ of lines, we use $UPCI(L)$ (resp., $LPCI(L)$) to denote the common intersection of the upper (resp., lower) half-planes for the lines in $L$.

Internal convex paths and common intersections of half-planes play important roles in our algorithm. We will represent internal convex paths and common intersections of half-planes by using a data structure called the hull tree [65, 67] or rank tree [33]. This data structure supports efficient implementation for the parallel operations of search, concatenation, and split (see [65, 67] or Chapter 4 for the details). We let $height(T)$ denote the height of a tree $T$.

5.2 Some Useful Observations

This section gives some useful geometric observations and develops the computational machinery needed by the algorithm.

The following type of tests will be frequently done by the algorithm: given a set $L$ of lines and a line segment $s$, find (i) whether there is a line $l \in L$ such that $s$ is below
or (ii) whether \( s \) is above all the lines in \( L \). We call such a test a lines-vs-segment test and denote it by \( \text{Test}(L, s) \).

(Note: Another type of tests which is also needed by the algorithm is: (i') whether there is a line \( l \in L \) such that \( s \) is above \( l \), or (ii') whether \( s \) is below all the lines in \( L \). The tests for (i') and (ii') are handled similarly to those for (i) and (ii), with the exception that (i) and (ii) (resp., (i') and (ii')) resort to the common intersections of the upper (resp., lower) half-planes of the relevant lines. Hence we omit the discussion for (i') and (ii').)

If \( L \) is fixed and one wants to test for many different segments \( s \), then doing every \( \text{Test}(L, s) \) by using a brute force method, which checks segment \( s \) against every line in \( L \), can be inefficient (each test requires \( O(|L|) \) work and \( O(\log |L|) \) time). In our algorithm, we would like to achieve \( O(\log |L|) \) time and \( O(|L|^\alpha) \) work for every \( \text{Test}(L, s) \), where \( \alpha \) is some constant, \( 0 < \alpha < 1 \). Our method for processing the tests makes use of the common intersections of the upper half-planes of relevant lines. It is obvious that \( s \) is above all the lines in \( L \) iff \( s \) is properly contained in \( \text{UPCI}(L) \). However, it is not necessarily true that if \( s \) does not intersect \( \text{UPCI}(L) \), then \( s \) is below a line in \( L \). Our solution to the tests is based on the following observation.

**Lemma 5.2** Given a non-vertical line segment \( s \) and a set \( L \) of lines, if \( L \) is partitioned into two subsets \( L' \) and \( L'' \) such that \( L' \) (resp., \( L'' \)) contains the lines of \( L \) whose slopes are all \( \geq \text{slope}(s) \) (resp., \( < \text{slope}(s) \)), then the following is true: (i) \( s \) is below a line in \( L \) iff \( s \) does not intersect either \( \text{UPCI}(L') \) or \( \text{UPCI}(L'') \), and (ii) \( s \) is above all the lines in \( L \) iff \( s \) is properly contained in both \( \text{UPCI}(L') \) and \( \text{UPCI}(L'') \).

**Proof.** We only prove for \( L' \) (the proof for \( L'' \) is symmetric). (ii) is clearly true, since \( s \) is above a line \( l \) iff \( s \) is properly contained in the upper half-plane \( \text{up}(l) \). To prove (i), let \( u_1 \) be the endpoint of \( s \) with the smallest \( x \)-coordinate. For a line \( l \in L' \) with \( \text{slope}(l) > \text{slope}(s) \), \( s \) is below \( l \) iff \( l \) intersects line \( l(s) \) at a point \( p \) such that \( x(p) < x(u_1) \). Let \( q \) be the leftmost intersection point between \( l(s) \) and the lines in \( L' \) whose slopes are \( > \text{slope}(s) \). Suppose \( q \) belongs to line \( l' \in L' \). Also, let \( l^* \) be the line in \( L' \) such that \( \text{slope}(l^*) = \text{slope}(s) \) and no other line in \( L' \) with the same slope as \( \text{slope}(s) \)
is above \(l^*\) (if such an \(l^*\) exists). If \(s\) does not intersect \(UPCI(L')\), then either \(s\) is below \(l^*\) or \(x(q) < x(u_1)\) (hence \(s\) is below \(l'\)). If \(s\) intersects \(UPCI(L')\), then \(s\) is certainly not below \(l^*\). Furthermore, \(q\) must be on the boundary of \(UPCI(L')\) and thus \(x(u_1) \leq x(q)\). Therefore, \(s\) is not below \(l'\) (and any other line in \(L'\)). \(\Box\)

The computational lemma below follows from the proof of Lemma 5.2.

Lemma 5.3 Given a non-vertical line segment \(s\) and a set \(L\) of \(m\) lines, suppose that the slopes of all the lines in \(L\) are \(\leq \) (resp., \(\geq\)) \(\text{slope}(s)\) and that \(UPCI(L)\) is available. Then using \(k\) processors, \(Test(L, s)\) can be done in \(O(\log m/\log(k + 1))\) time if \(UPCI(L)\) is stored in an array, and in \(O((\log m)^2/\log(k+1))\) time if \(UPCI(L)\) is stored in a rank tree.

Proof. WLOG, we assume that the slopes of the lines in \(L\) are all \(\leq \text{slope}(s)\) (the other case is symmetric). \(UPCI(L)\) is an unbounded convex polygon with \(|bd(UPCI(L))| = O(m)|. Because \text{slope}(s) \leq \text{slope}(l)\) for any \(l\) in \(L\), line \(l(s)\) intersects the boundary of \(UPCI(L)\) at most once. \(Test(L, s)\) can be easily answered by comparing the endpoints of \(s\) with the intersection of \(l(s)\) and \(bd(UPCI(L))\) (see the proof of Lemma 5.2). Therefore, all we need to compute is \(l(s) \cap bd(UPCI(L))\). If \(bd(UPCI(L))\) is stored in an array, then \(l(s) \cap bd(UPCI(L))\) can be easily found in \(O(\log m/\log(k + 1))\) time using \(k\) processors, by performing \((k+1)\)-ary searches on \(bd(UPCI(L))\). If \(UPCI(L)\) is stored in a rank tree \(T\), the time complexity then becomes \(O((\log m)^2/\log(k + 1))\) since an operation of accessing the \(i\)-th segment of \(bd(UPCI(L))\) requires \(O(\text{height}(T)) = O(\log m)\) time using one processor. \(\Box\)

Note that in Lemma 5.3, if \(k = O(m^\alpha)\) for any constant \(\alpha, 0 < \alpha < 1\), then the time complexities become \(O(1)\) (when \(UPCI(L)\) is stored in an array) and \(O(\log m)\) (when \(UPCI(L)\) is stored in a rank tree).

Based on Lemmas 5.2 and 5.3, the next lemma shows the basic operation done by the procedure for performing the lines-vs-segment tests.

Lemma 5.4 Given a non-vertical line segment \(s\) and a set \(L\) of \(m\) lines, suppose that \(L\) is partitioned into \(m^{1/c}\) subsets \(L_1, L_2, \ldots, L_{m^{1/c}}\) of equal size such that the slope of
every line in $L_{i+1}$ is $\geq$ the slopes of all the lines in $L_i$, and that $UPCI(L_1), UPCI(L_2), \ldots, UPCI(L_{m^{1/c}})$ are available (each stored in a rank tree), where $c > 1$ is a constant. Then, in $O(\log m)$ time using $O(m^{c'})$ processors, either the result of $Test(L,s)$ is found, or the test range is restricted to a unique $L_j$ (i.e., $Test(L,s)$ is completed by doing $Test(L_j,s)$), where $c'$ is a constant and $1/c < c' < 1$.

Proof. There are two possible cases: (1) there is a unique subset $L_j$ which contains two lines $l'$ and $l''$ such that $slope(l') < slope(s) < slope(l'')$, or (2) there is no such $L_j$. Given the partition of $L$, it is easy to find which case holds, within the desired complexity bounds. We then process $Test(L,s)$ using $m^{c'}$ processors, as follows. In case (2), apply Lemma 5.3 and do $Test(L_i,s)$, in parallel, for each $i = 1, 2, \ldots, m^{1/c}$. Each $Test(L_i,s)$ takes $O(\log m)$ time using $O(m^\alpha)$ processors (by Lemma 5.3), where $\alpha = c' - (1/c)$ is a constant and $0 < \alpha < 1$. The answer to $Test(L,s)$ can be easily obtained from the answers to the $Test(L_i,s)$'s (based on Lemma 5.2), in the desired complexity bounds. Therefore, case (2) is done in $O(\log m)$ time using $O(m^{c'})$ processors. In case (1), suppose that $L_j$ is the unique subset which gives rise to this case. We first in parallel do $Test(L_i,s)$ for each $i \neq j$ (in $O(\log m)$ time using $O(m^\alpha)$ processors). If the answer to $Test(L,s)$ can be derived from the answers to the $m^{1/c} - 1 Test(L_i,s)$'s (e.g., there is a line in $L - L_j$ that is above $s$), then we are done. Otherwise, the answers to the $m^{1/c} - 1 Test(L_i,s)$'s must be combined with the answer to $Test(L_j,s)$ in order to obtain the result for $Test(L,s)$; hence $Test(L,s)$ will be completed by performing $Test(L_j,s)$.

Note that if Lemma 5.4 can be recursively applied to $Test(L_j,s)$, then we only need to apply Lemma 5.4 repeatedly a constant number of times in order to reduce the size of the test range to $O(m^{c'})$ (at that point the brute force method can take over). In this way, $Test(L,s)$ is processed in totally $O(\log m)$ time using $O(m^{c'})$ processors.

Lemmas 5.3 and 5.4 both require that the common intersections of the upper half-planes be available before the tests are performed. The computation for the common intersection of $m$ half-planes, in general, requires $O(\log m)$ time and $\Omega(m \log m)$ total
work. In our situation, there can be as many as \( O(n) \) rays (and thus \( O(n) \) half-planes) to be considered. It would be impossible to compute the common intersection of \( O(n) \) half-planes in \( O(\log n) \) time using \( O(n/\log n) \) processors if the \( O(n) \) rays were arbitrarily given. Next, we show that if polygon \( P \) is weakly visible from a convex edge, then it is possible to obtain a subset of \( \text{Ray}^+(P) \) (resp., \( \text{Ray}^-(P) \)), denoted by \( \text{DR}^+(P) \) (resp., \( \text{DR}^-(P) \)), with the following properties: (i) \( \text{HE}^+(P) \) (resp., \( \text{HE}^-(P) \)) can be computed by using only \( \text{DR}^+(P) \) (resp., \( \text{DR}^-(P) \)), and (ii) \( \text{DR}^+(P) \) (resp., \( \text{DR}^-(P) \)) can be easily partitioned into two subsets, each containing rays sorted by slopes. The rays in \( \text{DR}^+(P) \) (resp., \( \text{DR}^-(P) \)) are called the dominating rays of \( \text{Ray}^+(P) \) (resp., \( \text{Ray}^-(P) \)). We only discuss the case for \( \text{DR}^+(P) \) (the case for \( \text{DR}^-(P) \) is similar).

WLOG, we assume that \( P \) is weakly visible from convex edge \( e_n \), that \( e_n \) is horizontal, and that \( l(e_n) \) is below \( P - e_n \). We define the polar angle of each ray \( r_i^+ \in \text{Ray}^+(P) \), denoted by \( \alpha(r_i^+) \), as follows: let the starting vertex \( v_i \) of \( r_i^+ \) be at the origin; then \( \alpha(r_i^+) \) is the polar angle from the positive x-axis counterclockwise to \( r_i^+ \). Note that \( 0 \leq \alpha(r_i^+) < 2\pi \). For rays \( r_i^+ \) and \( r_j^+ \) in \( \text{Ray}^+(P) \), \( i < j \), we say \( r_i^+ \) dominates \( r_j^+ \) if \( \alpha(r_i^+) \geq \alpha(r_j^+) \). Let \( \text{DR}^+(P) \) consist of the rays in \( \text{Ray}^+(P) \) that are not dominated by any ray in \( \text{Ray}^+(P) \).

**Lemma 5.5** For rays \( r_i^+ \) and \( r_j^+ \) in \( \text{Ray}^+(P) \), \( i < j \), if \( r_i^+ \) dominates \( r_j^+ \), then \( \text{HE}^+_j \subseteq \text{HE}^+_k \) for some \( k \), \( i \leq k < j \).

**Proof.** Let \( Q \) be the polygon formed by segment \( \overline{v_{i+1}h_i^+} \) and the subchain of \( bd(P) \) from \( v_{i+1} \) counterclockwise to \( h_i^+ \) (\( h_i^+ \) is the first-hit point of \( r_i^+ \)). WLOG, assume that \( h_i^+ \neq v_j \). There are two possible cases: (a) \( v_j \) is in \( Q \) (see Figure 5.3 (a)), and (b) \( v_j \) is not in \( Q \) (see Figure 5.3 (b)). We first show that in case (a), \( \text{HE}^+_j \subseteq \text{HE}^+_i \). If \( \text{HE}^+_j \) were not a subset of \( \text{HE}^+_i \), then \( h_j^+ \) would have to be outside \( Q \). For this to happen, \( r_j^+ \) must intersect \( \overline{v_{i+1}h_i^+} \) before hitting \( bd(P) \) at \( h_j^+ \) (since \( v_j \) is in \( Q \)); furthermore, \( r_j^+ \) must start in the right half-plane of \( r_i^+ \) and hit \( h_j^+ \) in the left half-plane of \( r_i^+ \). When \( \alpha(r_i^+) - \alpha(r_j^+) \leq \pi \), such an intersection between \( r_j^+ \) and \( \overline{v_{i+1}h_i^+} \) is impossible because \( v_j \) is in the right half-plane of \( r_i^+ \) and \( \alpha(r_i^+) \geq \alpha(r_j^+) \) (e.g., Figure
Figure 5.3 Illustrating the proof of Lemma 5.5. (a) $v_j$ is in $Q$. (b) $v_j$ is not in $Q$. 
5.3 (a)). When α(r_i^+) − α(r_j^+) > π, such an intersection between r_j^+ and v_{i+1}h_i^+ is also impossible for the following reason. That α(r_i^+) − α(r_j^+) > π implies that α(r_i^+) > π and α(r_j^+) < π. If r_j^+ did not intersect bd(P) − e_j before crossing v_{i+1}h_i^+, then v_j would not be weakly visible from e_n, a contradiction (see Figure 5.3 (c)). Hence HE_j^+ ⊆ HE_i^+ in case (a). In case (b), chain bd(i+1)j first intersects the right half-plane of r_i^+ and then goes to the left half-plane of r_i^+ by crossing r_i^+ at h_i^+. Let h_i^+ be on e_l, l > i + 1. Since P is weakly visible from e_n, the internal convex path ICP(bd(i+1)j) makes right turns only. It is not hard to see that there exists a vertex v_k, l ≤ k < j, such that v_{k+1} is a vertex on ICP(bd(i+1)j) and HE_j^+ ⊆ HE_k^+ (Figure 5.3 (b)).

Lemma 5.5 implies that HE^+(P) can be computed by using DR^+(P) only. It is easy to compute DR^+(P) in O(log n) time using O(n/ log n) processors (by doing parallel prefix [88, 89]). Note that the rays in DR^+(P) are sorted by polar angles. We further partition DR^+(P) into two subsets DR^+_1(P) and DR^+_2(P) such that the rays in each subset are sorted by slopes. This partition of DR^+(P) is done by splitting DR^+(P) using a ray whose polar angle is π. DR^+_1(P) (resp., DR^+_2(P)) contains the rays of DR^+(P) whose polar angles are all ≤ (resp., >) π. From now on, we assume that DR^+_1(P) and DR^+_2(P) are already available.

5.3 Detecting the Weak Visibility of a Star-Shaped Polygon

This section deals with the following problem: given that P is star-shaped and its kernel contains a convex vertex (say v_1), compute the bad edges in HE^+(P) by using DR^+(P). Clearly, P is weakly visible from e_1 (since P is visible from v_1) and e_1 is a convex edge. The algorithm for this problem consists of two phases. Phase 1.A computes the internal convex paths and the data structure storing the common intersections of the half-planes for the rays in DR^+(P). This phase also identifies some bad edges. Phase 1.B completes the identification of all bad edges; it makes use of the internal convex paths and the data structure constructed in Phase 1.A. We divide the computation of this algorithm into two separate parts, one using DR^+_1(P)
and the other using \( DR_1^+ (P) \). Due to the similarity between the two parts, we only discuss the part using \( DR_1^+ (P) \).

5.3.1 Phase 1.A

We first sketch the outline and describe the main operation of the algorithm, and then give the computational details and analysis.

5.3.1.1 The Outline

We associate each \( r_a^+ \in DR_1^+ (P) \) with \( v_a \), and denote by \( R_C \) the set of the rays in \( DR_1^+ (P) \) whose starting vertices are on a subchain \( C \) of \( bd(P) \). The outline below shows the divide-and-conquer strategies used by this phase.

Input. A subchain \( C \) of \( bd(P) \) with \( |C| = m \), \( R_C \), and a positive integer \( d \).

Case a.1. If \( m \leq d \), then use one processor to perform the computation in \( O(m) \) time.

Case a.2. If \( d < m \leq d^6 \), then divide \( C \) into two subchains \( C_1 \) and \( C_2 \) of equal size, and recursively solve the two subproblems (i.e., \( (C_1, R_{C_1}, d) \) and \( (C_2, R_{C_2}, d) \)) in parallel. Then perform the computation for \( C \) and \( R_C \) by using the output from the recursive calls on the two subproblems, with \( m/d \) processors and in \( O(\log m + (d \log d)^{1/2}) \) time.

Case a.3. If \( m > d^6 \), then partition \( C \) into \( g = (m/d)^{1/3} \) subchains \( C_1, C_2, \ldots, C_g \) of size \( m^{2/3} d^{1/3} \) each. Then recursively solve the \( g \) subproblems in parallel. Finally, perform the computation for \( C \) and \( R_C \) by using the output from the \( g \) recursive calls, with \( m/d \) processors and in \( O(\log m) \) time.

Observe that, if we could perform the various cases of the above outline within the claimed complexity bounds, then a procedure with such an outline would run in \( O(d + \log m) \) time with \( O(m/d) \) processors, since the recurrences for the time complexity \( t(m, d) \) and the processor complexity \( p(m, d) \) of the procedure would be as follows.
\[ t(m, d) = \begin{cases} 
  c_1d & \text{if } m \leq d \\
  t(m/2, d) + c_2(\log m + (d \log d)^{1/2}) & \text{if } d < m \leq d^6 \\
  t(m^{2/3}d^{1/3}, d) + c_3 \log m & \text{if } m > d^6 
\end{cases} \]

\[ p(m, d) = \begin{cases} 
  1 & \text{if } m \leq d \\
  \max\{m/d, 2p(m/2, d)\} & \text{if } d < m \leq d^6 \\
  \max\{m/d, (m/d)^{1/3}p(m^{2/3}d^{1/3}, d)\} & \text{if } m > d^6 
\end{cases} \]

where \( c_1, c_2, c_3 \) are constants. From the above recurrences, the following bounds for \( t(m, d) \) and \( p(m, d) \) can be easily proved by induction.

\[ t(m, d) \leq \begin{cases} 
  \alpha_1d & \text{if } m \leq d \\
  \alpha_2d + \beta_2(\log m + (d \log d)^{1/2}) \log d & \text{if } d < m \leq d^6 \\
  \alpha_3d + \beta_3 \log m & \text{if } m > d^6 
\end{cases} \]

\[ p(m, d) = \begin{cases} 
  1 & \text{if } m \leq d \\
  m/d & \text{if } m > d 
\end{cases} \]

where \( \alpha_1, \alpha_2, \alpha_3, \beta_2, \beta_3 \) are constants. Choosing \( d = \log n \), the above implies that \( t(n, d) = O(\log n) \) and \( p(n, d) = O(n/\log n) \). Therefore, a call to the procedure with input \((C, R_C, \log n)\), where \(|C| = n\), takes \( O(\log n) \) time using \( O(n/\log n) \) processors.

The input of the initial call made by our algorithm is \((bd_{1n}, R_C, \log n)\).

We must discuss what is exactly computed within the above outline. Let \( L(R_C) \) denote the set of the lines containing the rays in \( R_C \). The following is done in Case a.1 to Case a.3.

(i) Computing the internal convex path \( ICP(C) \);

(ii) building the data structure which stores the common intersection of the relevant half-planes of \( L(R_C) \) (e.g., \( UPCI(L(R_C)) \)) (this data structure is needed for performing the lines-vs-segment tests);
(iii) performing the lines-vs-segment tests to identify the bad edges on $C$ (we may not be able to identify all the bad edges in this phase; some of them are left to Phase 1.B).

Since $P$ is star-shaped, in each of the three cases, the internal convex paths for the subchains of $bd(P)$ can be computed in the required complexity bounds by using the algorithm in [33]. By Lemma 5.5, $R_C$ (thus $L(R_C)$) is sorted by slopes. Hence the common intersection of the relevant half-planes for $L(R_C)$ is computable in the required complexity bounds (e.g., by the algorithms in [65, 33]). Therefore, we will focus on how to use, in this phase, the internal convex paths and the data structure for the lines-vs-segment tests computed in the recursive calls to identify the bad edges.

5.3.1.2 The Main Operation

The following operation is crucial in the conquer stages of Case a.2 and Case a.3 (say, in Case a.3): given $ICP(C_s)$ and $UPCI(L(R_{C_s}))$, $s = 1, 2, \ldots, g$, determine the bad edges on $C_j$ by using $R_{C_i}$, for every pair of $i$ and $j$, $1 \leq i < j \leq g$. We classify every $ICP(C_j)$ into one of four possible cases according to its size, and show how to determine the bad edges on $C_j$ by using $R_{C_i}$ in each case. The classification is as follows.

Case (a): $ICP(C_j)$ has more than 3 segments (see Figure 5.4 (a)). Then all the edges on $C_j$ are bad. In Figure 5.4 (a), the edges on $C_j$ are all contained in $HE_{b-1}^+ \cup HE_{d+1}^-$. Hence there is no need to use $R_{C_i}$ to identify the bad edges on $C_j$.

Case (b): $ICP(C_j)$ has exactly 3 segments (see Figure 5.4 (b)). Then the edges on subchains $bd_{ab}$ and $bd_{cd}$ of $C_j$ are all bad (cf. Figure 5.4 (b)), because they are contained in $HE_{b-1}^+ \cup HE_{c+1}^-$. Furthermore, if $\overline{v_bv_c}$ is not an edge of $P$ (i.e., $c > b+1$), then all the edges on $C_j$ are bad because the edges on subchain $bd_{bc}$ are also contained in $HE_{b-1}^+ \cup HE_{c+1}^-$. The only edge on $C_j$ that may not be bad is $\overline{v_bv_c}$, provided that $c = b + 1$. Thus the problem in this case, when $c = b + 1$, is that of finding whether $\overline{v_bv_{b+1}}$ is bad with respect to $R_{C_i}$. 
Figure 5.4 The four possible cases of $|ICP(C_j)|$. 
Case (c): $ICP(C_j)$ has 2 segments (see Figure 5.4 (c)). Then clearly all the edges on $C_j$, except the two edges $\overline{v_{b-1}v_b}$ and $\overline{v_bv_{b+1}}$, are bad (cf. Figure 5.4 (c)). We only need to check $\overline{v_{b-1}v_b}$ and $\overline{v_bv_{b+1}}$ by using $R_{C_i}$.

Case (d): $ICP(C_j)$ has 1 segment (see Figure 5.4 (d)). If there is a ray $r^+_t \in R_{C_i}$ such that $bd_{ab} \subseteq HE_t^+$, then certainly all the edges on $C_j$ are bad. Otherwise, we might have to “shoot” the rays of $R_{C_i}$ onto $C_j$ in order to find which of the edges on $C_j$ are bad (this ray-shooting on $C_j$ is to be done in Phase 1.B). Thus we need to check whether $bd_{ab} \subseteq HE_t^+$ for some $r^+_t \in R_{C_i}$. This check is done by testing the segment $\overline{va_vb}$ of $ICP(C_j)$ against all the rays in $R_{C_i}$.

From the discussion above, it is clear that the main computation in Case (a) to Case (d) is to test an edge of $C_j$ (in Case (b) and Case (c)) or a segment of $ICP(C_j)$ (in Case (d)) against the rays in $R_{C_i}$, in order to find out whether the edge of $C_j$ or $C_j$ itself is bad with respect to $R_{C_i}$. We call such a test a bad-segment test.

We need some notations for describing the solution to the bad-segment tests. WLOG, let $v_1$ be at the origin and $e_1$ be on the positive $x$-axis (hence $P - e_1$ is above $l(e_1)$). The polar angle of a point $p \in bd(P) - v_1$, denoted by $\alpha(p)$, is the angle from the positive $x$-axis counterclockwise to the ray starting at $v_1$ and passing $p$. Since $v_1$ is in the kernel of $P$ and $v_1$ is convex, it follows that $0 \leq \alpha(p) < \pi$ for each point $p \in bd(P) - v_1$, and that the polar angles of the points on $bd(P) - v_1$, from $v_1$ counterclockwise to $v_n$, are in non-decreasing order. For every $r^+_t \in DR_t^+(P)$, $h_t^+$ is on $bd_{t+2}$. For a ray $r^+_t \in DR_t^+(P)$ and a segment $s$, we say $s$ is properly contained in the upper-right (resp., upper-left) quarter-plane of $r^+_t$ iff (i) $s$ is contained in the intersection of the right (resp., left) half-plane of $r^+_t$ and the left half-plane of the ray starting at $v_1$ and passing $v_{t+1}$, and (ii) $s$ does not intersect $r^+_t$. Observe that for an edge $e_t$ of $P$, if $e_t \in HE_t^+$, then $e_t$ is properly contained in the upper-right quarter-plane of $r^+_t$.

WLOG, we assume that for each ray $r^+_t$ in $R_C$, the right half-plane of $r^+_t$ is equal to the lower half-plane of line $l(r^+_t)$.
We would like to obtain the answer to the bad-segment test on $R_{C_i}$ and a segment $s$ (of $C_j$ or $ICP(C_j)$) by performing a lines-vs-segment test $Test(L(R_{C_i}), s)$ (because we can handle $Test(L(R_{C_i}), s)$ by using Lemma 5.4). In general, however, a bad-segment test cannot be answered by a lines-vs-segment test. For example, line $l(r)$ intersecting a segment $s'$ does not necessarily imply that ray $r$ also intersects $s'$. Furthermore, even when the half-line defined by a ray $r \in R_{C_i}$ does intersect segment $s$ (of $C_j$ or $ICP(C_j)$), $r$ itself may first-hit a point on some $C_k$, $i \leq k < j$, before intersecting $s$ (i.e., $C_k$ blocks $r$ from reaching $s$ if $r$ is viewed as a beam of light emanating from its starting vertex). Therefore, even if the result of $Test(L(R_{C_i}), s)$ does indicate that $s$ is properly contained in the upper-right quarter-plane of $r^+_t$ for a $r^+_t \in R_{C_i}$, the rays of $R_{C_i}$ may be totally blocked from $C_j$. This means that, in this situation, no edge of $C_j$ truly belongs to $HE^+_t$ for any $r^+_t \in R_{C_i}$, and hence no edge of $C_j$ is bad with respect to $R_{C_i}$. If we had to find out whether $R_{C_i}$ is totally blocked from $C_j$, then for every $k$, $i \leq k < j$, we might either do $O(|ICP(C_k)|)$ bad-segment tests (for $R_{C_i}$ and each segment of $ICP(C_k)$), or “shoot” each ray of $R_{C_i}$ onto $ICP(C_k)$ (by doing a binary search on $ICP(C_k)$). Since we can have $|ICP(C_k)|$ proportional to $|C_k|$ and $|R_{C_i}|$ proportional to $|C_i|$, either method would be too expensive to be performed
within the desired complexity bounds. The next lemma saves us from doing these costly computations.

Lemma 5.6 If a ray \( r \in R_{C_i} \) first-hits some \( C_k \) at edge \( e_s \), \( i \leq k < j \), and if edge \( e_w \) on \( C_j \) is properly contained in the upper-right quarter-plane of \( r \), then there exists a vertex \( v_z \) on \( bd_{(s-1)(w-2)} \) such that \( e_w \in HE^+_z \).

Proof. Chain \( bd_{(s-1)w} \) must start in the right half-plane of \( r \). It then intersects \( r \) on \( e_s \), and eventually enters the right half-plane of \( r \) to join \( v_w \) (see Figure 5.5). Since \( P \) is visible from \( v_1 \), \( ICP(bd_{s(w-1)}) \) makes right turns only. Hence there must exist a vertex \( v_{z+1} \) on \( ICP(bd_{s(w-1)}) \) such that \( s - 1 \leq z \leq w - 2 \) and \( e_w \in HE^+_z \).

Lemma 5.6 implies that if edge \( e_w \) of \( C_j \) is properly contained in the upper-right quarter-plane of a ray \( r \in R_{C_i} \), then \( e_w \) is definitely a bad edge. Note that for any \( k' \) such that \( k' < i \) or \( k' > j \), \( C_{k'} \) cannot block the rays of \( R_{C_i} \) from reaching \( C_j \) (by Lemma 5.1).

The next lemma justifies the use of the lines-vs-segment tests for answering the bad-segment tests.

Lemma 5.7 For \( i < j \), a bad-segment test on \( R_{C_i} \) and a segment \( s \) of \( C_j \) or \( ICP(C_j) \) can be answered by performing \( Test(L(R_{C_i}), s) \).

Proof. Let \( r^+_i \) be a ray in \( R_{C_i} \). Recall that by our assumption, the left half-plane of \( r^+_i \) is the same as the upper half-plane of \( l(r^+_i) \). The lemma holds if the following is true: (i) \( s \) is properly contained in the upper-right quarter-plane of \( r^+_i \) iff \( s \) is below \( l(r^+_i) \), and (ii) \( s \) is properly contained in the upper-left quarter-plane of \( r^+_i \) iff \( s \) is above \( l(r^+_i) \). We only give the proof for (i) (the proof for (ii) is very similar). If \( s \) is properly contained in the upper-right quarter-plane of \( r^+_i \), then \( s \) is clearly below \( l(r^+_i) \). If \( s \) is below \( l(r^+_i) \), then \( s \) is properly contained in the lower half-plane of \( l(r^+_i) \) (i.e., the right half-plane of \( r^+_i \)) and certainly \( s \neq \overline{v_i v_{i+1}} \). The facts that \( i < j \), that \( s \neq \overline{v_i v_{i+1}} \), that \( P \) is visible from \( v_1 \), and that \( v_1 \) is convex, together imply that \( \alpha(v_{i+1}) \leq \alpha(p) < \pi \) for each point \( p \) on \( s \). This means that \( s \) is to the left of the ray \( r' \) starting at \( v_1 \) and passing \( v_{i+1} \), and thus \( s \) is in the left half-plane of \( r' \).
If the result of Test($L(RCi), s$) indicates that neither (i) $s$ is below a line in $L(RCi)$ nor (ii) $s$ is above all the lines in $L(RCi)$, then there must be a ray $r \in R Ci$ such that the half-line defined by $r$ intersects $s$. When this is the case, we need to distinguish two types of the intersection between $r$ and $s$.

Suppose that the result of Test($L(RCi), s$) indicates that neither (i) nor (ii) occurs. Let $r$ be a ray in $R Ci$ that intersects $s$. Let $s = \overline{v_a v_b}$, $a < b$, and let $r(s)$ be the ray starting at $v_a$ and passing $v_b$. If the starting vertex of $r$ is properly contained in the right half-plane of $r(s)$, then we say $r$ pseudo-hits $s$; otherwise, $r$ does not pseudo-hit $s$.

Furthermore, we say $R Ci$ only-pseudo-hits $s$ if (1) there is at least one ray in $R Ci$ that intersects $s$, and (2) for each ray $r' \in R Ci$, $r'$ intersecting $s$ implies that $r'$ pseudo-hits $s$; otherwise, we say $R Ci$ does not only-pseudo-hit $s$. We distinguish the types of the intersections between $R Ci$ and $s$ because only when $R Ci$ does not only-pseudo-hit $s$ can the rays in $R Ci$ first-hit $bd_ab - v_a$ (it is still possible that $R Ci$ is blocked from $bd_ab$). If $R Ci$ only-pseudo-hits $s$, then clearly no ray in $R Ci$ can first-hit $bd_ab - v_a$.

We define the polar angle of $s$ as $\alpha(s) = \alpha(r(s))$ (with obvious meaning for $\alpha(r(s))$). The following lemma characterizes the types of the intersections between a ray and $s$.

**Lemma 5.8** Suppose that a ray $r_i^+ \in R Ci$ intersects a segment $s$ of $ICP(C_i)$, $i < j$. Then $r_i^+$ pseudo-hits $s$ iff $\alpha(r_i^+) > \alpha(s)$.

**Proof.** First, we show that if $\alpha(r_i^+) \leq \alpha(s)$, then $r_i^+$ does not pseudo-hit $s$. There are two cases. Case (1): $\alpha(s) \leq \pi$. Then a ray $r'$ whose starting point is in the interior of the right half-plane of $r(s)$ and whose polar angle is $\leq \alpha(s)$ cannot intersect $s$. Hence it would be a contradiction if we had $r_i^+$ pseudo-hitting $s$. Case (2): $\pi < \alpha(s) < 2\pi$. Because $P$ is star-shaped and $v_1$ is convex, it follows that the left half-plane of $r(s)$ contains $v_1$ and that both $s$ and $e_i$ are above the $x$-axis (except possibly at one endpoint). If $r_i^+$ did pseudo-hit $s$, then we would have that $v_1$ is in the right half-plane of $r(s)$ and that $\alpha(s) - \alpha(r_i^+) > \pi$. Hence $\alpha(r_i^+) < \pi$. But then for edge $e_i$, $\alpha(v_i) > \alpha(v_{i+1})$, a contradiction.
Second, we show that if $\alpha(r_i^+) > \alpha(s)$, then $r_i^+$ pseudo-hits $s$. There are also two cases. Case (1): $\alpha(s) \leq \pi$. Let $s = \overline{va_v_b}$, $a < b$. Suppose that the starting vertex of $r_i^+$ were in the left half-plane of $r(s)$. If $\alpha(r_i^+) \leq \pi$, then it would be impossible for $r_i^+$ to intersect $s$, a contradiction. If on the other hand $\alpha(r_i^+) > \pi$, then $v_a$ would not be visible from $v_1$, also a contradiction. Case (2): $\pi < \alpha(s) < 2\pi$. If the starting vertex of $r_i^+$ were in the left half-plane of $r(s)$, then obviously $r_i^+$ could not intersect $s$, again a contradiction.

The next Lemma shows that the lines-vs-segment tests can be used to identify the intersection type between $R_{C_i}$ and $s$.

Lemma 5.9 The type of the intersection between $R_{C_i}$ and a segment $s$ of $ICP(C_j)$, $i < j$, can be determined by using the lines-vs-segment tests, in the same complexity bounds as those for performing $Test(L(R_{C_i}), s)$.

Proof. First we do $Test(L(R_{C_i}), s)$ to find out whether $s$ is below a line in $L(R_{C_i})$ or $s$ is above all the lines in $L(R_{C_i})$. If neither occurs, then by Lemma 5.8, we know that $R_{C_i}$ does not only-pseudo-hit $s$ iff there is a ray $r$ in $R_{C_i}$ such that $\alpha(r) \leq \alpha(s)$ and $r$ intersects $s$ (note that we are already sure that $l(r)$ is not above $s$). Hence, what we need to decide is whether every line $l(r)$ in $L(R_{C_i})$ (with $\alpha(r) \leq \alpha(s)$) is below $s$ (by using the lines-vs-segment tests). This can be done by using a procedure whose structure is similar to the one for Lemma 5.4. The only difference between this procedure and the one for Lemma 5.4 is that here we use $\alpha(s)$ (instead of $slope(s)$) to determine the unique subset of $R_{C_i}$ to which the procedure is to be recursively applied. Because $R_{C_i}$ is sorted by polar angles as well as by slopes, the complexity bounds of this procedure are the same as those for performing $Test(L(R_{C_i}), s)$. If the output of the procedure indicates that every line $l(r)$ in $L(R_{C_i})$ (with $\alpha(r) \leq \alpha(s)$) is below $s$, then we know that $R_{C_i}$ only-pseudo-hits $s$; otherwise, $R_{C_i}$ does not only-pseudo-hit $s$. □

For a segment $s$ of $C_j$ or $ICP(C_j)$, where $s = \overline{va_v_b}$, $a < b$, if the result of $Test(L(R_{C_i}), s)$ indicates that $s$ is neither below a line in $L(R_{C_i})$ nor above all the lines in $L(R_{C_i})$, then one of the following three situations occurs: (1) if $s$ is an edge
of $C_j$, then $s$ is not bad with respect to $R_{C_1}$, or (2) if $s$ is not an edge of $C_j$ and $R_{C_i}$ only-pseudo-hits $s$, then no edge on $bd_{ab}$ is bad with respect to $R_{C_1}$, or (3) if $s$ is not an edge of $C_j$ and $R_{C_i}$ does not only-pseudo-hit $s$, then $|ICP(C_j)|$ may be Case (d) (with $C_j = bd_{ab}$), which is to be handled in Phase 1.B.

5.3.1.3 Performing Test($L(R_{C_i}), s$) in Case a.2 and Case a.3

We need to discuss how the lines-vs-segment tests are actually performed in the algorithm. We first discuss this for Case a.3, and then for Case a.2. Our primary tool is Lemma 5.4.

In Case a.3, there are $O(g^2) = O((m/d)^{2/3})$ pairs of $C_i$ and $C_j$, $i < j$, where $m = |C|$. For each pair of $C_i$ and $C_j$, we need to do Test($L(R_{C_i}), s$) for $O(1)$ segments $s$ on $C_j$ or $ICP(C_j)$, if $ICP(C_j)$ is not of Case (a). There are totally $O(m/d)$ processors available. Thus $O(g) = O((m/d)^{1/3})$ processors are allocated to each pair of $C_i$ and $C_j$. It suffices to show how to do one Test($L(R_{C_i}), s$) using $O(g)$ processors.

To perform Test($L(R_{C_i}), s$) in $O(\log m)$ time, we need to achieve two things: (i) in performing the test, Lemma 5.4 is recursively applied only $O(1)$ times, and (ii) the size of the test range, after (i) is done, is reduced to $O((m/d)^{1/3})$ (so that the brute force method can take over).

In the conquer stage of Case a.3, $UPCI(L(R_{C_i}))$ are available from the recursive calls (each stored in a rank tree), for $i = 1, 2, \ldots, g$, and we need to compute $UPCI(L(R_C))$ from the $UPCI(L(R_{C_i}))$'s. Because $L(R_C)$ is sorted by slopes, for each $i$, $bd(UPCI(L(R_{C_i}))) \cap bd(UPCI(L(R_C)))$ consists of at most one connected component, and $bd(UPCI(L(R_{C_i}))) - bd(UPCI(L(R_C)))$ consists of at most two connected components. After $UPCI(L(R_C))$ is computed, we still retain the (at most) two connected components of $bd(UPCI(L(R_{C_i}))) - bd(UPCI(L(R_C)))$ in two separate rank trees. This structure can be readily maintained at every recursion level of Case a.3 in the desired complexity bounds. Using this structure, it is easy to obtain the information about each $UPCI(L(R_{C_i}))$ (either from $bd(UPCI(L(R_C)))$ or from the two connected components of $bd(UPCI(L(R_{C_i}))) - bd(UPCI(L(R_C))))$. Furthermore, for
a subchain $C_i^k$ of $C_i$, the information about $UPCI(L(R_{C_i^k}))$ can be obtained from $bd(UPCI(L(R_{C_i})))$ and $bd(UPCI(L(R_{C_i^k}))) - bd(UPCI(L(R_{C_i})))$. In general, the information about the common intersection of the half-planes computed at a lower level of the algorithm can be obtained from the information stored at its ancestor levels.

WLOG, we assume that in the recursive call on $(C_i, R_{C_i}, d)$, $C_i$ was partitioned into $O((m/d)^{2/3})$ subchains (the case where $C_i$ was partitioned as in Case a.2 can be easily taken care of within the desired complexity bounds). In performing $Test(L(R_{C_i}), s)$, we repeat the following two steps. (i) Apply Lemma 5.4 to $Test(L(R_{C_i}), s)$ using $O((m/d)^{1/3})$ processors. (Note that the information on $UPCI(L(R_{C_i}^k))$, for each subchain $C_i^k$ of $C_i$, is obtained using the structure described in the previous paragraph.) (ii) Either we have found the answer to $Test(L(R_{C_i}), s)$ in (i), or we have reduced the test to a unique subchain $C_i^k$ of $C_i$, in which case we continue the test by repeating (i) for $Test(L(R_{C_i^k}), s)$.

It is clear that every execution of step (i) above requires $O(\log m)$ time. What remains to be shown is that we only need to repeat step (i) $O(1)$ times before reducing the size of the test range to $O((m/d)^{1/3})$.

The size of chain $C$ is $m$ at the current recursion level of the algorithm. The following can be easily proved by induction: at the $i$-th recursion level below, $i \geq 1$, the chain size is $O(m^{f(i)}d^{1-f(i)})$ and the chain is partitioned into $O((m/d)^{f(i)(1/3)})$ subchains (for the $(i+1)$-th level), where $f(i) = (2/3)i$. We want to stop the recursive procedure for performing $Test(L(R_{C_i}), s)$ when it reaches the structure for the $i$-th level below (in the recursion tree of Phase 1.A), for some $i \geq 1$, such that the size of the chain at that level is $\leq (m/d)^{1/3}$. Hence we have

$$(m/d)^{1/3} \geq m^{f(i)}d^{1-f(i)},$$

which is equivalent to

$$m^{(1/3)-f(i)} \geq d^{(4/3)-f(i)}.$$ 

Choosing $i$ to be 5, the above inequality becomes

$$m^{49/243} \geq d^{1+(49/243)},$$
and it holds as long as $m \geq d^8$. Thus, the test range size is reduced to $O((m/d)^{1/3})$
by going down at most five levels in the recursion tree of Phase 1.A.

In Case a.2, there is only one pair of subchains of $C$, i.e., $C_1$ and $C_2$, and we do
$Test(L(R_{C_1}), s)$ for at most $O(1)$ segments $s$ on $C_2$ or $ICP(C_2)$. The lines-vs-segment
tests in Case a.2 use a simpler structure than in Case a.3, as follows. $C$ is divided into
$m/d$ subchains of size $d$ each. Let each of the $O(m/d)$ processors be in charge of the
$O(d)$ rays whose starting vertices belong to one such subchain. Every processor builds
the following structure for its $O(d)$ rays: (1) computing the common intersection of
the relevant half-planes for the $O(d)$ rays and storing the common intersection in an
array, (2) partitioning the $O(d)$ rays into $O((d/ \log d)^{1/2})$ subsets of size $O((d \log d)^{1/2})$
each, and (3) computing the common intersection of the relevant half-planes for each
subset of $O((d \log d)^{1/2})$ rays and storing the common intersection in an array. This
data structure is prepared once and remains unchanged throughout Case a.2. Each
processor can easily build its structure in $O(d)$ time because the rays are sorted by
slopes.

A $Test(L(R_{C_1}), s)$ in Case a.2 is done as follows. Since the rays are sorted by slopes,
the sets of the rays for all except possibly one processor (say, processor $p_k$) satisfy the
conditions of Lemma 5.3. Hence every processor $p_i$, $i \neq k$, tests $s$ against the set of
the rays for $p_i$, in parallel, by Lemma 5.3. This requires $O(\log d)$ time. If the $p_i$’s find
the answer to $Test(L(R_{C_1}), s)$, then we are done. Otherwise, the $O(m/d)$ processors
apply Lemma 5.3 again to all the subsets of the rays for $p_k$ (except possibly one subset,
as noted before), in $O((d \log d)^{1/2})$ total work (since there are $O((d/ \log d)^{1/2})$ subsets
and the test on each subset requires $O(\log d)$ time using one processor). Finally, the
brute force method is applied to the remaining subset of $p_k$ to complete the test,
in $O((d \log d)^{1/2})$ work. The sub-answers from the $O(m/d)$ processors are readily
combined in $O(\log m)$ time. Therefore, the total time for doing $Test(L(R_{C_1}), s)$ in
Case a.2 is $O(\log m + (d \log d)^{1/2})$. Note that it is easy to convert the structure used
for Case a.2 into the one for Case a.3, in the desired complexity bounds.
Comment: One can actually replace the \((d \log d)^{1/2}\) term above by \(\log^2 d\), but the presentation of the procedure achieving this will be considerably more complicated.

5.3.1.4 The Procedure for Phase 1.A

We need some notations for describing the procedure. A ray \(r \in R_C\) for a chain \(C\) is said to be in-hitting (resp., out-hitting) to \(C\) if the first-hit point of \(r\) is known (resp., not known) to be on \(C\). For a subchain \(C'\) of \(C\), if the subset \(R_{C'}\) of \(R_C\) contains an out-hitting ray to \(C\), then \(R_{C'}\) is out-hitting to \(C\); otherwise, \(R_{C'}\) is in-hitting to \(C\). We say that an in-hitting (resp., out-hitting) ray to \(C\) gives rise to a local bad interval on \(C\) (resp., a crossing bad interval from \(C\)). For an interval \(I\) on \(bd(P)\) whose interior does not contain \(v_1\), the endpoint of \(I\) that is encountered first (resp., second) when walking along \(bd(P)\) counterclockwise starting from \(v_1\) is called the first (resp., second) endpoint of \(I\). Note that for each \(r_i^+ \in R_C\), \(HE_i^+\) forms an interval of bad edges; furthermore, the union of all the crossing bad intervals from \(C\) caused by the out-hitting rays to \(C\) is one contiguous interval that contains the second endpoint of \(C\). We call this union the crossing bad interval of \(R_C\) from \(C\). No bad interval contains \(v_1\) because \(P\) is visible from \(v_1\). Since both endpoints of a bad interval are vertices of \(P\), we denote the interval by the vertex indices of its endpoints.

In the procedure below, the operation of "storing the bad interval \([a, b]\)", \(a < b\), means the following computation: if there is already a vertex \(v_c\) (which is the second endpoint of the bad interval \([a, c]\) stored before) associated with \(v_a\), then associate \(v_l\) with \(v_a\), where \(l = \max\{b, c\}\); otherwise, associate \(v_b\) with \(v_a\). If a chain \(C\) is known to be contained in a bad interval, then we say \(C\) is covered. We just discuss Case a.1 and Case a.3 because Case a.2 is a simple case of Case a.3.

In Case a.1, we do the following for \(C = bd_{st}\), where \(|C| \leq d\) and \(s < t\), in \(O(d)\) time and one processor.

(1) Based on the case of \(|ICP(C)|\) (cf. Subsubsection 5.3.1.2), store the bad intervals of \(C\). Mark \(C\) as covered if \([s, t]\) itself is known to be bad.
(2) Perform \([121]\) sequentially on \(C\). If \(C\) is covered, then ignore all the in-hitting rays to \(C\); otherwise, if an in-hitting ray \(r_a^+ \in R_C\) is found to hit on \(e_b - v_b\), then store the bad interval \([a + 2, b]\) provided that \(b > a + 2\). If \(R_C\) contains no out-hitting ray to \(C\), then mark \(R_C\) as in-hitting to \(C\); otherwise, mark \(R_C\) as out-hitting to \(C\), and find the smallest vertex index \(a'\) such that \(r_{a'}^+ \in R_C\) is out-hitting to \(C\). Let \(a' + 2\) be the first endpoint of the crossing bad interval of \(R_C\) from \(C\) (the second endpoint of this interval is beyond \(C\) and is to be decided later in the algorithm).

In the conquer stage of Case a.3, we do the following for \(C = bd_{st}\), where \(|C| = m\) and \(s < t\), in \(O(\log m)\) time using \(O(m/d)\) processors. Note that \(C\) is partitioned into subchains \(C_1, C_2, \ldots, C_g\).

(1') Based on the case of \(|ICP(C)|\) (cf. Subsubsection 5.3.1.2), store the bad intervals of \(C\). Mark \(C\) as covered if \([s, t]\) itself is known to be bad.

(2') For every pair of \(R_{C_i}\) and \(ICP(C_j)\) such that \(i < j\), \(R_{C_i}\) is out-hitting to \(C_i\), and \(|ICP(C_j)|\) is not Case (a), perform \(Test(L(R_{C_i}), s)\) for each segment \(s\) on \(ICP(C_j)\).

(3') For every \(R_{C_i}\) that is out-hitting to \(C_i\), do the following. If for every \(j\) such that \(j > i\) and \(|ICP(C_j)|\) is not Case (a), \(s\) is below some line \(\in L(R_{C_i})\) for each segment \(s\) on \(ICP(C_j)\), then mark \(R_{C_i}\) as out-hitting to \(C\). Otherwise, mark \(R_{C_i}\) as in-hitting to \(C\); also, let \(S_i = \{s' | s'\ \text{is a segment on one of these} \ ICP(C_j)\text{'s such that} s' \text{is not below any line in} L(R_{C_i})\}\), and let \(s_i\) be the segment in \(S_i\) such that the vertex indices of the endpoints for \(s_i\) are no bigger than the vertex indices of the endpoints for any other \(s' \in S_i\).

(4') For every \(R_{C_i}\) in-hitting to \(C\) and out-hitting to \(C_i\), do the following. Let \(s_i\) be on \(ICP(C_v)\) and \(s_i = v_b \bar{v}_{c_i}\), \(b_i < c_i\), and let the first endpoint of the crossing bad interval of \(R_{C_i}\) from \(C_i\) be \(a_i\). Store the bad interval \([a_i, b_i]\) provided that \(b_i > a_i\). If \(s_i\) is not above all the lines in \(L(R_{C_i})\) and \(R_{C_i}\) does not only-pseudo-hit
then the following is done: if $|ICP(C_i')|$ is Case (d), then the shooting of $R_{C_i}$ onto $C_i'$ may be left to Phase 1.B; otherwise, $|ICP(C_i')|$ is Case (b) or Case (c). (By Case (b) or Case (c), there is at most one edge $e$ of $bd_{b_i}C_i$, which may not be bad with respect to $R_{C_i}$, and $bd_{b_i} - e$ is surely bad.) Do $Test(L(R_{C_i}), e)$ for such a unique edge $e$ (if it exists) to determine whether $e$ is bad with respect to $R_{C_i}$, and store the bad interval accordingly.

(5') If there is an $R_{C_i}$ such that $R_{C_i}$ is out-hitting to $C$, then mark $R_C$ as out-hitting to $C$. Let the first endpoint of the crossing bad interval of $R_C$ from $C$ be the smallest $a$ in $\{a_i \mid a_i$ is the first endpoint of the crossing bad interval of $R_{C_i}$ from $C_i$ and $R_{C_i}$ is out-hitting to $C\}$.

(6') For every $C_j$ ($C_j = bd_{a_j}v_j$), if there is an $i$, $i < j$, such that $R_{C_i}$ is out-hitting to $C$ or $R_{C_i}$ is in-hitting to $C$ and the second endpoint of the crossing bad interval of $R_{C_i}$ from $C_i$ is known to be on $C_k$ for some $k$, $j < k$, then mark $C_j$ as covered and store the bad interval $[a', b']$.

(7') For every $R_{C_i}$ out-hitting to $C_i$ and in-hitting to $C$, if $R_{C_i}$ may be shot onto $C_i'$ in Phase 1.B, then do the following. Let the first endpoint $a$ of the crossing bad interval of $R_C$ from $C$ be on $C_k'$ for some $k'$, let the first endpoint of the crossing bad interval of $R_{C_i}$ from $C_i$ be $a_i$, and let $v_{l'}$ be the last vertex of $C_{l'}$. If $i' > k'$, then store the bad interval $[a_i, b']$ ($C_i'$ is contained in the crossing bad interval of $R_C$ from $C$). Otherwise, if there is an $R_{C_i}$ such that $R_{C_i}$ may be shot onto $ICP(C_i')$ in Phase 1.B or the crossing bad interval of $R_{C_i}$ from $C_i$ has its second endpoint on $C_l'$, where $l < i' < l'$, then also store the bad interval $[a_i, b']$. Otherwise, mark $R_{C_i}$ as "to be shot onto $C_i'$ in Phase 1.B."

The complexity bounds of the above procedure follow from the discussions in Subsubsections 5.3.1.2 and 5.3.1.3. We only show the correctness of the procedure.

The correctness of the procedure will follow if we can prove the following for every $R_{C_i}$ which is out-hitting to $C_i$: if $s_i$ exists (i.e., $R_{C_i}$ in-hitting to $C$, cf. step (3'))), then (i) no ray in $R_{C_i}$ can first-hit a point on $bd_{c_{i,1}} - v_{c_i}$, where $s_i = \overline{v_{b_i}v_{c_i}}$ and $b_i <
(ii) \([a_i, b_i]\) is a bad interval, where \(a_i\) is the first endpoint of the crossing bad interval of \(R_{C_i}\) from \(C_i\); if \(s_i\) does not exist (i.e., \(R_{C_i}\) out-hitting to \(C\)), then interval \([a_i, c]\) is bad, where \(c\) is the last vertex of \(C\). We just prove the case where \(s_i\) exists (the proof for the other case is similar).

Suppose that for an \(R_{C_i}\) out-hitting to \(C_i\), \(s_i\) exists. Then by Lemma 5.1 and by the definition of \(s_i\), \(R_{C_i}\) can never first-hit any point on \(bd_{c_1} - v_{c_1}\). This proves (i).

To prove (ii), we use an induction argument. We assume inductively that interval \([a_i, b']\) is bad, where \(v_{b'}\) is the last vertex of \(C_i\). (Note that the induction basis is provided by Case a.1.) Let \(s_i\) be on \(ICP(C_i')\) for a unique \(C_i'\). For any \(k, i < k < i'\), either \(|ICP(C_k)|\) is Case (a) (hence all the edges on \(C_k\) are bad), or \(|ICP(C_k)|\) is not Case (a) and each segment \(s' = v_{b_k}v_{c_k}\) on \(ICP(C_k)\) is below a line in \(L(R_{C_i})\) (by the definition of \(s_i\)), where \(b_k < c_k\). But then, by Lemma 5.6, all the edges on \(bd_{b_k,c_k}\) are bad. Thus in either situation, all the edges on \(C_k\) are bad. For a segment \(s'' = v_{b''}v_{c''}\) on \(ICP(C_i')\) such that \(b'' < c'' \leq b_i\), \(s''\) must be below a line in \(L(R_{C_i})\) (by the definition of \(s_i\)); hence all the edges on \(bd_{b''c''}\) are bad by Lemma 5.6. Therefore, interval \([b', b_i]\) is bad. Together with the induction hypothesis that \([a_i, b']\) is bad, we conclude that \([a_i, b_i]\) is bad.

5.3.2 Phase 1.B

By using the structure built in Phase 1.A, this phase finishes the identification of the bad edges in \(HE^+(P)\). Phase 1.B has the same algorithmic outline as Phase 1.A. Its computation follows the recursion tree of Phase 1.A, level by level, from the root down to the leaves.

The following operation is typical in this phase: for chains \(C\) and \(C'\) on \(bd(P)\), where \(C\) and \(C'\) are disjoint (except possibly at one endpoint) and \(|ICP(C')|\) is Case (d), identify the bad edges on \(C'\) by using \(R_C\). What we do is to shoot \(R_C\) onto each subchain \(C_i'\) of \(C'\). At each \(C_i'\), we encounter one of the four possible cases for \(ICP(C_i')\), and we may have to continue, recursively, the shooting of \(R_C\) onto a unique subchain \(C_j'\) of \(C'\), where \(|ICP(C_j')|\) is again Case (d).
In this top-down procedure (from the root down to the leaves of the recursion tree of Phase 1.A), one thing must be handled carefully, as follows. If one keeps using the same data structure for \( R_C \) (e.g., \( UPCI(L(R_C)) \)) at the upcoming recursion levels below \( C' \), then the same \( O(\log |C|) \) time is spent at each level. This may not give an \( O(\log n) \) time algorithm; instead, the time bound so resulted could be \( O(\log n \log \log n) \). To avoid this inefficiency, what we do when shooting \( R_C \) onto \( C' \) is to partition \( C \), say, \( g = (|C|/d)^{1/3} \) subchains \( C_1, C_2, \ldots, C_g \) (as in Phase 1.A). Then, by using the structures for the \( UPCI(L(R_{C_k})) \)'s (rather than \( UPCI(L(R_C)) \)), every \( R_{C_k} \) is shot independently onto each of the \( C_i \)'s. Observe that the union of the bad intervals resulted from shooting the \( R_{C_k} \)'s independently onto the \( C_i \)'s is one contiguous interval (because all the intervals so resulted are subintervals of the crossing bad interval of \( R_C \) from \( C \)). Also, note that each \( UPCI(L(R_{C_k})) \) can be recovered from \( bd(UPCI(L(R_C))) \) and from the (at most) two connected components of \( bd(UPCI(L(R_{C_k}))) - bd(UPCI(L(R_C))) \) in the desired complexity bounds.

The computation for the root level of the recursion tree has been done in Phase 1.A, hence this phase begins with the child level of the root. It consists of three cases, i.e., Case b.1, Case b.2, and Case b.3, which correspond to the three cases in Phase 1.A, respectively. We only discuss Case b.1 and Case b.3 because Case b.2 is a simple case of Case b.3.

At a recursion level of Case b.3, let chain \( C = bd_{st} \) be partitioned into \( g = (m/d)^{1/3} \) subchains \( C_1, C_2, \ldots, C_g \), where \( |C| = m \) and \( s < t \). We do the following in \( O(\log m) \) time using \( O(m/d) \) processors.

1. If there is a \( C' \neq C \) such that the bad edges on \( C' \) are to be identified by using \( R_C \) (inductively, we assume that there is at most one such \( C' \) for \( C \) and \( |C'| = m \)), then for every \( R_{C_i} \) that is out-hitting to \( C \), do \( Test(L(R_{C_i}), s) \) for each segment \( s \) on \( ICP(C_j') \), where \( |ICP(C_j')| \) is not Case (a). Compute \( S_i \) and \( s_i \) (as defined in (3') of Case a.3, Phase 1.A).
(2) For each $R_{C_i}$ in step (1), if $s_i$ does not exist, then mark $C'$ as covered and store the bad interval $[a_i, b']$, where $a_i$ is the first endpoint of the crossing bad interval of $R_{C_i}$ from $C_i$ and $v_{b'}$ is the last vertex of $C'$. Otherwise, the computation on $R_{C_i}$ and $s_i$ is similar to (4') of Case a.3 in Phase 1.A (hence, either we no longer shoot $R_{C_i}$ onto any subchain of $C'$ or we may shoot $R_{C_i}$ onto a unique $C'_i$ of $C'$). Suppose $R_{C_i}$ is to be shot onto $C'_i$. If there is an $l$ such that $R_{C_i}$ is to be shot onto $C'_i$ and $l' > i'$, then store the bad interval $[a_i, b_{i'}]$, where $v_{b_{i'}}$ is the last vertex of $C'_i$; otherwise, shoot $R_{C_i}$ onto $C'_i$, at the next level below.

(3) If $C$ is covered, then mark every $C_i$ as covered and store the bad interval $[s, t]$. Otherwise, for every $R_{C_k}$ that is out-hitting to $C_k$ and in-hitting to $C$, if there is a unique subchain $C_{k'}$ ($\neq C_k$) of $C$ such that the bad edges on $C_{k'}$ are to be identified by using $R_{C_k}$, then shoot $R_{C_k}$ onto $C_{k'}$, at the next level below.

The correctness of the above procedure can be easily proved by induction on the levels of the recursion tree (the induction basis is provided by Case a.3 of Phase 1.A).

When $|C| \leq d$ (i.e., in Case b.1), we perform \[121\] on $C$ and $C'$ sequentially in $O(d)$ time, if the bad edges on $C'$ are to be identified by using $R_C$.

The recurrences of the complexity bounds for this phase are similar to those for Phase 1.A. Hence the time and processor bounds are $O(\log n)$ and $O(n/\log n)$, respectively.

5.3.3 Computing the Union of Bad Intervals

Suppose that a set of bad intervals, whose union contains $\mathcal{HE}^+(P)$, has been given (as the output from Phase 1.A and Phase 1.B). We denote this set of bad intervals by $\mathcal{Bad}^+$. Here we briefly show that the edges of $P$ contained in $\mathcal{Bad}^+$ can all be identified in $O(\log n)$ time using $O(n/\log n)$ processors.

The intervals in $\mathcal{Bad}^+$ are specified by the indices of their first and second endpoints. Furthermore, a sorted sequence of the first endpoints of the bad intervals along $bd(P)$ from $v_1$ counterclockwise to $v_n$ is easily obtained by a parallel prefix
Given this sorted sequence, we use a procedure with the same algorithmic outline as that of Phase 1.A to compute the union of $Bad^+$. The procedure partitions the set of bad intervals into subsets of equal size according to the first endpoints of the intervals, then, in parallel, computes the union of each subset, and finally combines the unions of the subsets together to obtain the union of $Bad^+$. The union of the bad intervals in each subset is represented by a rank tree. For example, suppose that $IN_1$ and $IN_2$ are two subsets of bad intervals such that the first endpoints of the intervals in $IN_1$ are all less than the first endpoints of the intervals in $IN_2$. In a conquer stage of the procedure, we face the following situation: the union of $IN_1$ and the union of $IN_2$ are available and at most one interval $I$ in the union of $IN_1$ (i.e., the most counterclockwise one) can intersect one or more of the most clockwise intervals in the union of $IN_2$. By doing a binary search on the union of $IN_2$ by using the second endpoint of $I$, we find the intersection of $I$ with the union of $IN_2$. Thus the union of $IN_1 \cup IN_2$ can be easily obtained from the union of $IN_1$ and the union of $IN_2$. This procedure is quite simple and will not be discussed in more details.

Given the union of $Bad^+$, $bd(P) - Bad^+$ is trivially computed in $O(\log n)$ time using $O(n/\log n)$ processors (by a parallel prefix). This takes care of $HE^+(P)$. Therefore, given $HE^+(P)$ and $HE^-(P)$, $WVE(P)$ is computable in the desired complexity bounds. This concludes the discussion on detecting the weak visibility of the star-shaped polygon $P$.

5.4 Checking the Weak Visibility of a Polygon from an Edge

This section deals with the following problem: check whether a simple polygon $P$ is weakly visible from a specified edge $e$ of $P$ (i.e., the case solved sequentially in [19, 61]). Using the parallel algorithm for computing the region inside a simple polygon that is weakly visible from a specified edge [72], this problem can be solved in $O(\log n)$ time using $O(n)$ CREW PRAM processors. We show how to solve this problem optimally in $O(\log n)$ time using $O(n/\log n)$ CREW PRAM processors.
Our solution consists of two parts: a preprocess and a two-phase procedure. The preprocess first reduces the problem to that of checking the weak visibility of a simple polygon from a convex edge and then further simplifies this problem. The two-phase procedure handles the (simplified) problem of checking the weak visibility of a polygon from a convex edge.

5.4.1 The Preprocess

WLOG, we assume that the specified edge is $e = \overline{v_nv_1}$, $e$ is on the $x$-axis, and $x(v_1) > x(v_n)$.

Let a horizontal ray $r^+$ (resp., $r^-$) start at $v_1$ (resp., $v_n$) and go to the right (resp., left) of $v_1$ (resp., $v_n$). If $r^+ \cap P$ (resp., $r^- \cap P$) consists of more than one connected component, then we conclude that $P$ is not weakly visible from $e$. Otherwise, let $r^+$ (resp., $r^-$) first-hit $bd(P) - e_n$ at a point $p'$ (resp., $p''$) on edge $e_a$ (resp., $e_b$). Point $p'$ (resp., $p''$) does not exist if $v_1$ (resp., $v_n$) is convex. WLOG, we assume that $p'$ (resp., $p''$) exists and is not a vertex of $P$. If segment $\overline{v_1v_{a+1}}$ (resp., $\overline{v_nv_b}$) intersects $bd_{a+1} - e_{a+1}$ (resp., $bd_{a+1} - e_{a-1}$), then it is not hard to see that $P$ is not weakly visible from $e_n$. Otherwise, $\overline{v_1v_{a+1}}$ and $\overline{v_nv_b}$ together partition $P$ into (at most) three subpolygons $P_1$, $P_2$, and $P_3$ (see Figure 5.6). If $p'$ (resp., $p''$) does not exist, then $P_1$
(resp., $P_3$) does not exist. This partition of $P$ is readily done in the desired complexity bounds.

Edge $e_n$ is now convex on $P_2$ and $l(e_n)$ is below $P_2 - e$. It is easy to see that $P$ is weakly visible from $e$ only if $P_1$ (resp., $P_3$) is star-shaped with $v_1$ (resp., $v_n$) in its kernel. Checking whether $P_1$ (resp., $P_3$) is visible from $v_1$ (resp., $v_n$) is easily done by a parallel prefix, because we only need to make sure that the polar angles of the vertices of $P_1$ (resp., $P_3$), with respect to $v_1$ (resp., $v_n$), are in sorted order as we walk along $bd(P_1)$ (resp., $bd(P_3)$) counterclockwise starting at $v_1$ (resp., $v_n$). Hence what remains to be done is to check whether $P_2$ is weakly visible from the convex edge $e$.

In the rest of this section, we assume that $P = P_2$ (see Figure 5.6) and $P$ still has $n$ vertices.

Next we simplify a little further the problem of checking the weak visibility of $P$ from the convex edge $e$ by testing each vertex of $P$ for certain local properties. The purpose of the tests is to make sure that $P$ is not too "spiral" and to make the computation of the internal convex paths in the two phases of the algorithm easier.

The tests are based on the simple observation below. Let $r(e_i)$ be the ray starting at $v_i$ and containing $e_i$. Then $P$ is not weakly visible from $e$ if for some $i$, $1 < i < n$, one of the following is true: (i) $e$ does not intersect the left half-plane of $r(e_i)$, or (ii) $\alpha(r(e_{i-1})) < \pi$, $\alpha(r(e_i)) > \pi$, and $v_i$ is nonconvex. Each of these tests is trivially done in $O(1)$ work. From now on, we assume that these tests have been done and $P$ is not known to be non-weakly visible from $e_n$.

5.4.2 The Basic Idea

We still use $ICP(bd_{ij})$ to represent the directed shortest path from $v_i$ to $v_j$ that goes through only the vertices of $bd_{ij}$ (i.e., the computation of $ICP(bd_{ij})$ is based only on $bd_{ij}$ and disregards $bd(P) - bd_{ij}$). Note that for $i < j$, it is not necessary for $ICP(bd_{ij})$ to make consistent right turns; furthermore, $ICP(bd_{ij})$ so computed may even intersect the exterior of $P$ (because it can intersect $bd(P) - bd_{ij}$). If this is indeed the case, then $P$ is not weakly visible from $e$ (see [19, 61]).
Unless otherwise specified, we assume that $i < j$ for every $bd_{ij}$ in the rest of this section.

The lemma below gives the basic idea for solving this weak visibility problem. For $i < i'$ (resp., $i > i''$), let $s_i(i')$ (resp., $s_i(i'')$) be the unique segment on $ICP(bd_{ii'})$ (resp., $ICP(bd_{ii''})$) that contains $v_i$.

**Lemma 5.10** If $P$ is weakly visible from $e_n$, then for any $i, j, k$, $1 \leq i < j < k \leq n$, a sweep of the interior angle of $P$ at $v_j$, from edge $e_{j-1}$ clockwise to edge $e_j$, encounters $e_{j-1}, s_j(i), s_j(k)$, and $e_j$, in that order (cf. Figure 5.7).

**Proof.** Since $P$ is weakly visible from $e_n$, there is a point $p$ on $e_n$ such that the segment $v_jp$ is contained in $P$ (Figure 5.7). Segment $v_jp$ clearly separates $e_{j-1}$ and $s_j(i)$ from $e_j$ and $s_j(k)$. Also, $s_j(i)$ (resp., $s_j(k)$) separates $e_{j-1}$ (resp., $e_j$) from $v_jp$. Hence the lemma follows.  

By Lemma 5.10, if $s_j(i)$ and $s_j(k)$ are not in correct order with $e_{j-1}$ and $e_j$ for some $i, j, k$, $i < j < k$, then $P$ is not weakly visible from $e$. If they are in correct order, then there is a ray (say, the one starting at $v_j$ and containing $s_j(i)$) separating $ICP(bd_{ij})$ from $ICP(bd_{jk})$. Let the ray starting at $v_j$ and containing $s_j(i)$ be denoted by $r(s_j(i))$. If $r(s_j(i))$ separates $ICP(bd_{ij})$ from $ICP(bd_{jk})$, then $ICP(bd_{ik})$...
can be computed efficiently from $ICP(bd_{ij})$ and $ICP(bd_{jk})$. This is the idea used in the recursive algorithm.

It is well-known that if there is a ray (e.g., $r(s_j(1))$) separating $ICP(bd_{ij})$ from $ICP(bd_{jn})$ for every $j, 1 < j < n$, then $P$ is weakly visible from $e_n$ (e.g., see [77]). Our ultimate goal, therefore, is to compute $s_j(1)$ and $s_j(n)$ for every $j$ and check their relative order with $e_{j-1}$ and $e_j$ based on Lemma 5.10. Clearly, our main problem is in computing all the $s_j(1)$'s and $s_j(n)$'s, since given $s_j(1)$ and $s_j(n)$ for every $j$, checking their relative order with $e_{j-1}$ and $e_j$ is trivially done in $O(1)$ work.

The algorithm for computing the $s_j(1)$'s and $s_j(n)$'s consists of two phases. Phase 2.A computes the internal convex paths for certain sub-chains of $bd(P)$ based on Lemma 5.10 (hence at every recursion level, either it succeeds in computing $ICP(C)$ for a sub-chain $C$ of $bd_{1n}$ from its sub-chains, or it concludes that $P$ is not weakly visible from $e$). After Phase 2.A is completed, if $P$ is still not known to be non-weakly visible from $e$, then the algorithm proceeds with Phase 2.B. For every $j$, Phase 2.B constructs $ICP(bd_{1j})$ and $ICP(bd_{jn})$ implicitly (by using the internal convex paths computed in Phase 2.A) and reports $s_j(1)$ and $s_j(n)$.

We need to characterize $ICP(bd_{ij})$ before going to the two-phase algorithm. We say $ICP(bd_{ij})$ is in order iff the vertices of $ICP(bd_{ij})$ visited in the walk along $ICP(bd_{ij})$ from $v_i$ to $v_j$ are in increasing order of vertex indices. For a segment $s = \overline{v_a v_b}, a < b$, let $r(s)$ be the ray starting at $v_a$ and containing $s$. The following lemmas enable us to compute the internal convex paths efficiently.

**Lemma 5.11** Suppose that $P$ is not known to be non-weakly visible from $e$ when $ICP(bd_{ij})$ has been computed. If $ICP(bd_{ij})$ is in order and makes only right turns, then for any two consecutive segments $\overline{v_a v_b}$ and $\overline{v_b v_c}$ on $ICP(bd_{ij})$, where $a < b < c$, $\alpha(r(\overline{v_b v_c})) < \alpha(r(\overline{v_a v_b}))$.

**Proof.** If $\alpha(r(\overline{v_a v_b})) \geq \pi$, then the lemma is obviously true because $ICP(bd_{ij})$ is in order and makes only right turns. If $\alpha(r(\overline{v_a v_b})) < \pi$, then either $\alpha(r(\overline{v_a v_b})) < \alpha(r(\overline{v_a v_b}))$ or $\alpha(r(\overline{v_b v_c})) > \alpha(r(\overline{v_a v_b})) + \pi$ (since $ICP(bd_{ij})$ makes only right turns). But $\alpha(r(\overline{v_b v_c})) \geq \alpha(r(\overline{v_a v_b})) + \pi$ and $ICP(bd_{ij})$ being in order imply that the local
Corollary 5.1 Suppose that $P$ is not known to be non-weakly visible from $e$ when $ICP(bd_{ij})$ has been computed. If $ICP(bd_{ij})$ is in order and makes only right turns, then the polar angles of the segments along $ICP(bd_{ij})$ are in sorted order.

Proof. An immediate consequence of Lemma 5.11. □

Lemma 5.12 Suppose that $P$ is not known to be non-weakly visible from $e$ when $ICP(bd_{ik})$ has been computed from $ICP(bd_{ij})$ and $ICP(bd_{jk})$. If both $ICP(bd_{ij})$ and $ICP(bd_{jk})$ are in order and make only right turns, then $ICP(bd_{ik})$ is in order and makes only right turns.

Proof. Since $P$ is not known to be non-weakly visible from $e$, it follows that $e_{j-1}$, $s_j(i)$, $s_j(k)$, and $e_j$ are in correct order (cf. Lemma 5.10). Hence there is a ray starting at $v_j$ that separates $ICP(bd_{ij})$ from $ICP(bd_{jk})$. Let $\overline{v_a v_b}$ be the common tangent between $ICP(bd_{ij})$ and $ICP(bd_{jk})$ such that both $ICP(bd_{ij})$ and $ICP(bd_{jk})$ are on the same side of line $l(\overline{v_a v_b})$, where $v_a$ (resp., $v_b$) is on $ICP(bd_{ij})$ (resp., $ICP(bd_{jk})$) (see Figure 5.8). (Possibly $v_a = v_b = v_j$.) Then $ICP(bd_{ik}) = ICP(bd_{ia}) \cup \overline{v_a v_b} \cup ICP(bd_{bk})$. 

Figure 5.8 Illustrating the proof of Lemma 5.12.
That \( ICP(bd_{ij}) \) is in order and makes only right turns follows from these properties of \( ICP(bd_{ij}) \) and \( ICP(bd_{jk}) \).

Lemma 5.13 Suppose that \( P \) is not known to be non-weakly visible from \( e \) when \( ICP(bd_{ij}) \) has been computed. If \( ICP(bd_{ij}) \) is in order and makes only right turns, then \( ICP(bd_{ij}) \) can be partitioned into at most two subpaths \( C' \) and \( C'' \) such that \( C' \) (resp., \( C'' \)) is completely on the convex hull of \( C' \) (resp., \( C'' \)).

Proof. Let \( l \) be the horizontal line tangent to \( ICP(bd_{ij}) \) such that \( ICP(bd_{ij}) \) is contained in the upper half-plane of \( l \), and let \( l \) touch \( ICP(bd_{ij}) \) at \( v_a \). By Corollary 5.1, the polar angles of the segments on \( ICP(bd_{ia}) \) (resp., \( ICP(bd_{aj}) \)) are all \( \geq 
\pi \) (resp., \( \leq \pi \)). Hence, \( ICP(bd_{ia}) \) (resp., \( ICP(bd_{aj}) \)) is completely on the convex hull of \( ICP(bd_{ia}) \) (resp., \( ICP(bd_{aj}) \)).

5.4.3 Phase 2.A

This phase consists of three cases: Case c.1, Case c.2, and Case c.3. Its algorithmic outline and recurrences for the time and processor complexities are similar to those of Phase 1.A. Given input \( (C, |C|, d) \) to each recursive call, where \( C \) is a subchain of \( bd_{1n} \), either an answer "no" (indicating that \( P \) is already known to be non-weakly visible from \( e \)), or \( ICP(C) \) (indicating that \( P \) is not known to be non-weakly visible from \( e \) in the computation of \( ICP(C) \)), is returned. All the internal convex paths in this phase are stored in rank trees. We now discuss these three cases one by one.

Case c.1 uses one processor to perform an algorithm in [19, 61] sequentially. For a chain \( C = bd_{st} \) of size \( d \), the algorithm checks the weak visibility of \( bd_{st} \) from \( e \), in \( O(d) \) time. If the check concludes that \( bd_{st} \) is not weakly visible from \( e \), then it reports a "no" to its parent node in the recursion tree of Phase 2.A (since certainly \( P \) is also not weakly visible from \( e \)); otherwise, it returns \( ICP(C) \) stored in a rank tree (\( ICP(C) \) is a by-product of the process that checks the weak visibility of \( bd_{st} \) from \( e \)).

Case c.2 computes, for a chain \( C = bd_{st} \), \( ICP(C) \) from its two subchains \( C_1 \) and \( C_2 \) of equal size. Let \( v_j \) be the common vertex of \( C_1 \) and \( C_2 \). If a "no" is returned
from the recursive call on either \( C_1 \) or \( C_2 \), or if the right half-plane of \( r(s_j(s)) \) or the left half-plane of \( r(s_j(t)) \) does not intersect \( e_n \), then return a “no” for \( C \). Otherwise, by using \( ICP(C_1) \) and \( ICP(C_2) \) (each stored in a rank tree), check the order of \( s_j(s), s_j(t), e_{j-1}, \) and \( e_j \) based on Lemma 5.10 (this is trivially done in \( O(\log |C|) \) time and one processor). If they are not in correct order, then \( P \) is not weakly visible from \( e \); hence a “no” is returned for \( C \). Otherwise, use one processor to compute the common tangent between \( ICP(C_1) \) and \( ICP(C_2) \) (based on Lemmas 5.12 and 5.13), and then obtain \( ICP(C) \) from \( ICP(C_1), ICP(C_2), \) and their common tangent. This is done in \( O(\log |C|) \) work by using [65]. \( ICP(C) \) is stored in a rank tree. Also, for each \( i = 1, 2, \) retain \( ICP(C_i) - ICP(C) \) in a rank tree. Observe that \( ICP(C_i) - ICP(C) \) consists of at most one connected component.

Case c.3 is more complicated. Chain \( C = bd_{st} \) is partitioned into \( g = (m/d)^{1/3} \) subchains \( C_1, C_2, \ldots, C_g \) of equal size, where \( |C| = m \). There are \( O(g^3) \) processors available. After the recursive calls for the \( C_i \)'s, if \( P \) is not known to be non-weakly visible from \( e \), then \( ICP(C_1), ICP(C_2), \ldots, ICP(C_g) \) are all available. Let \( v_{z_j} = C_j \cap C_{j+1} \) for \( j = 1, 2, \ldots, g - 1 \) (i.e., \( v_{z_j} \) is the common vertex of \( C_j \) and \( C_{j+1} \)). For each \( v_{z_j} \), let \( Bef_j \) (resp., \( Aft_j \)) denote the subchain of \( C \) before (resp., after) \( v_{z_j} \), that is, \( Bef_j = bd_{sz_j} \) (resp., \( Aft_j = bd_{jt} \)). One important operation in this case is to find \( s_{z_j}(s) \) and \( s_{z_j}(t) \) for each \( v_{z_j} \). This is because if \( s_{z_j}(s) \) and \( s_{z_j}(t) \) are in correct order with \( e_{z_{j-1}} \) and \( e_{z_j} \) (cf. Lemma 5.10) for each \( v_{z_j} \), then there is a ray separating every pair of \( ICP(C_i) \) and \( ICP(C_j) \) with \( i \neq j \), which makes the computation of \( ICP(C) \) efficient. Note that \( ICP(Bef_j) \) and \( ICP(Aft_j) \) are in general not explicitly available for the computation of \( s_{z_j}(s) \) and \( s_{z_j}(t) \). We only discuss how to use \( Aft_j \) to compute \( s_{z_j}(t) \) (the case for \( s_{z_j}(s) \) is similar).

The following lemma is useful in computing the \( s_{z_j}(t) \)'s.

Lemma 5.14 Suppose that \( P \) is not known to be non-weakly visible from \( e \) in computing the \( ICP(C_k)'s \) and that every \( ICP(C_k) \) is in order and makes only right turns. For a \( v_{z_j} \), let \( r \) be the ray starting at \( v_{z_j} \) and passing \( v_1 \), and let \( r' \) be the ray on line \( l(\overline{v_{z_j}v_n}) \) which starts at \( v_{z_j} \) and does not contain \( v_n \) (see Figure 5.9). For any \( i \in \)
Proof. When $ICP(C_i) - v_{z_j}$ intersects $lfp(r) \cap rtp(r')$, there are two cases to consider.

Case (1): there is a vertex $v_w$ ($\neq v_{z_j}$) of $ICP(C_i)$ that is in $lfp(r) \cap rtp(r')$ (see Figure 5.9 (a)). Let $r^*$ be the ray starting at $v_{z_j}$ and passing $v_w$. If $y(v_w) < y(v_{z_j})$, then $e_n$ does not intersect the left half-plane of $r^*$ (except possibly at $v_1$), implying that there is a point on $Aft_j$ (e.g., $v_{z_j}$ in Figure 5.9 (a)) that is not weakly visible from $e_n$. If $y(v_w) \geq y(v_{z_j})$, then $e_n$ does not intersect the left half-plane of $r^*$ (except possibly at $v_n$), implying that again there is a point on $Aft_j$ that is not weakly visible from $e_n$.

Case (2): no vertex of $ICP(C_i) - v_{z_j}$ is in $lfp(r) \cap rtp(r')$ and a segment $v_a v_b$ on $ICP(C_i)$ intersects both $r$ and $r'$, where $z_j < a < b$. Since $v_a v_b$ intersects both $r$ and $r'$, either $y(v_a) < y(v_{z_j}) < y(v_b)$ or $y(v_b) < y(v_{z_j}) < y(v_a)$. Hence chain $bd_{ab}$ cannot intersect $lfp(r) \cap rtp(r')$ by entering $lfp(r) \cap rtp(r')$ by crossing one of $r$ or $r'$ and then exiting from $lfp(r) \cap rtp(r')$ by crossing the other (because if it did, then the local condition (ii) would have been satisfied at a vertex of $bd_{1n}$ in the preprocess and we would have had known that $P$ is not weakly visible from $e$ in the preprocess).

Thus only $v_a v_b$ (but not $bd_{ab}$) crosses both $r$ and $r'$ (e.g., see Figure 5.9 (b)). Because $bd_{ab}$ does not cross both $r$ and $r'$ like $v_a v_b$ does, there must be an edge on $bd_{ab}$ that crosses segment $v_{z_j} v_n$. If $y(v_a) < y(v_{z_j}) < y(v_b)$ (see Figure 5.9 (b)), then there is a vertex $v_c$ in the interior of $bd_{ab}$ that is on $ICP(C_i)$, a contradiction to the assumptions that $ICP(C_i)$ is in order and makes only right turns and that $v_a v_b$ is a segment of $ICP(C_i)$. Thus we must have $y(v_b) < y(v_{z_j}) < y(v_a)$ (see Figure 5.9 (c)). But then $s_a(z_j)$ and $s_a(b)$ are not in correct order with $e_{a-1}$ and $e_a$ by Lemma 5.10; therefore, $P$ is not weakly visible from $e$. \hfill \Box

Observe that in Phase 2.A, if $P$ is not known to be non-weakly visible from $e$, then all the $ICP(C_k)$'s have the properties stated in Lemmas 5.11 and 5.12 (that all the $ICP(C_k)$'s are in order and make only right turns can be easily proved by induction.
Figure 5.9 Illustrating the proof of Lemma 5.14.
based on Case c.1 and Lemma 5.12). Therefore, we can readily check whether each
\( ICP(C_i) - v_z \) intersects \( lfp(r) \cap rtp(r') \) in \( O(\log m) \) time using \( O(g) \) processors.

We compute the \( s_z(t)'s \) as follows. We first do the checking based on Lemma 5.14
for every \( v_z \) (in \( O(\log m) \) time using \( O(g^2) \) processors). Suppose that \( P \) is not known
to be non-weakly visible from \( e \) after the checking, then we compute the common
tangent between \( v_z \) and \( ICP(C_i) \) for each \( C_i \subseteq Aft_j \), by using the ray starting at \( v_z \),
and containing \( \overline{v_1v_z} \) as a ray separating \( v_z \) from \( ICP(C_i) \). This is easily computed in
\( O(\log m) \) time using \( O(g) \) processors for each \( C_i \).

For a \( v_z \), suppose that among the \( O(g) \) common tangents so obtained, the tangent \( \overline{v_zv_b} \) between \( v_z \) and \( ICP(C_i) \) (\( v_b \) is on \( ICP(C_i') \)) is the one first encountered if we use a ray originating at \( v_z \) to sweep
around \( v_z \), clockwise, by starting at \( \overline{v_1v_z} \). Then \( s_z(t) \) is equal to \( \overline{v_zv_b} \). Therefore,
each \( s_z(t) \) is computable in \( O(\log m) \) time using \( O(g^2) \) processors.

Given \( s_z(s) \) and \( s_z(t) \) for each \( v_z \), if \( s_z(s) \) and \( s_z(t) \) are in correct order with
\( e_{z-1} \) and \( e_z \) (cf. Lemma 5.10), then the ray \( r(s_z(s)) \) (starting at \( v_z \) and containing
\( s_z(s) \)) separates \( Bef_j \) from \( Aft_j \), and hence it separates \( ICP(C_i) \) from \( ICP(C_k) \) for
each \( C_i \subseteq Bef_j \) and each \( C_k \subseteq Aft_j \). After \( r(s_z(s)) \) is available, we can compute the
common tangent between each pair of \( ICP(C_i) \) and \( ICP(C_k) \) in \( O(\log m) \) time using
\( O(g) \) processors (by using [65, 33]).

We are now ready to describe the procedure for Case c.3.

1. If a "no" is returned from any \( C_i \), then a "no" is returned for \( C \).

2. For every \( v_z \), compute \( s_z(s) \) and \( s_z(t) \). For any \( v_z \), if the computation for
\( s_z(s) \) or for \( s_z(t) \) concludes that \( P \) is not weakly visible from \( e \), or \( s_z(s) \) and
\( s_z(t) \) are not in correct order with \( e_{z-1} \) and \( e_z \), then return a "no" for \( C \).
(Otherwise, each \( r(s_z(s)) \) separates \( Bef_j \) from \( Aft_j \).)

3. For any \( v_z \), if the right half-plane of \( r(s_z(s)) \) or the left half-plane of \( r(s_z(t)) \)
does not intersect \( e \), then return a "no" for \( C \). Otherwise, by using all the
\( r(s_z(s))'s \), compute the common tangent between each pair of \( ICP(C_i) \) and
\( ICP(C_k) \), \( 1 \leq i < k \leq g \).
(4) By using the $r(s_j(s))'s$ and the set of the $O(g^2)$ common tangents obtained in step (3), build a complete binary tree of internal convex paths whose leaves are associated with $C_1, C_2, \ldots, C_g$, respectively. The root of the tree is associated with $bd_{st} = C$ and it stores $ICP(C)$ (in a rank tree).

We denote the complete binary tree of internal convex paths built in step (4) above by $T_C$. The root of $T_C$ is denoted by $root(T_C)$, and the left (resp., right) child of an internal node $u$ of $T_C$ is denoted by $lch(u)$ (resp., $rch(u)$). The information associated with each node of $T_C$ is as follows. The root $root(T_C)$ is associated with $C$ and it stores $ICP(C)$. For an internal node $u$, the subchain of $C$ associated with $u$ is the union of the subchains associated with the descendent leaves of $u$. Suppose that the subchain associated with $u$ is $bd_{ac}$ and the subchains associated with $lch(u)$ and $rch(u)$ are respectively $bd_{ab}$ and $bd_{bc}$. Observe that $ICP(bd_{ab}) - ICP(bd_{ac})$ (resp., $ICP(bd_{bc}) - ICP(bd_{ac})$) consists of at most one connected component. The information stored at $lch(u)$ (resp., $rch(u)$) is $ICP(bd_{ab}) - ICP(bd_{ac})$ (resp., $ICP(bd_{bc}) - ICP(bd_{ac})$), represented by a rank tree (see Figure 5.10). The height of $T_C$ is $O(\log m)$. This tree structure is used in [67] for triangulating a one-sided monotone polygon (in [67], each node of the tree stores a portion of the convex hull for its associated subchain; see [67] for the definitions and the details). The construction of $T_C$ is done by an algorithm.
in [67, 33], in $O(\log m)$ time using $O(g^3)$ processors, provided that the $r(s_{z_j}(s))$'s are already available.

In total, the procedure for Case c.3 requires $O(\log m)$ time and $O(g^3)$ processors.

Summing up the three cases, the time and processor complexities for Phase 2.A are respectively $O(\log n)$ and $O(n/\log n)$.

5.4.4 Phase 2.B

At the root of the recursion tree of Phase 2.A, if $P$ has been decided to be non-weakly visible from $e$, then the algorithm stops. Otherwise, we proceed with Phase 2.B to compute $s_j(1)$ and $s_j(n)$ for every $j$. WLOG, we just show how to compute the $s_j(1)$'s (the computation for the $s_j(n)$'s is similar).

In Phase 2.A, we have built a complete binary tree of the internal convex paths (cf. Case c.2 and Case c.3). We denote this tree by $T$. $T$ has $O(n/d) = O(n/\log n)$ nodes. The root of $T$ stores $ICP(bd_{1n})$. Each non-rooted node $u$ uses a rank tree to store at most one connected portion of the internal convex path for the subchain of $bd_{1n}$ that is associated with $u$. Hence there are totally $O(n/\log n)$ internal convex paths stored at the nodes of $T$.

5.4.4.1 The Algorithmic Structure and Main Operation

The algorithmic structure of Phase 2.B is the same as the structure of tree $T$. The procedure follows a top-down paradigm. It starts at $root(T)$, then goes to the two children of $root(T)$, and so on, level by level, until the leaves of $T$ are reached. Because $height(T) = O(\log n)$, we must process each level of $T$ (in most part of the procedure) in $O(1)$ time, in order to achieve an $O(\log n)$ time algorithm.

Let $u \neq root(T)$ and $u$ be an internal node of $T$. Let the chains associated with $u$, $lch(u)$, and $rch(u)$ be respectively $bd_{ac}$, $bd_{ab}$, and $bd_{bc}$, $1 < a < b < c$. The main operation of Phase 2.B is based on the lemma below. WLOG, we assume that up to the level of $u$ in $T$, the algorithm does not find that $P$ is not weakly visible from $e$. 
Lemma 5.15 Suppose that the common tangent between $ICP(bd_{1a})$ and $ICP(bd_{ac})$ touches $ICP(bd_{1a})$ at $v_q$ and $ICP(bd_{ac})$ at $v_r$, the common tangent between $ICP(bd_{1a})$ and $ICP(bd_{ab})$ touches $ICP(bd_{1a})$ at $v_w$ and $ICP(bd_{ab})$ at $v_z$, and the common tangent between $ICP(bd_{ab})$ and $ICP(bd_{bc})$ touches $ICP(bd_{ab})$ at $v_{x'}$ (see Figure 5.11). Then the common tangent between $ICP(bd_{1a})$ and every vertex on $bd_{ab}$ touches $ICP(bd_{1a})$ on $ICP(bd_{wa})$ (with $w \geq q$), and the common tangent between $ICP(bd_{1b})$ and every vertex on $bd_{bc}$ touches $ICP(bd_{1b})$ either on $ICP(bd_{qb}) = ICP(bd_{qw}) \cup \overline{v_wv_z} \cup ICP(bd_{zb})$ (when $v_r \in ICP(bd_{bc})$) or on $ICP(bd_{x'b})$ (when $v_r \in ICP(bd_{ab})$).

Proof. Since $P$ is not known to be non-weakly visible from $e$, the internal convex paths used in the algorithm are in order and make only right turns. Hence the lemma follows.

We call $ICP(bd_{qa})$ in Lemma 5.15 the left internal convex path to the chain associated with $u$ (i.e., $bd_{ac}$) and denote it by $LICP_u$. Likewise, $LICP_{lch(u)} = ICP(bd_{wa})$, and $LICP_{rch(u)} = ICP(bd_{qb})$ (when $v_r \in ICP(bd_{bc})$) and $LICP_{rch(u)} = ICP(bd_{x'b})$ (when $v_r \in ICP(bd_{ab})$). Observe that $LICP_{lch(u)}$ and $LICP_{rch(u)}$ are disjoint except possibly at $v_w$. For convenience, we also let $ICP_u$ denote $ICP(bd_{ac})$. 

Figure 5.11 Illustrating Lemma 5.15.

![Diagram of Lemma 5.15](image-url)
Suppose that $LICP_u ( = ICP(bd_{q_a}))$ and the common tangent between $LICP_u$ and $ICP_u$ are available at the beginning of the computation at node $u$. Then the following process performed at $u$ is based on Lemma 5.15.

**Process(u)**

(1) If $LICP_u = \emptyset$ or the common tangent between $LICP_u$ and $ICP_u$ touches $ICP_u$ on $ICP_{ich(u)}$, then let $LICP_{ich(u)} = LICP_u$, let $LICP_{rch(u)} = ICP(bd_{z'})$ by splitting $ICP_{ich(u)}$ at $v_2'$ (where $v_2'$ is on $ICP_{ich(u)}$ is an endpoint of the common tangent between $ICP_{ich(u)}$ and $ICP_{rch(u)}$, and $v_2'$ was computed in Phase 2.A), and go to (6).

(2) Compute the common tangent between $LICP_u$ and $ICP_{ich(u)}$. Let this common tangent touch $LICP_u$ at $v_w$ and $ICP_{ich(u)}$ at $v_x$ (see Figure 5.11).

(3) If $v_x = v_b$, then check the order among $s_b(1)$, $s_b(c)$, $e_b-1$, and $e_b$ (cf. Lemma 5.10). If they are not in correct order, then give a "no" to $lch(u)$ and $rch(u)$, and go to (6).

(4) Split $LICP_u$ at $v_w$ and split $ICP_{ich(u)}$ at $v_x$.

(5) Let $LICP_{ich(u)} = ICP(bd_{wa})$ and $LICP_{rch(u)} = ICP(bd_{qw}) \cup \overline{v_wv_x} \cup ICP(bd_{zw})$.

(6) Perform Process($lch(u)$) and Process($rch(u)$) at the next level, in parallel.

At the first level of Phase 2.B, we let $LICP_{root(T)} = \emptyset$. Suppose that the common tangent between $ICP(bd_{1(n/2)})$ and $ICP(bd_{(n/2)n})$ is $\overline{v_qv_r}$, where $v_q$ is on $ICP(bd_{1(n/2)})$. Then $LICP_{rch(root(T))} = ICP(bd_{q(n/2)})$, and $LICP_{ich(root(T))} = \emptyset$.

The main steps performed in Process($u$) are: (i) computing the common tangent between two internal convex paths, (ii) splitting an internal convex path into two subpaths, and (iii) combining two internal convex paths together to form an internal convex path. We need to perform each of these steps in $O(1)$ time for most part of the algorithm. Obviously, an appropriate data structure for representing the internal convex paths is essential in this phase.
5.4.4.2 The Representation of the Internal Convex Paths

In Phase 2.B, we can no longer use rank trees to represent the internal convex paths because their heights are logarithmic. Instead, we represent the internal convex path stored at each node of $T$ with an array. Therefore, before Phase 2.B is executed, we need to convert the rank tree representation for the internal convex paths stored in $T$ into the array representation. The conversion of the rank tree representation into the array representation for the internal convex paths requires an appropriate allocation of the processors to the internal convex path stored at each node of $T$. For a path of size $m$ stored at a node $u$ in $T$, we allocate $\max\{1, m/\log n\}$ processors to $u$. This processor allocation is easy to do in $O(\log n)$ time using $O(n/\log n)$ processors, because the size of each internal convex path is known (from its rank tree) and there are totally $O(n/\log n)$ paths in $T$. After this processor allocation, for each path in $T$, each processor copies a subpath of size $O(\log n)$ from the rank tree storing this path into a proper subarray of the array for this path; this is trivially done in $O(\log n)$ time. From now on, we assume that the internal convex path stored at each node of $T$ is represented by an array.

The process at a node $u$ of $T$ involves $ICP_u$ and $LICP_u$, both of which need to be represented in such a way that enables us to compute their common tangent in $O(1)$ time.

Recall that $lch(u)$ (resp., $rch(u)$) stores the portion of $ICP_{lch(u)}$ (resp., $ICP_{rch(u)}$) that is not on $ICP_u$. Only $root(T)$ has its internal convex path, i.e., $ICP(bd_{1n})$, stored in a single array. $lch(root(T))$ may have one portion of $ICP(bd_{1(n/2)})$ stored in $root(T)$. The left child of $lch(root(T))$ may have one portion of $ICP(bd_{1(n/4)})$ stored in $lch(root(T))$, which may again have a portion stored in $root(T)$. In general, a node $u$ may have a portion of $ICP_u$ stored in each of its ancestors in $T$. That is, $ICP_u$ is obtained from the $O(\log n)$ arrays stored at its ancestors. Therefore, we represent $ICP_u$ by using $O(\log n)$ subarrays.

Let $ICP_u$ be represented by $A_u(1), A_u(2), \ldots, A_u(k)$, in order, where each $A_u(i)$ is a subarray of an array stored at an ancestor of $u$. Each $A_u(i)$ is specified by two
pointers, one pointing to the first element of \( A_u(i) \) and the other pointing to the last element of \( A_u(i) \). Suppose those \( 2k \) pointers are available in the beginning of \( \text{Process}(u) \) for an internal node \( u \) of \( T \). In \( \text{Process}(u) \), we split \( \text{lCP}_u \) at \( v_{z'} \), where \( v_{z'} \) (resp., \( v_{z''} \)) on \( \text{lCP}_{\text{rch}(u)} \) (resp., \( \text{lCP}_{\text{ch}(u)} \)) is the endpoint of the common tangent between \( \text{lCP}_{\text{rch}(u)} \) and \( \text{lCP}_{\text{ch}(u)} \), as follows. Let \( v_{z'} \) be contained in \( A_u(i) \) for some \( i \). Then \( A_u(i) \) is split at \( v_{z'} \) into two subarrays \( A'_u \) and \( A''_u \), such that \( v_{z'} \) is the last element of \( A'_u \) and \( v_{z''} \) is the first element of \( A''_u \). Let the representation of \( \text{lCP}_{\text{rch}(u)} \) be the union of \( \text{lCP}_{\text{rch}(u)} \cap \text{lCP}_u \) (represented by \( O(\log n) \) pieces from \( u \)) and \( \text{lCP}_{\text{rch}(u)} - \text{lCP}_u \) (one single piece stored at \( \text{rch}(u) \)). That is, \( \text{lCP}_{\text{rch}(u)} \) is represented by \( A_{u(1)}, A_{u(2)}, \ldots, A_{u(i-1)}, A'_u, \) and \( B_{\text{rch}(u)} \), in order, where \( B_{\text{rch}(u)} \) is the array representing \( \text{lCP}_{\text{rch}(u)} - \text{lCP}_u \). The similar thing is done for \( \text{lCP}_{\text{ch}(u)} \).

We associate with \( \text{lCP}_u \) \( k \) size parameters \( \text{size}_u(1), \text{size}_u(2), \ldots, \text{size}_u(k) \), where \( \text{size}_u(i) = |A_{u(1)}| + \cdots + |A_{u(i)}| \). Using the size parameters, we can quickly access the \( j \)-th vertex on \( \text{lCP}_u \) for any \( j \) (to be shown in the next subsubsection). When \( \text{lCP}_u \) is split to form \( \text{lCP}_{\text{ch}(u)} \) and \( \text{lCP}_{\text{rch}(u)} \), the size parameters for the representation of \( \text{lCP}_{\text{rch}(u)} \cap \text{lCP}_u \) can be easily updated (in \( O(1) \) time using \( O(\log n) \) processors) because we just need to subtract/add a same number from/to all the parameters in the list for \( \text{rch}(u) \) and then add a new parameter (for \( \text{lCP}_{\text{rch}(u)} - \text{lCP}_u \)) to the beginning of the list. The update on the parameters for \( \text{ch}(u) \) is even easier (only a new term is added to the end of the parameter list).

The representation for \( \text{lICP}_u \) is the same as \( \text{lCP}_u \). Hence the "split" and "combine" steps on \( \text{lICP}_u \) are also the same as on \( \text{lCP}_u \). We just need to show that \( \text{lICP}_u \) can likewise be represented by \( O(\log n) \) subarrays, as follows. The number of subarrays for \( \text{lICP}_{\text{ch}(u)} \) cannot be larger than that for \( \text{lICP}_u \) because \( \text{lICP}_{\text{ch}(u)} \subseteq \text{lICP}_u \) (cf. Lemma 5.15). For \( \text{rch}(u) \), either \( \text{lICP}_{\text{rch}(u)} = \{v_{z'}\} \cup (\text{lCP}_{\text{ch}(u)} - \text{lCP}_u) \) (where \( v_{z'} \) on \( \text{lCP}_{\text{ch}(u)} \) is the endpoint of the common tangent between \( \text{lCP}_{\text{ch}(u)} \) and \( \text{lCP}_{\text{rch}(u)} \)), or \( \text{lICP}_{\text{rch}(u)} = \text{lICP}_u \cup \{v_z\} \cup \text{lICP}_{\text{ch}(u)}' \) (where \( \text{lICP}_u \subseteq \text{lICP}_u \), \( \text{lICP}_{\text{ch}(u)}' \subseteq (\text{lCP}_{\text{ch}(u)} - \text{lCP}_u) \), and \( v_z \) on \( \text{lCP}_{\text{ch}(u)} \) is the endpoint of the common tangent between \( \text{lICP}_u \) and \( \text{lCP}_{\text{ch}(u)} \)) (see Figure 5.11). In the former case, the number
of subarrays for \( \text{LlCP}_{\text{rch}(u)} \) is two (\( \{v_x\} \) is a subarray of a single element). In the latter case, at most two more subarrays are added to the representation of \( \text{LlCP}'_u \) to form \( \text{LlCP}_{\text{rch}(u)} \). Therefore, at each level of \( T \), the number of subarrays in the \( \text{LlCP} \) representation can increase by at most two.

5.4.4.3 The Procedure for Phase 2.B

Phase 2.B consists of three cases, which are based on the size of the chain \( C_u \) associated with a node \( u \) of \( T \). The outline of the procedure at node \( u \) is as follows.

Input. \( \text{LlCP}_u, \text{ICP}_u, m = |C_u|, \) and \( n = |P| \).

Case d.1. When \( m \leq \log n \), use \( (1 + |\text{LlCP}_u|/\log n) \) processors to perform the computation on \( C_u \) in \( O(\log n) \) time.

Case d.2. When \( \log n < m \leq \log^3 n \), use \( (1 + |\text{LlCP}_u|/\log n) \) processors to perform Process(\( u \)) in \( O((\log \log n)^2) \) time. Then in the next stage, recursively perform Process(\( \text{rch}(u) \)) and Process(\( \text{lch}(u) \)), in parallel.

Case d.3. When \( m > \log^3 n \), use \( O((m + |\text{LlCP}_u|)/\log n) \) processors to perform Process(\( u \)) in \( O(1) \) time. Then in the next stage, perform Process(\( \text{lch}(u) \)) and Process(\( \text{rch}(u) \)) recursively, in parallel.

The procedure is initially called on root(\( T \)) with input (\( \emptyset, \text{ICP}(\text{bd}_{1n}), n - 1, n \)).

Because Case d.3 is repeated \( O(\log n) \) times, Case d.2 is repeated \( O(\log \log n) \) times, and Case d.1 is done once, the time complexity for the procedure is \( O(\log n + (\log \log n)^3) = O(\log n) \), if we perform each of the three cases within the claimed time bound.

We allocate processors to the nodes of \( T \) by using two different schemes, one for the \( \text{ICP}_u \)'s and the other for the \( \text{LlCP}_u \)'s. When performing Process(\( u \)) for a node \( u \) in \( T \), \( |C_u|/\log n \) processors are allocated to \( \text{ICP}_u \), and \( |A_u(i)|/\log n \) processors are available from each subarray \( A_u(i) \) in the representation of \( \text{LlCP}_u \). The total number of processors used for the \( \text{ICP}_u \)'s at every level of \( T \) is clearly \( O(n/\log n) \).
The allocation of processors to the LICP \( u \)'s is done by preassigning processors to the array that represents the internal convex path stored at each node of \( T \), according to the size of the array. Since the total sum of the sizes of the arrays stored in \( T \) is \( O(n) \), the amount of preassigned processors is \( O(n / \log n) \). At every level of \( T \), because the LICP paths are disjoint except possibly at their endpoints (e.g., see Lemma 5.15), the number of processors available from each subarray \( A_u(i) \) in the representation of \( LICP_u \) is \( |A_u(i)| / \log n \). Therefore, the total number of processors used by the algorithm is \( O(n / \log n) \).

What remains to be shown is how to perform each of the above three cases in the desired time bound using the available amount of processors. We first discuss Case d.3 and Case d.2, and then discuss Case d.1.

Recall that one of the main steps involved in Process(\( u \)) is to compute the common tangent between two internal convex paths. The following lemma is used by both Case d.3 and Case d.2 in computing the common tangent.

Lemma 5.16 (Atallah and Goodrich [18]) Let \( S_1 \) and \( S_2 \) be two point sets separated by a line, with \( |S_1| + |S_2| = O(m) \). If the convex hulls of \( S_1 \) and \( S_2 \) are respectively stored in two arrays, then the common tangent between the convex hulls of \( S_1 \) and \( S_2 \) can be computed in \( O(1) \) time using \( m^{1/c} \) CREW PRAM processors, where \( c > 1 \) is a positive constant.

Proof. See Theorem 1 and Algorithm A in [18].

When performing Process(\( u \)) in Case d.3, we let the first \( O(\log n) \) processors allocated to \( ICP_u \) keep the representations for \( ICP_u \) and \( LICP_u \) (each processor keeps a size parameter and the two pointers for a subarray). Although neither \( ICP_u \) nor \( LICP_u \) is stored in a single array, using \( O((\log n)^{1/2}) \) processors, accessing the \( j \)-th vertex on \( ICP_u \) (resp., \( LICP_u \)) can be easily done in \( O(1) \) time (by using the size parameters, we can quickly determine which subarray \( A_u(i) \) contains the \( j \)-th vertex). Hence, we simulate each access to an array element in Lemma 5.16 by using \( O((\log n)^{1/2}) \) processors. Note that in Case d.3, there are always enough processors for this simulation of Lemma 5.16. Using \( O(\log n) \) processors, the other two main
steps in Process\(u\) (i.e., the “split” and “combine”) can be easily done in \(O(1)\) time (as shown in Subsubsection 5.4.4.2). At the end of Process\(u\), the ICP and LICP representations for the children of \(u\) are copied to the first \(O(\log n)\) processors allocated to the ICP’s of these children.

In Case d.2, we represent each of ICP\(_u\) and LICP\(_u\) by a rank tree of height \(O(\log \log n)\) (each leaf of the rank tree stores the two pointers for a subarray). Thus each access to the \(i\)-th vertex on ICP\(_u\) (resp., LICP\(_u\)) requires \(O(\log \log n)\) time and one processor. The “split” and “combine” steps are easily done in \(O(\log \log n)\) time and one processor. There are \((1 + |LICP_u|/\log n)\) processors available. If \(|LICP_u| > \log^3 n\), then Lemma 5.16 is simulated using \(O(|LICP_u|/\log n) > \log^2 n\) processors, in \(O(\log \log n)\) time; otherwise, simply use one processor to compute the common tangent by the sequential algorithm in [112], in \(O((\log \log n)^2)\) time.

Suppose that at the end of Case d.2, \(P\) is not known to be non-weakly visible from \(e\). Before switching from Case d.2 to Case d.1, we need to convert the subarray representation of LICP\(_u\) into a single array representation, for each leaf \(u\) of \(T\). Because there are \(O(n/\log n)\) leaves in \(T\) and the size of each LICP\(_u\) is known, this conversion of the representations for the LICP\(_u\)’s can be trivially done in \(O(\log n)\) time using \(O(n/\log n)\) processors. After the conversion, we allocate \((1 + |LICP_u|/\log n)\) processors to each leaf \(u\) of \(T\) and do the following. (Note that \(|C_u| \leq \log n\).) (1) Partition the single array containing LICP\(_u\) into \(|LICP_u|/\log n\) subarrays \(B_u(1), B_u(2), \ldots\), of size \(\log n\) each (every \(B_u(i)\) contains a subpath of LICP\(_u\)). (2) In parallel, perform an algorithm in [19, 61] on each pair of \(B_u(i)\) and \(C_u\) (in \(O(\log n)\) time and using one of the \((1 + |LICP_u|/\log n)\) processors). (3) If a pair of \(B_u(k)\) and \(C_u\) reports that \(P\) is not weakly visible from \(e\), then we are done; otherwise, each pair of \(B_u(i)\) and \(C_u\) gives a candidate of \(s_j(1)\) for every \(v_j\) on \(C_u\). (4) For every \(v_j\) on \(C_u\), find \(s_j(1)\) from the \(O(|LICP_u|/\log n)\) candidates, in \(O(\log |LICP_u|)\) time and \(O(|LICP_u|/\log n)\) work. Therefore, in Case d.1, the computation on every \(C_u\) is done in \(O(\log n)\) time using \(O(1 + |LICP_u|/\log n)\) processors.

In conclusion, Phase 2.B runs in \(O(\log n)\) time using \(O(n/\log n)\) processors.
5.5 Detecting the Weak Visibility of a Simple Polygon

This section reduces the problem of detecting the weak visibility of a simple polygon $P$ to the two problems solved in Sections 5.3 and 5.4. The idea of the reduction is derived from the one used in [121], but our reduction procedure is very different from [121]. We first briefly review the reduction in [121], and then describe our reduction procedure.

5.5.1 The Reduction by Sack and Suri

In [121], the problem of detecting the weak visibility of a simple polygon is reduced to the following two problems: (i) computing all the weakly visible edges of a polygon that is known to be weakly visible from a specified edge, and (ii) checking whether a polygon is weakly visible from a specified edge. The reduction in [121] is as follows.

An arbitrary vertex of $P$, say $v_1$, is chosen, and the visible region of $P$ from $v_1$ is computed. Let the visible region of $P$ from $v_1$ be $V(v_1)$. $V(v_1)$ is obviously star-shaped. For a vertex $v'_a$ of $V(v_1)$, where $v'_a$ is on edge $e_a$ of $P$ ($v'_a$ may not be a vertex of $P$), $v_1v'_a$ is completely contained in $V(v_1)$. If $v_1v'_a$ contains a vertex $v_b$ of $P$ in its interior, then the region in $P$ enclosed by $v'_av_b$ and the subchain of $bd(P)$ which connects $v'_a$ and $v_b$ and does not contain $v_1$ is called a pocket of $P$ (with respect to $v_1$); furthermore, if $b < a$ (resp., $b > a$), then it is a right (resp., left) pocket. For example, in Figure 5.12, the pocket to the right of $v_bv'_a$ is a right pocket and the pocket to the left of $v_cv'_d$ is a left pocket.

Suppose that we walk along $bd(V(v_1))$ counterclockwise starting at $v_1$, and we label the right pockets visited during the walk by $r_1, r_2, \ldots, r_s$ and the left pockets by $l_1, l_2, \ldots, l_t$. The sequence of the $r_i$'s and $l_j$'s visited in the walk belongs to one of the three cases below:

(i) the 0-switch case, of the form $(r_1, \ldots, r_s, l_1, \ldots, l_t)$;

(ii) the 1-switch case, of the form $(r_1, \ldots, r_{s-1}, l_1, r_s, l_2, \ldots, l_t)$;
(iii) the *multiple-switch* case, of any other form.

It is shown in [121] that if it is of the multiple-switch case, then $WVE(P)$ is empty; if it is of the 1-switch case, then $|WVE(P)| \leq 3$ and the three candidate edges of $WVE(P)$ can be easily identified (given the three candidates of $WVE(P)$, the problem then becomes the one of checking the weak visibility of $P$ from a specified edge). The difficult case, therefore, is the 0-switch case.

To solve the 0-switch case, $P$ is partitioned into three regions as follows. Let $e'$ (resp., $e''$) be the edge of $V(v_1)$ that defines the *last* right pocket $r_s$ (resp., the *first* left pocket $l_i$) of $P$. Let $s_r$ (resp., $s_l$) be the maximal segment in $V(v_1)$ that contains $e'$ (resp., $e''$). Then $s_r$ and $s_l$ both have $v_1$ as an endpoint and they divide $P$ into three regions $L_P$, $M_P$, and $R_P$ (see Figure 5.12). Obviously, $M_P$ is star-shaped with $v_1$ in its kernel (hence $M_P$ is weakly visible from each of the edges that contains $v_1$).
Every edge of $WVE(P)$ must intersect $M_P$. For those edges of $P$ that intersect either $s_l$ or $s_r$ (there are only $O(1)$ of them and they can be easily identified), test whether they are in $WVE(P)$ by checking the weak visibility of $P$ from each of them. To compute those edges of $WVE(P)$ that are in the interior of $M_P$, denoted by $WVE_{M_P}(P)$, the weak visibility of $L_P$ (resp., $R_P$) from $s_l$ (resp., $s_r$) is first checked. If either check returns an answer "no", then $WVE_{M_P}(P)$ is empty. Otherwise, compute $WVE(M_P)$, $WVE(L_P \cup M_P)$, and $WVE(R_P \cup M_P)$, from which $WVE_{M_P}(P)$ can be easily obtained.

5.5.2 Our Reduction Procedure

We first choose a convex vertex of $P$. WLOG, let this vertex be $v_1$. Note that there is always a convex vertex in $P$ (otherwise, the sum of the interior angles of $P$ would be $> n\pi$, a contradiction). We then compute $V(v_1)$ by using the algorithm by Atallah et al. [15] (see Chapter 3), in $O(\log n)$ time and $O(n/\log n)$ processors. The sequence of right and left pockets of $P$ is easily obtained by a parallel prefix along the list of vertices of $V(v_1)$. If the sequence is of the multiple-switch case, then $WVE(P) = \emptyset$. If it is of the 1-switch case, then $WVE(P)$ is computed by applying the algorithm in Section 5.4 at most three times. Therefore, WLOG, we assume that the sequence is of the 0-switch case.

Next, we partition $P$ into $L_P$, $M_P$, and $R_P$, as in [121] (see Figure 5.12). The edges of $WVE(P)$ that intersect either $s_l$ or $s_r$ can be computed by the algorithm in Section 5.4, in $O(\log n)$ time using $O(n/\log n)$ processors. Therefore, we focus on the computation of $WVE_{M_P}(P)$ (the edges of $WVE(P)$ in the interior of $M_P$). WLOG, we assume that the interior of $M_P$ contains at least one edge of $P$ and that $L_P$ (resp., $R_P$) is weakly visible from $s_l$ (resp., $s_r$). (The weak visibility of $L_P$ (resp., $R_P$) from $s_l$ (resp., $s_r$) can be checked by the algorithm in Section 5.4.)

Because $M_P$ is star-shaped with convex vertex $v_1$ in its kernel, $WVE(M_P)$ is computed by using the algorithm in Section 5.3. Hence we only need to eliminate the
edges in \( WVE(M_P) \) that are bad with respect to the rays from \( L_P \) and the rays from \( R_P \). We only show the computation using \( L_P \) (the computation using \( R_P \) is similar).

We label counterclockwise the vertices of \( M_p \) as \( w_1, w_2, \ldots, w_m \), where \( w_1 = v_1 \) and \( w_m = v'_d \) (cf. Figure 5.12). The edge \( w_iw_{i+1} \) of \( M_P \) is denoted by \( e_i(M_P) \). Also, we denote \( bd(M_P) - e_m(M_P) \) by \( bd_1m(M_P) \) and \( M_P \cap L_P \) by \( s_i^* \). Note that \( s_i^* \subseteq s_i \). WLOG, we assume that \( v_1 \) is at the origin and \( s_i \) is on the positive \( y \)-axis.

The problem then is to shoot the rays from \( L_P \) (i.e., \( Ray^+(L_P) \) and \( Ray^-(L_P) \)) onto \( bd_1m(M_P) \).

We only consider the rays from \( L_P \) that intersect \( s_i^* \) (these rays may hit \( bd(L_P) - s_i^* \) before intersecting \( s_i^* \)). The rays from \( L_P \) that do not intersect \( s_i^* \) cannot hit \( M_P \) and thus have no effect on eliminating the bad edges in \( WVE(M_P) \). For each \( r \in Ray^+(L_P) \cup Ray^-(L_P) \), whether \( r \) intersects \( s_i^* \) or not is easily decided in \( O(1) \) work.

From now on, we assume that \( Ray^+(L_P) \) and \( Ray^-(L_P) \) consist of only the rays that intersect \( s_i^* \).

The following lemmas are useful in our reduction.

Lemma 5.17 \( M_P - s_i \) and \( L_P - s_i \) are on the opposite sides of line \( l(s_i) \).

Proof. The facts that \( M_P \) is visible from \( v_1 = w_1 \), that \( v_1 \) is convex in \( P \), and that \( e_m(M_P) \) is on \( l(s_i) \), together imply that \( w_1 \) is convex in \( M_P \) and that \( M_P - s_i \) is to the right of \( l(s_i) \). Because \( L_P \) is weakly visible from \( s_i^* \), \( L_P - s_i \) is to the left of \( l(s_i) \) if \( s_i^* \) is a convex edge of \( L_P \). Vertex \( w_m \) is an endpoint of \( s_i^* \) and it is a convex vertex in \( L_P \) since the interior of \( M_P \) contains at least one edge of \( P \). If \( w_1 \) is an endpoint of \( s_i^* \), then \( w_1 \) is a convex vertex in \( L_P \) because \( v_1 \) is convex in \( P \). If \( w_1 \) is not an endpoint of \( s_i^* \), then \( v_n \) must be an endpoint of \( s_i^* \). The edges of \( L_P \) adjacent to \( v_n \) are all on the left pocket defined by \( s_i^* \); hence \( v_n \) is a convex vertex in \( L_P \).

Lemma 5.18 If a ray \( r \in Ray^+(L_P) \cup Ray^-(L_P) \) intersects \( bd_1m(M_P) \) at a point \( p \) on \( e_j(M_P) \), then segment \( \overline{vp} \) does not intersect \( bd_1j(M_P) - w_j \) and \( r - \overline{vp} \) does not intersect \( bd(j+1)m(M_P) - w_{j+1} \), where \( v \) is the starting vertex of \( r \).
Proof. By Lemma 5.17, \( v \in L_p \) and \( p \in M_P \) are on the opposite sides of \( l(s_l) \). The lemma then follows from that \( M_P \) is visible from the convex vertex \( w_1 \) and that \( r \) intersects \( s_l^* \).

Because \( L_p \) is weakly visible from the convex edge \( s_l^* \), the dominating ray set \( DR^+(L_P) \) (resp., \( DR^-(L_P) \)) for \( Ray^+(L_P) \) (resp., \( Ray^-(L_P) \)) is characterized by Lemma 5.5 (i.e., the edges on \( bd_{1m}(M_P) \) that are bad with respect to \( Ray^+(L_P) \) (resp., \( Ray^-(L_P) \)) can be computed by using \( DR^+(L_P) \) (resp., \( DR^-(L_P) \)) only). Note that \( DR^+(L_P) \) (resp., \( DR^-(L_P) \)) is obtained by a parallel prefix.

The lemmas below are the counterparts of the lemmas in Subsubsection 5.3.1.2. They ensure that the computation of shooting the rays of \( DR^+(L_P) \cup DR^-(L_P) \) onto \( bd_{1m}(M_P) \) can be done by using the lines-vs-segment tests. (See Subsubsection 5.3.1.2 for the notations used in these lemmas.)

**Lemma 5.19** A bad-segment test on \( DR^+(L_P) \) (resp., \( DR^-(L_P) \)) and a segment \( s \) of \( ICP(C') \) can be done by using \( Test(L(\text{DR}^+(L_P)), s) \) (resp., \( Test(L(\text{DR}^-(L_P)), s) \)), where \( C' \) is a subchain of \( bd_{1m}(M_P) \).

Proof. For each ray \( r \in DR^+(L_P) \cup DR^-(L_P) \) and a segment \( s \) of \( ICP(C') \), since the starting vertex of \( r \) (in \( L_P \)) and \( s \) (in \( M_P \)) are on the opposite sides of the vertical line \( l(s_l) \) (by Lemma 5.17), \( s \) is below (resp., above, intersected by) \( r \) iff \( s \) is below (resp., above, intersected by) \( l(r) \). Hence the lemma holds. 

Observe that for each segment \( s \) on \( ICP(C') \) (where \( C' \subseteq bd_{1m}(M_P) \)), either 0 \( \leq \alpha(s) < 3\pi/2 \) or \( 3\pi/2 < \alpha(s) < 2\pi \), and for each \( r \in DR^+(L_P) \cup DR^-(L_P) \), either 0 \( \leq \alpha(r) < \pi/2 \) or \( 3\pi/2 < \alpha(r) < 2\pi \) (because the starting vertex of \( r \) is to the left of \( l(s_l) \) and \( r \) intersects \( s_l^* \)).

**Lemma 5.20** Suppose that a ray \( r \in DR^+(L_P) \cup DR^-(L_P) \) intersects a segment \( s \) on \( ICP(C') \) for a subchain \( C' \) of \( bd_{1m}(M_P) \). Then \( r \) pseudo-hits \( s \) iff \( \alpha(s) \neq \pi/2 \) and

1. \( \pi/2 < \alpha(s) \leq \pi \) and \( 3\pi/2 < \alpha(r) < \pi + \alpha(s) \), or
2. \( \pi < \alpha(s) < 3\pi/2 \) and \( (3\pi/2 < \alpha(r) \) or \( \alpha(r) < \alpha(s) - \pi) \), or
3. \( 3\pi/2 < \alpha(s) < 2\pi \) and \( (\pi/2 < \alpha(r) \) or \( \alpha(s) < \alpha(r)) \), or
(4) \(0 \leq \alpha(s) < \pi/2\) and \(\alpha(s) < \alpha(r) < \pi/2\).

Proof. When \(\alpha(s) = \pi/2\), \(r\) cannot pseudo-hit \(s\) because the starting vertex \(v\) of \(r\) is to the left of \(l(s)\) and \(s \subset M_p\) is to the right of \(l(s)\). We only show case (1) because the other three cases are proved similarly. In case (1), \(\pi/2 < \alpha(s) \leq \pi\). If \(r\) pseudo-hits \(s\), then \(v\) is in the interior of the right half-plane of \(r(s)\). But when \(v\) is in the interior of the right half-plane of \(r(s)\), \(r\) cannot intersect \(s\) if \(\pi + \alpha(s) \leq \alpha(r)\) or \(\alpha(r) < \alpha(s)\). Hence we must have \(3\pi/2 < \alpha(r) < \pi + \alpha(s)\). If \(3\pi/2 < \alpha(r) < \pi + \alpha(s)\), then \(v\) must be in the interior of the right half-plane of \(r(s)\) (because otherwise, \(r\) would not intersect \(s\), a contradiction). Hence \(r\) pseudo-hits \(s\). \(\square\)

Suppose that UPCLI\((L(DR^+(L_p)))\), LPCI\((L(DR^-(L_p)))\), and ICP\((bd_{1m}(M_p))\) are already available (they can all be constructed in the desired complexity bounds as shown in Subsubsection 5.3.1.1). Based on Lemma 5.18, eliminating the bad edges in \(WVE(M_p)\) using \(DR^-(L_p)\) can be done by simply using the procedure for Phase 1.B in Section 5.3. This is because, to \(DR^-(L_p)\), Lemma 5.18 and Lemma 5.1 are equivalent, i.e., the first-hit point \(h_i^-\) of a ray \(r_i^+ \in DR^+(L_p)\) (if \(h_i^-\) is on \(bd_{1m}(M_p)\)) is the first intersection point between \(bd_{1m}(M_p)\) and \(r_i^-\) encountered as we walk along \(bd_{1m}(M_p)\) clockwise starting from \(w_m\). (This fact is stated in Observation 2 of [121].) The problem we face here is, however, that \(DR^+(L_p)\) does not satisfy Lemma 5.1. This is because Lemma 5.18 implies that the first-hit point \(h_i^+\) of a ray \(r_i^+ \in DR^+(L_p)\) (if \(h_i^+\) is on \(bd_{1m}(M_p)\)) is the first intersection point between \(bd_{1m}(M_p)\) and \(r_i^-\) encountered as we walk along \(bd_{1m}(M_p)\) clockwise starting from \(w_m\) (but Lemma 5.1 uses a counterclockwise walk from \(w_1\)). Therefore, we need to discuss how to eliminate the bad edges in \(WVE(M_p)\) using \(DR^+(L_p)\).

The procedure below removes from \(WVE(M_p)\) the bad edges with respect to \(DR^+(L_p)\). Let point \(p^*\) on \(e_c(M_P) - w_c\) be the most counterclockwise first-hit point by the rays in \(DR^+(L_p)\) (\(p^*\) being the most counterclockwise first-hit point means that the subchain of \(bd_{1m}(M_p)\) from \(p^*\) counterclockwise to \(w_m\) does not contain any other first-hit point by the rays in \(DR^+(L_p))\). The procedure either correctly locates \(p^*\), or it makes sure that the edges on chain \(bd_{1c}(M_P)\) are all bad (without
knowing exactly where \( p^* \) is). This procedure is recursively called at most three times. After that, the problem size becomes small enough to be handled once for all by using \( O(n/ \log n) \) processors in \( O(\log n) \) time. As in Section 5.3, the procedure makes use of \( UPCI(L(DR^+(L_P))) \) and \( ICP(bd_{1m}(M_P)) \). The initial input subchain \( C \) of \( bd_{1m}(M_P) \) to the procedure is \( bd_{1m}(M_P) \) itself.

Let chain \( C = bd_{st}(M_P) \) be partitioned into \( n^{1/3} \) subchains \( C_1, C_2, \ldots, C_{n^{1/3}} \), of equal size, where \( |C| > n^{1/3} \) and \( s < t \). Let \( DR^+(L_P) \) be also partitioned into \( n^{1/3} \) subsets \( R_1, R_2, \ldots, R_{n^{1/3}} \), of equal size. The procedure shoots \( DR^+(L_P) \) onto \( C \) in \( O(\log n) \) time using \( O(n/ \log n) \) processors, as follows.

1. For every \( R_i \), do Test\((L(R_i), s)\) (by Lemma 5.4) for each segment \( s \) on \( ICP(C_j) \), where \( |ICP(C_j)| \) is not Case (a) (cf. Subsubsection 5.3.1.2). Let \( S_i = \{ s' | s' \) is a segment on one of these \( ICP(C_j) \)'s and \( s' \) is not above all the lines in \( L(R_i) \} \).

2. Let \( s^* = w_aw_b \) be the segment in \( S^* = S_1 \cup S_2 \cup \cdots \cup S_{n^{1/3}} \) such that the vertex indices (i.e., \( a \) and \( b \)) of the endpoints for \( s^* \) are no smaller than the vertex indices of the endpoints for any other \( s' \in S^*, \) where \( a < b \).

3. If \( s^* \) does not exist (each \( ICP(C_j) \) such that \( |ICP(C_j)| \) is not Case (a) is above all the lines in \( DR^+(L_P) \)), then stop. Otherwise, let \( s^* \) be on \( ICP(C_{j'}) \) for a unique \( j' \).

4. If \( s^* \) is below a line in \( L(DR^+(L_P)) \), then mark the bad interval \([1, b] \) (on \( bd_{1m}(M_P) \)), and stop. If a ray in \( DR^+(L_P) \) pseudo-hits \( s^* \), then mark the bad interval \([1, b - 1] \), check the weak visibility of \( P \) from \( e_{b-1}(M_P) \) (by the algorithm in Section 5.4), and stop.

5. (Now no ray in \( DR^+(L_P) \) pseudo-hits \( s^* \).) If \( |ICP(C_{j'})| \) is Case (d) and \( |C_{j'}| > n^{1/3} \), then mark the bad interval \([1, a - 1] \), check the weak visibility of \( P \) from \( e_{a-1}(M_P) \) (by the algorithm in Section 5.4), and recursively call the procedure on \( DR^+(L_P) \) and \( C_{j'} \); if \( |ICP(C_{j'})| \) is Case (d) but \( |C_{j'}| \leq n^{1/3} \), then exit the procedure. Otherwise, \( |ICP(C_{j'})| \) is either Case (b) or Case (c).
(6) If according to Case (b) or Case (c), \([a, b] \) is a bad interval, then mark the bad interval \([1, b] \) and stop; otherwise, there is exactly one edge \(e \) on \(bd_{ab}(M_P) \) that may not be bad (and all other edges on \(bd_{ab}(M_P) \) are definitely bad). Use the algorithm in Section 5.4 to check the weak visibility of \(P \) from \(e \), mark the bad interval according to the result of the check, and stop.

When the size of chain \(C \) in the above procedure has been reduced to \(\leq n^{1/3} \), we exit from the procedure and do the following: (a) for every edge \(e \) of \(C \), find whether \(e \) is below a ray in \(DR^+(L_P) \) or \(e \) intersects a ray in \(DR^+(L_P) \), (b) find the most counterclockwise edge \(e' \) on \(C \) that is intersected by a ray in \(DR^+(L_P) \), (c) find the most counterclockwise point \(p \) on \(e' \) intersected by a ray in \(DR^+(L_P) \) (let \(p \) be on \(e_b(M_P) - w_{b'} \)), and (d) mark the bad interval \([1, b'] \) and check the weak visibility of \(P \) from each edge of \(M_P \) that contains \(p \) (by using the algorithm in Section 5.4). We perform (a) for each \(e \) by using \(O(n^{2/3}/\log n) \) processors and in \(O(\log n) \) time (this is done by first applying Lemma 5.4 to \(e \) and the \(R_i \)'s, and then applying the brute force method to \(e \) and the unique remaining \(R_j \)). The most counterclockwise edge \(e' \) intersected by a ray in \(DR^+(L_P) \) can be easily obtained in (b) from the \(O(n^{1/3}) \) candidates, in \(O(\log n) \) time. The most counterclockwise point \(p \) in (c) is found by simply checking \(e' \) against each ray in \(DR^+(L_P) \).

The above computation for eliminating the bad edges in \(WVE(M_P) \) by using \(DR^+(L_P) \) clearly requires \(O(\log n) \) time and \(O(n/\log n) \) processors. We only need to prove that the algorithm indeed makes sure that all the edges on \(bd_{1c}(M_P) \) are bad, where \(p^* \) on \(e_c(M_P) - w_c \) is the most counterclockwise first-hit point by \(DR^+(L_P) \). Observe that if a ray in \(DR^+(L_P) \) intersects a point \(q \) on \(e_{a'}(M_P) - w_{a'} \), then by Lemma 5.18, \([1, a'] \) is a bad interval on \(bd_{1m}(M_P) \). Let \(s^* = \overline{w_a w_b} \) be the segment on \(ICP(C_{j'}) \) obtained in step (2) of the procedure, \(a < b \).

We use an induction argument. We would like to show that each time the recursive procedure is called (with the input chain \(C = bd_{st}(M_P), s < t \), we either have \(p^* \) on \(bd_{st}(M_P) \) and we know that interval \([1, s] \) is bad, or \(p^* \) is not on \(bd_{st}(M_P) \) and we have already made sure that intervals \([1, s] \) and \([t, c] \) are both bad. When \(C = bd_{1m}(M_P), \)
the induction basis is trivially true (because every ray of $DR^+(L_p)$ intersects $s^*_i$, $p^*$ is certainly on $bd_{1m}(M_p)$). Assume that what we would like to show is true for $C = bd_{st}(M_p) \subseteq bd_{1m}(M_p)$. When $C$ is processed by the procedure, there are two possible cases: Case (1) (where $p^*$ is on $C$) and Case (2) (where $p^*$ is not on $C$).

We first prove for Case (1). Let $p^*$ belong to subchain $C_k$ of $C$ for the largest index $k$. By Lemma 5.18, there is no $C_j$ such that $j > k$ and $DR^+(L_p)$ intersects $ICP(C_j)$. There are two subcases: (1.1) $ICP(C_k)$ is not Case (a), and (1.2) $ICP(C_k)$ is Case (a). In Subcase (1.1), step (1) of the procedure is applied to each segment of $ICP(C_k)$, and by Lemma 5.18 and by the definition of $s^*$ (i.e., the vertex indices of the endpoints for $s^*$ are the largest), we correctly restrict our search for $p^*$ to $bd_{ab}(M_p)$. In Subcase (1.2), step (1) of the procedure is not applied to the segments of $ICP(C_k)$. Hence the procedure fails to locate $p^*$. If $s^*$ exists (i.e., $S^*$ is nonempty), then $bd_{ab}(M_p)$ certainly does not contain $p^*$ (see Figure 5.13). But we claim that in this situation, no edge on the subchain $C'$ of $C$ from $w_b$ counterclockwise to $p^*$ is in $WVE(M_p)$.

Proof of the claim. Every edge on $C_k$ is clearly bad since $|ICP(C_k)|$ is Case (a). For a subchain $C_i \subseteq C'$, $i < k$, either $|ICP(C_i)|$ is Case (a) (hence the edges on $C_i$ are all bad) or $|ICP(C_i)|$ is not Case (a) and each segment $s$ on $ICP(C_i)$ is above all the lines in $L(DR^+(L_p))$. Let $r'$ be the ray of $DR^+(L_p)$ that first-hits
$p^*$ (see Figure 5.13). Since each segment $s = \overline{w_a w_b}$ on $ICP(C_i)$ is above all the lines in $L(DR^+(L_P))$, $s$ is above $r'$ (let $a' < b'$). Thus all the edges on $bd_{a'b'}(M_P)$ are above $r'$. Because $r'$ first-hits $e_c(M_P) - w_c$ on $C_k$, by an observation which is similar to Lemma 5.6, there exists a vertex $w_z$ on the subchain of $C$ from $w_{c+1}$ clockwise to $w_{b'}$ such that all the edges on $bd_{a'b'}(M_P)$ are bad with respect to $r_z^-$. □

If $s^*$ does not exist (i.e., $S^*$ is empty), then interval $[s, c]$ must be bad (the proof is similar to the one for the claim above, by letting $C'$ be the subchain of $C$ from $w_s$ counterclockwise to $p^*$). In the situation where $s^*$ does not exist, the algorithm stops; by the induction hypothesis (that $[1, s]$ is a bad interval), we are sure that $[1, c]$ is a bad interval. When $s^*$ exists, if the procedure recursively calls on subchain $C_{j'} = bd_{ab}(M_P)$, then it reports $[1, a]$ as a bad interval (because no ray in $DR^+(L_P)$ pseudo-hits $s^*$). In the recursive call on $C_{j'}$, the induction hypothesis is clearly maintained for the case where $p^*$ is on $C_{j'}$, and is also maintained for the case where $p^*$ is not on $C_{j'}$ (since both $[1, a]$ and $[b, c]$ are surely bad).

The proof for Case (2) is essentially the same as Case (1). The only difference in this case is that the ray $r'$ first-hits $p^*$ outside $C$. By using a similar argument as the one for the proof of the claim above, we can show that $[1, a]$ and $[b, c]$ are both bad if $s^*$ exists, and $[1, c]$ is bad if $s^*$ does not exist.

The algorithm stops at one of the following cases:

(a') $s^*$ does not exist. We have discussed this case.

(b') $s^*$ is below a ray in $DR^+(L_P)$. Then obviously $p^*$ is not on $bd_{ab}(M_P)$. Since $[b, c]$ is surely bad, the procedure reports that $[1, b]$ is bad.

(c') A ray in $DR^+(L_P)$ pseudo-hits $s^*$. Then if $p^*$ is on $bd_{ab}(M_P)$, it can only be at $w_b$. Hence the algorithm reports the bad interval $[1, b - 1]$ and checks the weak visibility of $P$ from $e_{b-1}(M_P)$. If $p^* \neq w_b$, then the check will return a “no” and hence $[1, b]$ is bad. If $p^* = w_b$, the check does not affect the final result.
(d') No ray in $DR^+(L_P)$ pseudo-hits $s^*$ and $|ICP(bd_{ab}(M_P))|$ is Case (b) or Case (c). Then by the discussion in Subsubsection 5.3.1.2, at most one edge on $bd_{ab}(M_P)$ is in $WVE(M_P)$. Hence we just check the weak visibility of $P$ from that edge and report the bad interval accordingly.

(e') No ray in $DR^+(L_P)$ pseudo-hits $s^*$, $|ICP(bd_{ab}(M_P))|$ is Case (d), and we exit from the recursive procedure. Then, we locate the most counterclockwise point $p$ on $bd_{ab}(M_P)$ intersected by a ray in $DR^+(L_P)$, report the bad interval $[1, b']$ (where $p$ is on $e_{b'}(M_P) - w_{b'}$), and check the weak visibility of $P$ from each of the (at most two) edges containing $p$. If $p \neq p^*$, then each check will certainly return a "no", and we are sure that the edges on the subchain of $bd_{ab}(M_P)$ from $p$ counterclockwise to $w_b$ are all bad (since they must be all above $DR^+(L_P)$). If $p = p^*$, the checks do not affect the final result.

In conclusion, the above algorithm correctly eliminates from $WVE(M_P)$ all the bad edges by using $DR^+(L_P)$.

5.6 Applications

Using the parallel algorithm for detecting the weak visibility of a simple polygon, several problems on simple polygons or weakly visible polygons can be solved optimally in parallel. In this section, we describe the solutions to these problems.

Given an $n$-vertex simple polygon $P$, we can now find whether $P$ is weakly visible, and if it is, report all the edges from each of which $P$ is weakly visible, within $O(\log n)$ time using $O(n/\log n)$ processors. In the rest of this section, we assume WLOG that it is already known (after applying our weak visibility algorithm) that $P$ is indeed a weakly visible polygon and edge $e = e_n$ is one of the weakly visible edges for $P$. Furthermore, we assume that we have partitioned $P$, with respect to edge $e$, into at most three subpolygons $P_1$, $P_2$, and $P_3$, as shown in Figure 5.14 (this partition scheme has been used in Subsection 5.4.1). This partition of $P$ has the properties
that $e$ is a convex weakly visible edge for $P_2$ and that $P_1$ (resp., $P_3$) is star-shaped with $v_1$ (resp., $v_n$) in its kernel.

5.6.1 Computing Shortest Paths in a Weakly Visible Polygon

The first problem we consider is that of computing shortest paths inside $P$. Given two points $p$ and $q$ inside $P$, this problem is to compute a path, called the shortest path, connecting $p$ and $q$ which does not intersect the exterior of $P$ and whose total Euclidean distance is minimized. Given an arbitrary simple polygon, the parallel solutions to this shortest paths problem usually include a preprocess of triangulating the polygon (see [57, 72]). The best parallel polygon triangulation algorithms run in $O(\log n)$ time using either $O(n)$ processors on the CREW PRAM [67, 135] or $O(n/\log n)$ processors on the (more powerful) CRCW PRAM [69]. Hence these CREW PRAM algorithms for computing shortest paths in a simple polygon are not yet optimal. Here we show that after the weak visibility algorithm indicates that $P$ is indeed weakly visible, the shortest paths problem on $P$ can be solved without triangulating $P$ first, in $O(\log n)$ time using $O(n/\log n)$ CREW PRAM processors, which is optimal.
Figure 5.15 Computing the shortest path $SP(p, q)$.

We denote the shortest path between $p$ and $q$ by $SP(p, q)$. If line segment $pq$ does not intersect the exterior of $P$ (whether this is the case or not can be easily identified in $O(\log n)$ time using $O(n/\log n)$ processors), then we are done, because $SP(p, q) = pq$. Thus, WLOG, we assume that segment $pq$ does intersect the exterior of $P$, and we need to compute $SP(p, q)$. The parallel algorithm for computing $SP(p, q)$ is as follows.

There are two cases: (i) both $p$ and $q$ belong to the same polygon $P_i$, $i \in \{1, 2, 3\}$, and (ii) $p$ and $q$ belong to two distinct $P_i$ and $P_j$, where $i$ and $j$ are both in $\{1, 2, 3\}$.

For case (i), we only consider the following subcase: both $p$ and $q$ are in $P_2$ (because the other subcases of case (i) are actually simpler). In this subcase, the shortest path $SP(p, q)$ obviously does not intersect the exterior of polygon $P_2$. Let $r_p(pq)$ (resp., $r_q(pq)$) be the ray starting at $p$ (resp., $q$) and containing $pq$. WLOG, suppose that $r_p(pq)$ (resp., $r_q(pq)$) first-hits $bd(P_2)$ at $p'$ (resp., $q'$), where $p' \in e_i$, $q' \in e_j$, and $i < j$ (see Figure 5.15). Note that $p'$ and $q'$ can be computed by using parallel prefix. Let $C_{p'q'}$ denote the chain on $bd(P_2)$ from $p'$ counterclockwise to $q'$. Then $SP(p, q)$ only
passes the vertices of $P_2$ that are on $C_{p'q'}$ and $SP(p,q)$ makes only right turns. To see this, observe that (1) $SP(p',q')$ makes only right turns (since $C_{p'q'}$ is weakly visible from $e$), and (2) there exists a point $p''$ (resp., $q''$) on $e$ such that $p$ (resp., $q$) is visible from $p''$ (resp., $q''$) and that $SP(p,q)$ does not intersect the exterior of the region $Q$ in $P_2$, where $Q$ is enclosed by $pp', C_{p'q'}, q'q, qq'', q''p''$, and $p''p$ (see Figure 5.15). Given $C_{p'q'}$, we first compute $SP(p',q')$ (by using the algorithm in Subsection 5.4.3), then compute the common tangent between $p$ and $SP(p',q')$ (resp., $q$ and $SP(p',q')$). Let the common tangent between $p$ and $SP(p',q')$ (resp., $q$ and $SP(p',q')$) touch $SP(p',q')$ at $v_k$ (resp., $v_l$). Then $SP(p,q) = SP(p,v_k) \cup \overline{v_kv_l} \cup SP(v_l,q)$.

We now consider case (ii) where $p$ and $q$ are in two distinct $P_i$ and $P_j$. We first consider the subcase where $p$ is in $P_2$ and $q$ in $P_3$ (note that the subcase when $p$ is in $P_2$ and $q$ in $P_1$ is symmetric). In this subcase, we first compute the shortest path $SP(p,v_b)$ (cf. Figure 5.14), the shortest path $SP(p,v_n)$, and the shortest path $SP(q,v_b)$, by using the algorithm for case (i) above. We then compute the common tangent between $SP(p,v_b)$ and $SP(q,v_b)$ (these two paths are separated by the ray starting at $v_b$ and containing $\overline{v_bv_n}$). Let this common tangent be $\overline{v_kv_l}$, where $v_k$ is on $SP(p,v_b)$ and $v_l$ on $SP(q,v_b)$. If this common tangent intersects $\overline{v_kv_l}$, then $SP(p,q) = SP(p,v_k) \cup \overline{v_kv_l} \cup SP(v_l,q)$ (in this situation, $SP(p,q)$ makes consistent right turns). Otherwise, $SP(p,q) = SP(p,v_n) \cup \overline{v_nv_q}$ (note that $q$ is visible from $v_n$), since in this situation, $SP(p,q)$ makes exactly one left turn at $v_n$. All the above computations can be done in $O(\log n)$ time using $O(n/\log n)$ processors. If $p$ is in $P_1$ and $q$ in $P_3$, then there are three possibilities. (1) If both $p$ and $q$ are below the line $l(e)$ that contains $e$, then $SP(p,q) = \overline{pv_1} \cup e \cup \overline{v_nq}$ (because $P_1$ is visible from $v_1$ and $P_3$ is visible from $v_n$). (2) If both $p$ and $q$ are above $l(e)$, then $SP(p,q)$ is computed as in case (i). (3) If exactly one of $p$ and $q$ is above $l(e)$, then $SP(p,q)$ is computed as in the subcase where $p$ is in $P_2$ and $q$ in $P_3$. Hence all these three situations can be resolved within the claimed complexity bounds.
5.6.2 Triangulating a Weakly Visible Polygon

Many geometric problems can be solved efficiently on triangulated polygons. A simple polygon can be triangulated in linear time sequentially [30]. For triangulating a weakly visible polygon, a simple sequential linear time algorithm is given in [126]. In general, triangulating a simple polygon in parallel requires $O(\log n)$ time using either $O(n)$ processors on the CREW PRAM [67, 135] or $O(n/\log n)$ processors on the (more powerful) CRCW PRAM [69]. Hence these CREW PRAM triangulation algorithms are suboptimal. We show that if $P$ is known to be weakly visible, then its triangulation can be done optimally in parallel, in $O(\log n)$ time using $O(n/\log n)$ CREW PRAM processors. Our approach is different from [126].

We triangulate $P$ by separately triangulating each of $P_1$, $P_2$, and $P_3$. Since both $P_1$ and $P_3$ are star-shaped, their triangulations can be done in the required complexity bounds by using the parallel algorithm for triangulating star-shaped polygons in [33]. So we only need to deal with $P_2$. Note that $P_2$ is weakly visible from convex edge $e$. The parallel algorithm for triangulating $P_2$ is very similar to the algorithms in [67, 33] for triangulating a monotone polygon. The differences between the algorithm here and those in [67, 33] are: (1) we compute the internal convex paths (but [67, 33] compute the upper (or lower) convex hulls) of the subchains on $bd(P_2)$, and (2) before we compute the common tangent between the internal convex paths for $C'$ and $C''$, where $C'$ and $C''$ are two disjoint subchains of $bd(P_2)$ (except possibly at one of their common endpoint), we must find a ray that separates $ICP(C')$ and $ICP(C'')$ (while in the case of a monotone polygon, such a ray is readily available – it is always vertical). Both (1) and (2) can be taken care of by using our algorithm in Subsection 5.4.3 (actually, Subsection 5.4.3 deals with a harder problem because there the polygon is not known to be weakly visible). Therefore, triangulating $P_2$ can be done in $O(\log n)$ time using $O(n/\log n)$ processors.
5.6.3 Solving the One-Cruising-Guard Problem

The one-cruising-guard problem is as follows: Given an edge $e$ (=$e_n$) from which polygon $P$ is weakly visible, compute the contiguous portion $g(e)$ on $e$ of minimum length such that $P$ is weakly visible from $g(e)$. Intuitively, $g(e)$ is the shortest segment on the specified "wall" $e$ by which a guard has to patrol back and forth in order to keep the polygonal "house" $P$ completely under surveillance. This problem was raised by Ching et al. [38], and they sketched a sequential linear time algorithm for it. Here we show that this problem can be solved optimally in $O(\log n)$ time using $O(n/\log n)$ processors.

For each $V_j$, let $s_j(1)$ (resp., $s_j(n)$) be the line segment on the shortest path inside $P$ connecting $v_j$ and $v_1$ (resp., $v_j$ and $v_n$) such that $s_j(1)$ (resp., $s_j(n)$) is adjacent to $v_j$. Let $r(s_j(1))$ (resp., $r(s_j(n))$) be the ray which starts at $v_j$ and contains $s_j(1)$ (resp., $s_j(n)$). Let ray $r(s_j(1))$ (resp., $r(s_j(n))$) intersect $e$ at point $r_{v_j}$ (resp., $l_{v_j}$). (It is possible that $r_{v_j} = l_{v_j}$.) Then $l_{v_j}r_{v_j} \subseteq e$ is the maximum portion on $e$ that is visible from $v_j$. Let interval $I_j = [l_{v_j}, r_{v_j}]$ (on $e$) represent $l_{v_j}r_{v_j}$. Note that for all $j = 2, 3, \ldots, n-1$, $s_j(1)$ and $s_j(n)$, and hence $r(s_j(1))$ and $r(s_j(n))$, can be computed by using the algorithm in Section 5.4, in $O(\log n)$ time using $(n/\log n)$ processors. Therefore, we assume that $I_j$ have already been computed for all $j = 2, 3, \ldots, n-1$. Also, we let both $I_1$ and $I_n$ be the complete interval for $e$.

To characterize the shortest contiguous portion $g(e)$ on $e$ from which $P$ is weakly visible, Ching et al. [38] defined a set of $O(n)$ intervals on $e$, where each interval is determined by the relative positions of $r_{v_j}$ and $l_{v_{j+1}}$ on $e$. The way we define the set of intervals on $e$ is different from [38]. In fact, our characterization of this problem implies a sequential linear time algorithm which is different from [38].

We define a set of intervals, called the characteristic intervals, as follows. For every edge $e_j$, $j = 1, 2, \ldots, n-1$, if $I_j \cap I_{j+1} \neq \emptyset$, then let $CI_j = I_j \cap I_{j+1}$, and call $CI_j$ a type-1 characteristic interval and $e_j$ a type-1 edge (see Figure 5.16 (a)); if $I_j \cap I_{j+1} = \emptyset$, then let $CI_j = [r_{v_j}, l_{v_{j+1}}]$, and call $CI_j$ a type-2 characteristic interval and
$e_j$ a type-2 edge (see Figure 5.16 (b)). The next lemma shows the relations between $g(e)$ and the two types of characteristic intervals.

Lemma 5.21 The shortest weakly visible portion $g(e)$ must contain at least one point on each type-1 interval, and must contain completely each type-2 interval.

Proof. Because $P$ is a simple polygon, the following can be easily seen to hold for each $e_j$: (i) If $CI_j$ is of type-1, then the whole edge $e_j$ is visible from any point on $CI_j$, and (ii) if $CI_j$ is of type-2, then $CI_j$ is the shortest contiguous portion on $e$ from which $e_j$ is weakly visible (see Figure 5.16). Hence the lemma follows.

Based on Lemma 5.21, $g(e)$ is computed as follows. We denote the left (resp., right) endpoint of an interval $I$ by $le(I)$ (resp., $re(I)$).

1. Compute $CI_i$ for every edge $e_i$, $i \neq n$.

2. Among all the type-1 intervals $CI_i$, find the rightmost $le(CI_i)$ and the leftmost $re(CI_i)$. If $le(CI_i)$ is on or to the left of $re(CI_i)$, then let an interval $TI_1 = [le(CI_i), re(CI_i)]$ (in this case, every type-1 interval contains $TI_1$, and thus every type-1 edge is completely visible from any point on $TI_1$ by Lemma 5.21); otherwise, let $TI_1 = [re(CI_i), le(CI_i)]$ (in this case, at least two type-1 intervals, i.e., $CI_i$ and $CI_{i'}$, do not contain $TI_1$, and hence the shortest contiguous portion on $e$ from which the union of all the type-1 edges is weakly visible is the whole interval $TI_1$).

3. Among all the type-2 intervals $CI_j$, find the rightmost $re(CI_j)$ and the leftmost $le(CI_j)$. Let an interval $TI_2 = [le(CI_j), re(CI_j)]$ (by Lemma 5.21, the shortest contiguous portion on $e$ from which the union of all the type-2 edges is weakly visible is the whole interval $TI_2$).

4. In the case when $TI_1$ is contained in every type-1 interval:

   (4.1) If there exists at least one type-2 interval, then

   (4.1.1) if $TI_1 \cap TI_2 \neq \emptyset$, then $g(e) = TI_2$;
Figure 5.16 Illustrating the two types of characteristic intervals.
(4.1.2) otherwise, \( g(e) = [\text{le}(TI_1), \text{le}(TI_2)] \) if \( TI_1 \) is to the right of \( TI_2 \), and
\[ g(e) = [\text{re}(TI_1), \text{re}(TI_2)] \] if \( TI_1 \) is to the left of \( TI_2 \).

(4.2) Otherwise, \( g(e) \) can be any point on \( TI_1 \).

(5) In the case when \( TI_1 \) is not contained in every type-1 interval:

(5.1) If there exists at least one type-2 interval, then
\[ g(e) = [\min\{\text{le}(TI_1), \text{le}(TI_2)\}, \max\{\text{re}(TI_1), \text{re}(TI_2)\}] \]

(5.2) Otherwise, \( g(e) = TI_1 \).

The correctness of the above algorithm follows from Lemma 5.21. The complexity bounds of the algorithm are very easy to see. Steps (2) and (3) require parallel prefix. The other steps of the algorithm can all be trivially done in \( O(\log n) \) time using \( O(n/\log n) \) processors. Hence the one-cruising-guard problem is solved in \( O(\log n) \) time using \( O(n/\log n) \) processors. The sequential and parallel algorithms for this problem are useful in solving the problem of computing the shortest weakly visible subedge for a simple polygon [35].

5.6.4 Other Applications

There are other applications for the weak visibility algorithm. For polygon \( P \) that is weakly visible from \( e_n \), we denote by \( w(v_j) \), for every vertex \( v_j \) on \( bd_{1,n} \), the wedge from ray \( r(s_j(1)) \) clockwise to ray \( r(s_j(n)) \), and call this wedge the maximum visible wedge of \( v_j \). Computing the maximum visible wedges for all the vertices on a weakly visible polygonal chain arises as a subproblem in some other geometric problems, for example, in finding the minimum visible distance between two nonintersecting simple polygons [8]. Our algorithm in Section 5.4 optimally computes the maximum visible wedges for all vertices in \( O(\log n) \) time using \( O(n/\log n) \) processors (in [8], this computation was done in super-linear total work).

Another application is to check whether the boundary of a simple polygon \( P \) is weakly visible from a circle (or a convex polygon) that encloses \( P \) (such a polygon
is called a \textit{weakly externally visible polygon} \cite{126, 24, 25}). A sequential linear time algorithm for detecting whether a polygon is weakly externally visible is given in \cite{126}. We provide an optimal parallel algorithm for this problem, as follows. (1) Compute the convex hull $CH(P)$ of $P$; (2) for every edge $\overline{v_i v_j}$ of $CH(P)$ that is not an edge of $P$, check whether the subchain $bd_{ij}$ of $bd(P)$ is weakly visible from $\overline{v_i v_j}$, where $bd_{ij}$ and $\overline{v_i v_j}$ form a simple polygon that does not intersect the interior of $P$; (3) if every check performed in (2) reports “yes”, then $bd(P)$ is weakly visible from every enclosing circle, and otherwise, it is not weakly visible from any enclosing circle. This parallel algorithm runs in $O(\log n)$ time using $O(n/\log n)$ processors because we can perform (1) by using the algorithm for computing the convex hull of a simple polygon \cite{33}, (2) by using the algorithm in Section 5.4, and (3) by using parallel prefix.
6. CONCLUSION

In this thesis, we have presented several new parallel techniques for solving shortest paths, visibility, and other related geometric problems. These techniques, in conjunction with the new geometric insights that we have given, form new parallel paradigms in computational geometry which are very different from those used for sequential algorithm design. Using these techniques, we have obtained efficient CREW PRAM or EREW PRAM algorithms for a number of important geometric problems; many our algorithms are even optimal.

We have designed efficient CREW PRAM algorithms for building a data structure that supports fast processing of queries about the lengths of the rectilinear shortest paths between arbitrary points and about the actual paths, in the presence of a set of disjoint rectangular obstacles. The techniques involved in our algorithms include: (i) efficiently finding a “staircase separator” for partitioning the obstacles and using it to guide the recursion, (ii) reducing the transitive closure computation in the “conquer” stage to a constant number of (min, +) matrix multiplications (instead of the usual logarithmic number of matrix multiplications), and (iii) showing that the matrices being multiplied in the “conquer” stage have a special structure that enables us to avoid the super-quadratic work bottleneck that is usually the price paid for doing parallel matrix multiplication. In addition to these techniques (which are likely to be useful in other contexts), we have used a number of observations that are specific to this particular kind of path problems. We achieved (ii) and (iii) by partitioning the boundaries of the obstacles in a way which ensures that the resulting path length matrices we used have a monotonicity property (such a property is apparently absent before applying our partitioning scheme). The most general version of our algorithms requires a novel pipelining of the computation up and down the recursion tree of the
algorithms, with \( O(n) \) computational "flows" that originate from all the nodes of the recursion tree and proceed only to the nodes whose associated problem sizes are larger than that of the flow's origin.

We have also presented a parallel algorithm for computing the visible portions of an \( n \)-vertex simple polygonal chain \( P \) from a point \( q \) in the plane. This algorithm works for any polygonal chain that does not self-intersect. The algorithm runs in \( O(\log n) \) time using \( O(n/\log n) \) processors on the EREW PRAM, and is optimal. The techniques used in the algorithm are a combination of the quarter-root divide-and-conquer and two-way divide-and-conquer strategies, and include a method for a logarithmic time computation of the intersections between special polygonal chains that intersect each other twice. We have applied this algorithm to solve optimally several geometric problems in parallel. This algorithm is likely to find applications in solving other geometric problems involving polygonal chains.

We have further elaborated the parallel technique used for the visibility algorithm into a more general form. The generalized technique consists of the rank tree data structure, several kinds of parallel operations on rank trees, and a combination of the two-way divide-and-conquer and many-way divide-and-conquer strategies which are used in conjunction with the parallel tree operations. The parallel tree operations include parallel searching, parallel concatenation, and parallel split. These parallel tree operations can be performed without having read conflicts. We have showed that this technique is useful in solving many geometric problems optimally on the EREW PRAM.

We have shown how to solve optimally in parallel the problem of detecting the weak visibility of a simple polygon by using several new techniques and geometric observations. We built a data structure for processing in parallel the queries that check the position of a line segment with respect to a set of lines in the plane. This data structure is useful for identifying the non-weakly visible edges of a simple polygon. We have also provided a parallel scheme for checking the weak visibility of a simple polygon from an edge. This scheme is based on a new idea for computing in
parallel the internal convex paths in a polygon and in the same time testing the weak visibility of that polygon from the specified edge. These techniques and geometric observations have been applied to solve optimally a number of problems on weakly visible polygons in parallel.

There are still many important problems that remain open in parallel computational geometry. This research can possibly be extended in the following directions.

One direction is to investigate parallel solutions to other related fundamental geometric problems in the 2-dimensional space. Examples of such problems include: computing Voronoi diagrams [2, 39, 45, 71], triangulating simple polygons [67, 69, 135], computing shortest paths with arbitrary obstacles and in various metrics [14, 29, 57, 72], computing visibility with respect to various geometric objects [15, 16, 23, 29, 34, 44, 72], solving the linear programming problem [52], etc. Optimal or efficient parallel algorithms for many of these problems are not known even on the CREW PRAM.

Another direction is to study parallel algorithms for geometric problems in higher dimensions. For example, the convex hull problem in the 3-dimensional space is still not solved optimally on the CREW PRAM [2, 9, 39, 49]. For many problems that are important in applications, such as shortest paths and visibility problems in the 3-dimensional space, very few efficient parallel algorithms are known. For instance, [70, 116] are among the few articles which deal with PRAM algorithms for 3-dimensional visibility problems. Parallel algorithms for shortest paths problems in higher dimensions are even more scarce. Solving in parallel various intersection problems in higher dimensions also requires much more research effort [2, 10, 16, 39, 66, 68, 70, 78].

The third direction is to study other kinds of parallel algorithms for solving geometric problems. One such area is studying parallel randomized algorithms for solving geometric problems. Usually, parallel deterministic algorithms are quite involved and are not easy to implement. Also, the constants hidden behind the "big oh" of the time
complexities for parallel deterministic algorithms are often quite large. In comparison, parallel randomized algorithms are reasonably simple to implement, have small constants, and succeed with high probability. Hence, parallel randomized algorithms are often preferred in practical applications. Reif and Sen have designed parallel randomized techniques and used them to solve efficiently a number of fundamental geometric problems on the PRAM (see [115, 117, 118]). There are other parallel randomized algorithms for solving important geometric problems on the PRAM (for example, see [40, 62, 78, 123]). Proving P-completeness for geometric problems is also very interesting [13].
BIBLIOGRAPHY
BIBLIOGRAPHY


VITA
VITA

Mr. Ziyi Danny Chen was born in Taishan, Guangdong, the People's Republic of China, on October 15, 1960. In 1977, he graduated from Fucheng High School, in Taishan, Guangdong. He then studied in the Philosophy Department, Wuhan University in Wuhan, Hubei, P.R.C., from 1978 to 1981. On September 6, 1981, Mr. Chen came to the United States and entered the University of San Francisco, where he received the B.S. degree in Computer Science and Mathematics (double major) in 1985. Mr. Chen continued his education at Purdue University in West Lafayette, Indiana, and received the M.S. and Ph.D. degrees in Computer Science in 1988 and 1992, respectively. While at Purdue, Mr. Chen was supported as research and teaching assistants.

Also while at Purdue, Mr. Chen met Xiaobo Sharon Hu in a CS514 class, during the spring of 1986, and married her in Lafayette, Indiana, on May 27, 1990.