Geometric Hierarchies and Parallel Subdivision Search

by

Nounou Norman Dadoun

B.Sc. (Honours), The University of Western Ontario, 1977
M.Sc., The University of British Columbia, 1983

A Thesis Submitted in Partial Fulfillment of
The Requirements For the Degree of
Doctor of Philosophy

in
The Faculty of Graduate Studies
(Computer Science)

We accept this thesis as conforming
to the required standard

The University of British Columbia
August 1990
© N. Dadoun, 1990
In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science
The University of British Columbia
Vancouver, Canada

Date Oct 2/90
Abstract

Geometric hierarchies have proven useful for the problems of point location in planar subdivisions and 2- and 3-dimensional convex polytope separation on a sequential model of computation. In this thesis, we formulate a geometric hierarchy paradigm (following the work of Dobkin and Kirkpatrick) and apply this paradigm to solve a number of computational geometry problems on a shared memory (PRAM) parallel model of computation. For certain problems, we describe what we call cooperative algorithms, algorithms which exploit parallelism in searching geometric hierarchies to solve their respective problems.

For convex polygons, the geometric hierarchies are implicit and can be exploited in cooperative algorithms to compute convex polygon separation and to construct convex polygon separating/common tangents. The paradigm is also applied to the problem of tree contraction which is, in turn, applied to a number of specialized point location applications including the parallel construction of 2-dimensional Voronoi Diagrams.

For point location in planar subdivisions, we present parallel algorithms to construct a subdivision hierarchy representation. A related convex polyhedra hierarchy is constructed similarly and applied to the parallel construction of 3-dimensional convex hulls. The geometric hierarchy paradigm is applied further to the design of a data structure which supports cooperative point location in general planar subdivisions. Again, a related polyhedral hierarchy can be used to exploit parallelism for a cooperative separation algorithm for convex polyhedra.
# Table of Contents

Abstract ii

Table of Contents iii

List of Figures v

Acknowledgments vi

Chapter 1: Introduction 1

1.0 Preliminaries 1

1.1 Computational Geometry 3

1.2 Model of Parallel Computation 4

1.3 Basic Tools for Parallel Computation 10

1.4 Definitions & Representations 18

1.5 The Geometric Hierarchy Paradigm 20

1.6 Previous Related Work 23

1.7 Thesis Outline 28

Chapter 2: Linear Subdivision Search and Convex Polygon Hierarchies 32

2.1 Introduction and Related Work 32

2.2 Polygon Hierarchies 36

2.3 Polygon Separation 45

2.4 Polygon Separating and Common Tangents 60

2.5 A Sorted Point Set Convex Hull Algorithm 69
Chapter 3: Parallel Search in Restricted Subdivisions

3.1 Introduction and Related Work 76
3.2 Tree Contraction 79
3.3 Reduction to list ranking 82
3.4 Geometric Applications of Tree Contraction: Region Trees 88

Chapter 4: Parallel Processing for Search in Planar Subdivisions

4.1 Introduction and Related Work 96
4.2 Subdivision Hierarchies and Fractional Independent Sets 101
4.3 Polyhedral Applications 114

Chapter 5: Cooperative Search in Planar Subdivisions

5.1 Introduction and Related Work 118
5.2 A Static Subdivision Hierarchy 119
5.3 A Dynamic Subdivision Hierarchy 126
5.4 Polyhedral Separation Applications 130

Chapter 6: Conclusion

6.1 Summary 133
6.2 Directions for further work 134

References 136
List of Figures

Figure 2.1: The envelope between $\tilde{P}$ and $\hat{P}$ containing the boundary of $P$. 41

Figure 2.2: Monotone Polygonal Sectors. 44

Figure 2.3: Kitty Corner/Same Side Lemma Cases (a) and (b). 50

Figure 2.4: Kitty Corner Lemma Case (c). 52

Figure 2.5: Same Side Lemma Case (c). 57

Figure 2.6: Separating tangents. 64

Figure 2.7: Common tangents. 67

Figure 3.1: Prune and Bypass operations 83

Figure 3.2 Slicing off a Voronoi point 92

Figure 5.1 Two levels of a static hierarchy 125

Figure 5.2 A dynamic hierarchy 128
Acknowledgments:

In any undertaking of this magnitude (and duration), a space this short is certainly inadequate to thank by name all those who are so deserving. I have been fortunate in having many friends (far and near) who have contributed positively but intangibly to the successful completion of this thesis. First of all, the multitudinous masses with whom I have shared accommodation at '2862' for periodically worrying about my long hours and forgiving my occasional lapses in the cooking department. To the custodians of the Coastal Jazz and Blues Society, Bob, Ken and John (and my other musical buddies, Don, Carl-Babe etc.), without whom this thesis would have been completed much more quickly (but not nearly as joyfully).

The graduate student community (both past and present) in the computer science department for their technical and spiritual support, Barry Brachman for suggesting that Macs lack slack while not convincing me that SUNs are part of the subgenius, Teresa Przytycka for many stimulating and insightful discussions on the nature of parallel computation and innovative grammatical construction. Rick Morrison, George Tsiknis, Alex Kean, Steve Wismath, Lisa Higham, (and the list goes on) for their comraderie and academic spirit. The computer science faculty and staff have been, in general, as friendly and supportful as the grads. Carol Whitehead deserves special mention for kindly allowing this thesis to reside (quietly) in her Mac. Karl Abrahamson, Alan Wagner, Feng Gao, Richard Rosenberg, and many others deserve thanks for their stimulation and encouragement. Finally, my research supervisor David Kirkpatrick who has always managed to keep track of all of the levels of the hierarchy, from the elegant solutions at the top to the requisite attention to detail at the bottom, for advice, friendship and support above and beyond the call of duty.
Chapter 1: Introduction

1.0 Preliminaries

This thesis describes research in the area of computational geometry. As research in computational geometry progresses and problems within the area become better understood, it is natural to ask what problem solutions and problem-solving techniques can be translated to a parallel processing model of computation. Besides extending sequential results to exploit parallelism, in some settings it may be possible to lift parallel results from one dimension to higher dimensions. Many computational geometry results, including some which have been well-studied for a sequential model of computation, may benefit from new abstractions and implementation techniques when re-examined in the context of parallelism.

The recurring themes explored within the research described in this thesis are the design and use of hierarchical data structures in solving geometric problems on a parallel model of computation. A related consideration is the identification of data independence in geometric settings to allow the construction and search of hierarchical data structures in parallel.

The present research has focused upon the general problem area of subdivision search. Besides being a recurring problem in computational geometry, both in general and restricted planar subdivisions, there are a number of polygonal and polyhedral separation problems which can be formulated as instances of a subdivision search problem.

Much of the work presented here examines the application of the geometric hierarchy paradigm of Kirkpatrick [K83] (and subsequently, Dobkin and Kirkpatrick [DK82] [DK83] [DK90]) in a parallel setting. In the sequential setting, this paradigm has been useful for the efficient solution of many problems in computational geometry including convex polytope separation [DK82] [DK83] [DK85], planar point location [K83], the determination of other
properties of convex polytopes [DK90], the efficient construction of convex polyhedron intersection [C89], and others.

The next six sections (1.1 through 1.6) of this introductory chapter set the stage for the presentation of existing results and discussion of further work. Respectively, these sections provide brief descriptions of (1.1) the field of computational geometry, (1.2) properties of commonly used parallel models of computation, (1.3) basic tools for parallel algorithm design, (1.4) relevant definitions and representations, (1.5) the geometric hierarchy paradigm, and (1.6) a review of general hierarchical techniques along with a chronological development of results in the field which places this thesis work in context.

The final section (1.7) summarizes briefly the results which appear in the four chapters (2 through 5) which form the body of the thesis. The final chapter of the thesis contains a high level discussion of the results and discusses general directions for further work.

Portions of the thesis chapters have appeared in various published forms: an abridged version of Chapter 2 was presented at Allerton [DaK89c] and a full version has been submitted to Algorithmica for publication [DaK89b]. The tree contraction portion of Chapter 3 was presented at Allerton [ADKP87b] and has appeared as part of an article in the Journal of Algorithms [ADKP89]. The region tree portion of Chapter 3 was presented at the Computational Geometry Symposium as part of [DaK87a]. The Subdivision hierarchy construction of Chapter 4 was also part of [DaK87a] and a full version appeared in the Journal of Computer and Systems Science [DaK89a]. Techniques used in the subdivision hierarchy construction algorithm were applied to the problems of identifying fractional and maximal independent sets in planar graphs in [DaK87a] and a full version of these results appeared in Discrete Applied Mathematics [DaK87b]. The augmented hierarchy construction of Chapter 5 was part of [DaK89c] and is being prepared for journal submission [DaK90].
1.1 Computational Geometry

Computational Geometry is a field of research within Computer Science which takes an algorithmic approach to classical and modern geometric problems. According to Shamos and Hoey [SH76], two of the early researchers in the field, "the task of computational geometry is to relate the geometric properties of figures to the complexity of algorithms that manipulate them". As such, the bulk of research in computational geometry is concerned with the formulation of efficient algorithms which can be used to answer geometric queries (e.g. intersection, proximity, containment) for geometric objects (e.g. points, lines, polygons, polyhedra). These can sometimes be viewed as higher dimensional variants of one-dimensional problems such as sorting and searching.

It is sometimes useful to regard some of the problems addressed by researchers in computational geometry as idealizations of search and maintenance problems which occur in computer graphics, geographic databases, computational vision, robotics, VLSI design, and other fields of study. For example, the problem of identifying the facet of a planar subdivision which contains a given query point can be viewed as a idealization of the problem of determining which menu item has been selected on an interactive graphics display. These applied disciplines can provide a source of interesting problems and, in turn, can benefit from efficient algorithms for solving these problems.

In his thesis [S78], Shamos associated the term 'Computational Geometry' with a variety of geometric problems and related problem solving techniques. Since this nominal beginning, the area has matured with a large body of computational geometry research. Much of this research has been surveyed by Preparata and Shamos [PS85], Lee and Preparata [LP84], Mehlhorn [M84], and Edelsbrunner [E87]. With the proliferation of results in various aspects of computational geometry (as published in the proceedings of the ACM Symposium on Theory of Computing, the IEEE Symposium on Foundations of Computer Science and other
academic conferences), several forums dedicated to these results have been organized including the annual ACM Symposium on Computational Geometry, the annual Canadian Symposium on Computational Geometry and others.

The bulk of this thesis concentrates on one of the central problem areas in computational geometry: geometric searching. This involves searching some collection of geometric data for an answer to some geometric query. In general, we consider the repeated query formulation of geometric search in which we organize the data into some kind of data structure to facilitate searching. As discussed later, in the design of a data structure which facilitates certain geometric queries, we consider the preprocessing time required to build the structure, the storage space required to store it, and the resulting query time which it supports. Our focus is the application of parallelism to the construction and interrogation of hierarchical geometric search data structures.

1.2 Model of Parallel Computation

In general, we adopt the conventions and notations commonly used in the study of sequential algorithm design including the "big Oh" order notation used to describe the asymptotic complexity of algorithms [AHU74] [K76].

We use the term parallel computation to refer to synchronous multi-processing. This is distinguished from distributed computation which deals with issues of communication and asynchrony. There is no single generally accepted model of parallel computation. In the following, we discuss a number of different models along with their associated level of abstraction. A number of these models are described more extensively in various surveys of parallel computing [HLSM82] [KR88] [AlG89].
At the lowest level is the *circuit model* in which the processing elements are gates connected by wires and the multiprocessing is achieved by (synchronous) parallel flows through the circuit. At a higher level of abstraction is the *fixed interconnection network model*, which consists of a number of processors connected in a fixed topology. There are a variety of different interconnection schemes such as hypercubes, meshes, or butterfly/cube-connected cycles. These schemes differ in various ways, their extendability, maximum degree of any given processor node, and bisection-width. In general, the circuit and fixed interconnection models are closely associated with 'real-world' considerations of parallel computation. For example, although a two-dimensional mesh of $n$ processors may require $\Omega(\sqrt{n})$ steps for communication between two processors, many image processing applications map very naturally onto that architecture.

In the study of parallelism and its application to problem solving and algorithm design, it has been convenient to define a higher level abstract machine for which algorithms can be designed. Such an abstract machine should allow the examination of the benefits of parallelism without regard for the underlying interconnection architecture. A common abstraction used is the Parallel Random Access Machine (PRAM) in which a collection of RAM processors have access to a global shared memory which is used for all interprocess communication [FW78]. This is a natural extension of the RAM model which is common in the study of sequential problem solving and algorithm design [AHU74].

Within the definition of a PRAM, there are rules which govern how the common memory is to be accessed by different processors. We obtain successively stronger models by varying the rules governing the simultaneous access to shared memory: An EREW PRAM does not allow any concurrent read or write access to any memory cell, a CREW PRAM allows concurrent read but not write access to any memory cell, and a CRCW PRAM allows concurrent read or write access to any memory cell. When concurrent write is available, additional rules are required to resolve write conflicts. A CRCW COMMON PRAM allows
concurrent write only if all values simultaneously written to a given cell are the same. In a CRCW ARBITRARY PRAM, any processor attempting a simultaneous write to a given memory cell may succeed and the success of the computation should not depend on which processor actually succeeds. In a CRCW PRIORITY PRAM, each processor has an associated identifier and in any simultaneous write attempt to a given memory location, the processor with the highest identifier will succeed.

It is important to note that the PRAM is a highly idealized model of parallel computation and is not likely to be implemented directly due to the potentially high-bandwidth communication requirement associated with its memory accesses. One PRAM assumption of concern is that memory access time remains constant, independent of the number of processors. In a shared memory interconnection scheme, this is clearly unrealistic due to bounded fan-out/fan-in interconnection restrictions. In a shared memory bus architecture, the bandwidth of the bus itself is a similar limiting restriction. In view of this drawback, it is necessary to provide theoretical and technological justifications as to the validity of its choice as a model of computation for parallel algorithm design.

The various models of computation which are used for the study of parallel algorithms are very strongly related in the following theoretical sense: A problem is said to be in the complexity class NC if there exists an efficient algorithm for its solution, that is one that can be implemented in polylogarithmic time using a polynomial number of processors [P79]. It has been shown that the class NC is robust in the sense that there exist polynomial-hardware/polylogarithmic-time simulations between the commonly studied theoretical models of parallel computation such as circuits, PRAMs, Alternating Turing Machines (ATMs) and vector machines [KR88]. Therefore, an efficient algorithm (in the sense defined above) for one model can be translated into an efficient algorithm for any of the other models.
The PRAM is a natural extension of a Random Access Machine model which essentially ignores questions of processor interconnection. This allows the algorithm designer to examine the possibility of parallel implementation of RAM algorithms and to concentrate on the abstract opportunities for applying parallelism without concern for particular interconnection schemes (such as the hypercube, mesh or butterfly architectures). This enables us to explore the maximum benefit which can be obtained through the application of parallelism thus providing a theoretical benchmark for the comparison of different algorithms and approaches. As well, in recent years, a number of researchers have explored techniques for simulating a PRAM on various interconnection architectures, including techniques for emulation of shared memory on a mesh of processors [MS90] and on a butterfly architecture [R87].

Therefore, we assume a synchronous Single Instruction/Multiple Data (SIMD) shared memory PRAM for our algorithms. Many of the algorithms presented here solve their geometric problems by constructing a data structure in parallel which is used subsequently for concurrent access or they require synchronization in a cooperative setting. Although in some cases, an exclusive read/exclusive write (EREW) memory model suffices to implement our algorithms, we typically assume the availability of concurrent read. Where appropriate we elaborate on questions of memory access.

As is typical for sequential computational geometry [PS85], we further idealize the computing model with some simplifying assumptions. We assume that any single processor can perform standard arithmetic and comparison operations in constant time with no round-off error. To employ Cole and Vishkin's deterministic coin tossing technique (cf. [CV86a], [AM88] & Chapter 4), it is necessary to incorporate other assumptions about processor power.
the ability to perform bit operations on log \( k \)-bit numbers in constant time and non-uniformity (allowing the value of log*\( n \) to be built into an algorithm)\(^1\).

In the design of algorithms, the efficiency of an algorithm is typically described by its asymptotic worst-case requirement of the resources of time and space relative to \( n \), the size of the input [AHU74]. Sometimes trade-offs are possible, exchanging a decrease in the consumption of one resource for an increase in the other. In the specification of a geometric data structure which is to be used to answer particular queries, three resource measures are of interest: \( S(n) \), the space required for storing the data structure; \( T_{\text{PREP}}(n) \), the preprocessing time required to build the data structure; and \( T_{\text{QUER}}(n) \), the search time required to answer a query once the structure is built.

In the specification of parallel algorithms, another complexity measure must be included: \( P(n) \), the number of processors which are used by the algorithm. In the design of parallel algorithms, a major goal is the minimization of the work \( W(n) \) performed by the algorithm as given by the product \( W(n) = T(n) \times P(n) \), where \( T(n) \) is the time bound of the algorithm and \( P(n) \) is the number of processors required by the algorithm for an input of size \( n \).

It seems natural to compare the work performed by a parallel algorithm which solves a given problem to the work performed by a 'good' sequential algorithm which solves the same problem. This is natural in the sense that a 'good' sequential algorithm can provide a benchmark for how well a particular parallel algorithm performs. An efficient parallel algorithm is said to exhibit optimal speedup with respect to a given sequential algorithm for the same problem if, up to constant factors, it performs the same amount of work as the sequential algorithm. Of course, a parallel algorithm provides a corresponding sequential algorithm with

\(^1\) Unless otherwise specified, all logarithms are base 2. We also define log*\( n \) to be the number of applications of the log function required to reduce \( n \) to a constant value, say 4.
an $O(T(n) * P(n))$ time bound by employing a single processor to simulate \( P(n) \) parallel processors.

There are several common interpretations of the term optimal as used in reference to parallel algorithms. One is that a parallel algorithm is optimal if the work it performs matches the fastest known sequential algorithm for a given problem [CV86b]. Another is that a parallel algorithm is optimal if the work it performs matches the sequential lower bound for the problem it solves (although its running time may not be minimized) [G87a]. We use the term optimal in a more classical sense to refer to algorithms whose upper bounds match their respective problem's lower bounds for the same model of computation. Our usage attempts to reserve the term optimal for the notion of 'the best one can do'.

We are interested primarily in the design of efficient algorithms (in the sense defined above). Therefore, we examine the benefits derivable from the application of massive parallelism in which the number of processors is polynomially related to the size of the input. For the algorithms presented here it is sufficient to assume the availability of a number of processors linear in the size of the input. In some instances, it is possible to further reduce the processor requirements of a given algorithm using Brent's scheduling principle [B74] (cf. section 1.2) without increasing the running time, thus effecting a reduction in the work performed by that algorithm.

Since we deal exclusively with graphs (including planar graphs) whose edge set is linear in the size of the vertex set, we assume without loss of generality that each vertex and edge has associated with it a dedicated processor. Each processor has a distinct log \( n \)-bit identifier which can be used to make local decisions. The processor identifier of a processor assigned to a vertex (respectively edge) may also be referred to as the vertex (respectively edge) number.

The most common application of parallelism for subdivision search problems has been in parallel preprocessing of a data structure for sequential processing of queries [C80] [AG86b].
However, in the design and use of a data structure for subdivision search, we can examine both the number of processors used in constructing a data structure and the number of processors used in exploiting the structure to answer a single query. We use the term cooperative algorithm to refer to a search algorithm in which a number of processors cooperate to answer a single query. Hence in our specification of cooperative algorithms, the number of processors available, \( k \) for \( 1 \leq k \leq n \), is an input parameter. Ideally, as the number of processors available decreases and approaches one, the time bound of the cooperative algorithm should increase smoothly to match the best known sequential algorithm.

In a sense, the cooperative algorithms described here have a 'decreasing return' in their application of parallelism; the speedup may only be polylogarithmic in the number of processors. One might ask why issues of massive parallelism should be explored for problems which already admit a sublinear sequential algorithm. In addition to their purely theoretical interest, such problems may appear as recurring subproblems in a setting for which there are a varying number of processors available (cf. section 2.5).

1.3 Basic Tools for Parallel Computation

There are certain recurring basic problems whose solutions have become part of the folklore of sequential computation but whose solutions in the parallel setting are part of recent and ongoing research. A large body of this research has been surveyed recently by Karp and Ramachandran [KR88]. The tools and techniques which are used currently as the building blocks of parallel algorithms may not only differ from those used for sequential algorithms but may change as parallel algorithm design becomes better understood. Accordingly, we describe
some of the basic tools which are employed in this thesis and which are currently part of the standard repertoire of PRAM algorithm design.

The dynamic expression evaluation problem is the problem of evaluating an arithmetic expression with no free preprocessing [MR85]. A number of parallel techniques which have widespread application to graph theoretic and geometric problems can be related by regarding them as solutions to the dynamic expression evaluation problem for expressions of different forms. Sequentially, this problem (and the instantiations described below) can be solved with a constant number of passes over a linked list representation of the expression, thus requiring linear time.

Given an associative binary operation and an array of values to be combined using this operation, one may also desire an array of partial results, a list of results for all the prefix arrays derived from the input array. Typically, this is described using addition and is referred to as parallel prefix sums. More formally, given an array of values $x_1, x_2, ..., x_n$ and an associative binary operation $+$, compute all the prefix sums $P_j$ for $1 \leq j \leq n$ where $P_j = \sum_{i=1}^{j} x_i$.

Exploiting the associativity of the given operation, we can schedule the evaluation of the expression implied by the above definition however we wish. We can bracket the expression to form a completely balanced parenthesized expression corresponding to a balanced binary tree in which the leaves are operands and the internal nodes are operators combining subtrees (subexpressions) of almost equal size (differing by at most one leaf/operand). In this way, this evaluation problem admits to a simple algorithm [LF80] [KR88] by superimposing a balanced binary tree on the array and using that to schedule the processors to combine the elements independently by "sweeping" up the levels of the tree. This same 'binary tree scheduling' is used in a back-sweep phase to fill in the remaining prefix sums.
This parallel prefix sums algorithm runs in $O(\log n)$ parallel time using $n/\log n$ EREW processors and therefore demonstrates optimal speedup with respect to the linear time sequential scan algorithm. This simple algorithm has a surprising number of applications, from determining the occurrence of critical values in an array to purging an array of unwanted elements. For example, given an array with a number of live elements to be retained and a number of dead elements to be removed, a 0 is assigned to all dead entries and a 1 is assigned to all live entries. An application of the parallel prefix sums algorithm outlined above provides each live element with its index in the desired compacted array.

The tree contraction problem can be viewed as an instantiation of expression evaluation for an arbitrary parenthesized expression. In the sense that parallel prefix is an instantiation of expression evaluation for an (unbracketed) expression with a single associative operation, the parallel prefix problem can be regarded as a special case of the tree contraction problem. As defined in chapter 3, the tree contraction problem is to reduce a rooted ordered tree to its root by a sequence of independent vertex removals. This is an abstraction which divorces the tree contraction problem from its most familiar application, expression evaluation. Chapter 3 contains a more extensive discussion of tree contraction (and other applications) and presents a tree contraction algorithm which is a natural generalization of the standard parallel prefix algorithm outlined above.

Within a linked list, the rank of a given element is defined as the number of elements following that element. Given a set of items in a list structure with a next pointer associated with each item, the list ranking problem is to determine the rank of each element in the list. Once each element has its rank, an ordered array of elements can be constructed simply by using the rank as an array index. The list ranking problem generalizes easily to the weighted list ranking problem, where each element has an associated weight and the goal is to determine the weighted rank — the sum of the weights of all subsequent elements — for each list.
element. Again, this can be regarded as an instantiation of the expression evaluation problem, this time for a completely unbalanced expression.

In the parallel setting, the list ranking problem has received much attention in recent research (cf. [ADKP87a], [CV86a], [CV86b], [AM88]) because of its fundamental role in many parallel algorithms. Of particular interest is the role of list-ranking in the so-called *Euler Tour Technique* [TV84] which can be used to compute various functions on trees including preorder number, postorder number, descendant numbering and others. This is used in chapter 3 to provide a reduction of tree-contraction to list-ranking.

The problem of parallel list-ranking was addressed by Wylie [W79] who proposed the familiar and simple pointer-jumping algorithm: each element is assigned a processor and in each step "jumps over" its next element by composing next pointers. The steps continue until each element reaches the end of the list. The sizes of the jumps are increasing powers of 2 and therefore in a list of $n$ elements, all elements can determine their distance from the last element (their rank) within $O(\log n)$ time using $n$ processors. Although this algorithm has the benefit of simplicity, its time-processor product is $O(n \log n)$ and is thus an $O(\log n)$ factor from realizing optimal speedup with respect to the linear time sequential scan algorithm.

As noted by Vishkin [V84], the list ranking problem can be reduced to the parallel prefix problem by assigning each element in the list a weight of 1 and reversing the list. Unfortunately, the standard parallel prefix algorithm requires its elements to be presented in consecutive array locations and therefore that algorithm cannot be applied directly to the list-ranking problem. However, this view has been the basis for the (independent) pair-wise element combination techniques which underlies many of the common list-ranking algorithms.

Much effort has gone into producing a list-ranking algorithm which runs in $O(\log n)$ time and utilizes $n/\log n$ processors. At the heart of this problem is a fundamental question of efficient parallel computation: how to break symmetry to determine how a computation is to
proceed without duplicating effort. For example, this has been one of the main difficulties in
developing a parallel algorithm for the maximal independent set problem. Wylie's algorithm
for list-ranking makes no attempt to break symmetry, each element/processor participates in the
algorithm until it determines its rank.

One tool which has been used to break symmetry in this setting is randomization [V84]
[MR85] [ADKP87a]. Basicly, each of the nodes in a list randomly chooses to be a by-passing
element or a by-passed element. After this random choice, if a by-passing element points to a
by-passed element then the by-passing element's next pointer is composed. The issues which
arise in the application of randomization to list-ranking are ensuring that a fixed fraction of the
elements are jumped over in each step and ensuring that the load is balanced among the
processors. Often, this load-balancing can be performed with an application of parallel prefix.

A deterministic variant of randomization which has been used to break symmetry in list-
ranking and other problems is deterministic coin-tossing [CV86a]. This is a technique which
uses processor identifiers to deterministically break symmetry in the list-ranking setting by
determining a ruling set: Given a linked list of $n$ elements and an integer $k$ ($1 < k \leq n$), a
$k$-ruling set $R$ is a subset of list elements such that given an element $a$ in $R$, i) no other element
of $R$ is adjacent to $a$, and ii) the length of the path to another element of $R$ is at most $k$. A
$k$-ruling set is an independent set whose size is a fixed fraction of the size of the list
(i.e. $|R| \geq n/(k+1)$). Since a $k$-ruling set decomposes a list into sublists of length at most
$k+1$, a $k$-ruling set can be used to define a $(k+1)$-colouring on its list. In particular, a 2-ruling
set defines both a 3-colouring and a maximal independent set for its list. The construction of
subdivision hierarchies in chapter 4 and the computation of fractional and maximal independent
sets in planar graphs [DaK87b] makes extensive use of deterministic coin-tossing. As further
discussed in chapter 4, the work of Goldberg et al [GPS87] and Hagerup et al [HCD87] also
makes use of deterministic coin tossing to break symmetry in a number of colouring and
independent set applications.
When items are to be ordered and they are presented with comparable keys (but no further information) a general sorting algorithm must be used. Until recently, the only optimal parallel algorithm for sorting was the AKS-sorting network of Ajtai, Komlos and Szemeredi \cite{AKS83}. Although their algorithm exhibits optimal speedup asymptotically, their construction uses expander graphs and thus has rather large constant factors. This has been simplified considerably by Cole \cite{C86b}, who presents a parallel merge sort based on a cascading technique which can sort \( n \) numbers in \( O(\log n) \) time (with smaller constants than the AKS-sorting network) employing \( n \) CREW (or in a slightly different construction, EREW) processors. Cole's algorithm has also been used as the basis for a number of geometric algorithms such as 3-dimensional maxima \cite{G87a} and convex hull construction \cite{CG88}.

For some parallel algorithms, it may be possible to reduce the number of processors required without an asymptotic increase in the time required. If a parallel computation is designed such that many processors are idle during much of the computation then it may be possible to employ Brent's Scheduling Principle \cite{B74} to effect a processor reduction. This exploits the fact that a single processor can simulate the work performed by a number of other processors. Briefly stated, any parallel computation which runs in \( s \) steps with a total of \( t \) operations can be simulated using \( p \) processors in time \( O(\lfloor \frac{t}{p} \rfloor + s) \). This principle assumes that the overhead of assigning processors to their tasks is negligible. Therefore, in an application of Brent's principle, it is sometimes necessary to dynamically reallocate the tasks in a load-balancing phase within each stage. The sequential subsets technique \cite{G87a} is a form of Brent's principle in which all task rescheduling takes place at the beginning (or the end) to reduce the problem size to match the number of processors available.

We turn to some parallel algorithm tools and design techniques which have been used specifically for geometric applications. A parallel algorithm design technique which has been applied to the convex hull construction problem \cite{AG86a} \cite{G87a} \cite{ACGOY88}, closest point pair problem \cite{G87a} and the cooperative search algorithms presented in chapter 2 is the
\(\sqrt{-}\text{subdivision technique.}\) Although many applications of this subdivision technique have exploited the availability of concurrent read, Miller and Stout [MS88] have applied this approach to the 2-dimensional convex hull problem using an EREW model of computation to obtain an \(n\) processor \(- O(\log n)\) time construction algorithm. In his thesis [G87a], Goodrich refers to the \(\sqrt{-}\text{-subdivision technique as an instance of what he calls many-way divide and conquer.}\)

We describe its usage as applied to the (upper) convex hull construction problem for \(n\) points with the availability of \(n\) CREW processors [AG86a] [G87a] [ACGOY88]. Presort the point set by \(x\)-coordinate, subdivide the points into \(\sqrt{n}\) equally sized subsequences and recursively find the upper hull of each. Since \(\begin{pmatrix} \sqrt{n} \\ 2 \end{pmatrix}\) is \(O(n)\), we can assign each processor to a constant number of subsequence combinations and find all the upper common tangents using a sequential algorithm [OvL81] in \(O(\log n)\) time. A (somewhat involved) application of sorting and list ranking serves to combine these hulls and common tangents into the complete upper hull in a further \(O(\log n)\) time. If \(T(n)\) denotes the time required by this algorithm to construct the upper convex hull on \(n\) points using \(n\) CREW processors then, for some constant \(c\),

\[
T(n) = c \log n + T(\sqrt{n}) = O(\log n).
\]

The application of this technique to the cooperative search problems in chapter 2 differs somewhat from its usage in the convex hull algorithm described above. The properties exploited by both are (i) \(\begin{pmatrix} \sqrt{k} \\ 2 \end{pmatrix}\) = \(O(k)\) and (ii) \(\log \sqrt{k} = (\log k)/2\). The implication for the search application is that it is asymptotically equivalent to reduce the length of a search sequence by a factor of \(\sqrt{k}\) as by a factor of \(k\).

We close this section by considering a standard one-dimensional search problem, the problem of looking up a value in a sorted table of \(n\) elements. The familiar binary search algorithm provides the sequential \(O(\log n)\) time solution to this problem, optimal for the RAM model of computation. Initialize the search range to be the entire table. At each step compare
the query value to the middle element of the search range and use the result of that comparison to reduce the search range by half. Continue until the search range is reduced to two or fewer elements.

The parallel extension of binary search depends strongly on whether or not concurrent read is available. We assume a $k$ processor CREW PRAM. Again, initialize the search range to be the entire table. Subdivide the table into $k$ equally-sized non-overlapping subsequences and assign each processor to a subsequence. Each processor can determine in constant time whether or not its subsequence contains the query value. The unique succeeding processor writes the index limits of its subrange into a predetermined memory location and the other processors read them using the concurrent read facility. Repeat this subdivision step until the search range is reduced to a constant size.

Each iteration reduces the size of the search range by a factor of $k$, therefore this $k$-ary search algorithm runs in $O(\log n/(1 + \log k))$ parallel time. Snir [S85] has provided a matching lower bound for this problem showing that this algorithm is optimal for the CREW model. Snir has also provided a $\Omega(\log n - \log k)$ lower bound for this problem in the absence of concurrent read. Essentially, there is no way of broadcasting the reduced range to all the processors in less than $O(\log k)$ parallel time. Therefore in this instance, the only benefit from parallelism comes from reducing the search range by a factor of $k$ in the first iteration. The unique succeeding processor executes the sequential binary search algorithm within its subrange to locate the query value. In chapter 2, we relate the problem of determining convex polygon separation and the problem of constructing convex polygon mutual tangents to the sorted table look-up problem and $k$-ary search.
1.4 Definitions & Representations

We assume general familiarity with standard graph theoretic concepts and terminology [BM76] and with basic Euclidean geometric concepts and objects (see [PS85] pp. 17-26 for a succinct introduction). We denote by \( n \) the number of vertices of a typical input geometric object: subdivision, polygon, polyhedron. Since we deal exclusively with graphs (including planar graphs) whose edge set is linear in the size of the vertex set, the size of a graph can be adequately described by the single parameter \( n \), the number of vertices.

In general, we define a subdivision as a partition of a geometric space into connected sets. This is naturally tied to the dimension of the geometric space: a one-dimensional partition induces a linear subdivision, a two-dimensional partition induces a planar subdivision and a three-dimensional partition induces a spatial subdivision. The point location or subdivision search problem can be stated as, "Given a subdivision and a query point, identify the partition (or facet) of the subdivision which contains the query point." A major (in many cases dominant) component of a number of geometric algorithms is the solution of a (possibly constrained) subdivision search problem [PS85].

For example, a special planar subdivision used for closest point queries is the Voronoi Diagram [PS85]. Given a set of vertices in two-dimensions, we associate with each vertex a convex (possibly unbounded) polygon which corresponds to the region of the plane comprised of points which are closer to that vertex than any other. Given a query point, the vertex closest to the query point can be identified by performing a subdivision search on the Voronoi diagram.

More specifically, a (planar) subdivision is is a partition of the plane into regions bounded by straight edges. A bounded subdivision is implicitly given by an embedded planar graph describing the face boundaries. Planar subdivision search involves identifying the face of a given subdivision occupied by a given point. A special case of this problem, the point-polygon
containment problem, is the instance where the subdivision is defined by a polygon and the query point is interior or exterior to the polygon. This special case arises frequently in computer graphics applications [FvD82].

There are several candidate representations for a planar subdivision which are linearly equivalent in the sequential setting. In general, the representation translation routines which run in linear sequential time can be implemented in logarithmic time with optimal speedup (with respect to these same routines) by employing an $O(\log n)$ time–$n/\log n$ processor list-ranking algorithm. We assume as input the Doubly Connected Edge List (DCEL) [MP78] [DK90] in which the graph $G_S$ underlying a planar subdivision $S$ is represented as $E$, an array of edges. Each edge $e_1$ in $E$ has associated with it six pointers, one each to its originating and terminating vertices $v_1$ and $v_2$, one each to its two incident faces $f_1$ and $f_2$, a pointer to the edge $e_2$ incident on $v_1$ which is first counter-clockwise from $e$ and a pointer to the edge $e_3$ incident on $v_2$ which is first counter-clockwise from $e$. Many equivalent representations are derivable easily from the DCEL. For example, the ordered list of edges incident on a vertex or the edge ring delimiting a face can be derived in linear sequential time or in $O(\log n)$ time employing $n/\log n$ processors using list ranking.

Throughout this thesis, all polygons and polyhedra considered are convex. In two dimensions, we assume a natural presentation of the input; the vertices of a convex polygon provided in an array in clockwise order of their appearance on the boundary. As described in chapter 2, this representation contains enough structure to provide the hierarchy of polygons used in our algorithms without preprocessing. This is in contrast to the corresponding polyhedral representation in which each of the polyhedron hierarchies must be constructed explicitly (cf. chapter 5).

We exploit the fact that the surface of a convex polyhedron is isomorphic to some planar subdivision. Both are surfaces of genus 0 and thus both can be described by planar graphs.
This relationship can be visualized by choosing any face $f$ of a convex polyhedron $P$ and stretching the graph defined on the vertices and edges of $P$ out onto the plane such that each face of $P$ corresponds to a bounded region of the plane and $f$ corresponds to the outer (unbounded) region of the plane. Therefore, Euler's formula (cf. chapter 4 and [BM76]) applies to both the surface of a convex polyhedron and a planar subdivision. We assume the DCEL representation defined above for input polyhedra. As outlined in the next section, the geometric hierarchy paradigm explored here exploits the relationship between the surface of a convex polyhedron and a planar subdivision to provide point-polyhedron separation and planar subdivision point location algorithms which have the same basic design.

1.5 The Geometric Hierarchy Paradigm

This thesis examines the use of a hierarchical data structuring technique for the solution of geometric problems on a parallel model of computation. The general approach explored here involves using a sequence of progressively simpler descriptions of the geometric object under study. Queries are typically answered by starting with the simplest description and increasing the detail within the region of interest by stepping through the sequence. This can be thought of as a higher dimensional variant of binary search viewed as a sequence of tests, each of which further restricts where a given query value can occur.

It is important to distinguish between this general hierarchical approach and the description of any individual data structure which is defined using this approach. For example, although the region tree structure of chapter 3 and the subdivision hierarchy of chapter 4 have been designed using the same hierarchical approach, the structures themselves appear to be quite different. For simplicity, the hierarchical data structure presented within this section is
described with respect to its application to sequential search for convex polyhedra. In chapter 5, we see how to modify this data structure's definition to support parallel search.

The term geometric hierarchy which is used to describe the type of data structure defined here is particularly apt. The 'geometric' descriptor can be regarded as referring not only to the type of objects represented but also the fact that the sizes of the hierarchy elements form a geometric sequence. Since this ensures that the overall size requirement of the hierarchy is a constant factor of the size of the input, this geometric sequence results in (optimal) linear storage and linear sequential preprocessing time. As well, this geometric hierarchy paradigm appears to be the only algorithmic approach to date which can exploit the topological relationship between planar subdivisions and convex polyhedra to produce point location and separation algorithms which have the same basic design.

As applied to the representation of a convex polytope, the geometric hierarchy consists of a sequence of (progressively simpler) polytopes; the first element of the sequence is the original polytope and the last element of the sequence is a polytope with a constant number of faces. Excepting the original polytope, each element of the sequence is derived from its predecessor by removing a set of independent vertices and forming a new (simpler) polytope on the resulting vertex set. The set of independent vertices is chosen such that (a) the size of the set is a fixed fraction of the size of the entire vertex set and (b) the change between consecutive sequence elements due to any single vertex removal is of constant size. Condition (a) ensures that the number of sequence elements is logarithmic in the (original) number of vertices. Condition (b) ensures that given a search query's answer for the $i+1^{st}$ element of the sequence, there is a constant amount of 'detail' to expand and examine in the $i^{th}$ element of the sequence and therefore the search query's answer for the $i^{th}$ element of the sequence can be determined by a single processor in constant time. Since the last element of the sequence has a constant number of faces and a single step through the sequence takes constant time, therefore a search query can be answered in time proportional to the length of the sequence.
More formally, there exist positive constants $\alpha > 1$ and $\beta$ such that for any convex polytope $P$ with vertex set $V(P)$, there is a (nested) sequence of convex polytopes $H(P) = P^0$, $P^1$, ..., $P^N$ (called a geometric hierarchy representation of $P$) satisfying:

i) $P^0 = P$;

ii) $|P^N| = 3$;

iii) $V(P^{i+1}) \subseteq V(P^i)$ and $|P^{i+1}| < |P^i|/\alpha$ and

iv) for any vertex $p_j \in P^i$ either $p_j \in P^{i+1}$ or degree($p_j$) in $P^i$ is less than $\beta$ and all of $p_j$'s neighbors in $P^i$ are in $P^{i+1}$.

The height of the hierarchy is defined as $N$, the length of the sequence. Conditions (ii) and (iii) ensure that the height of the hierarchy $N = O(\log n)$ and condition (iv) ensures that the vertices of $P^i$ not in $P^{i+1}$ are each low-degree and form an independent set in $P^i$. The algorithms which use this hierarchical representation follow the general hierarchical approach described above: Answer the query for $P^N$. Having answered the separation query for hierarchy element $P^{i+1}$, step to (its predecessor) element $P^i$ and use the previous answer and a small number of tests to answer the query for element $P^i$. Continue stepping until the query has been answered for $P^0 = P$.

As applied to convex polytopes, this hierarchical representation has been used to solve a number of geometric separation problems on a sequential model of computation [DK82] [DK83] [DK85] [DK90]. An equivalent hierarchical representation has also been applied to the geometric search problem of locating a point in a planar subdivision on a sequential model of computation [K83].

In this thesis, we investigate how this type of hierarchical representation can be constructed and utilized using a parallel processing model of computation. For this hierarchy's application to convex polygons, the question of construction is straightforward. The polygonal hierarchy
is implicit within a standard array representation, the question of exploiting parallelism in the search simply becomes one of being able to take 'larger' steps through the hierarchy. This can be thought of as a variant of the generalization of sequential binary search to k processor k-ary search as described in section 1.4. The relationship between convex polygon separation and binary search is elaborated further in chapter 2.

For this hierarchy's application to convex polyhedra/planar subdivisions, the hierarchy must be explicitly constructed in a preprocessing phase. As described in chapter 4, parallelism can be applied to the construction of the standard subdivision hierarchy. The issue here becomes one of 'breaking symmetry' to determine a sufficiently large (low-degree) independent set to remove to construct an element of the hierarchy from its predecessor.

As with the polygonal hierarchy, the question of exploiting parallelism to perform cooperative search in the subdivision/polyhedral hierarchy involves being able to take 'larger' steps through the hierarchy. As opposed to being implicitly available within the polygonal hierarchy, this requires an explicit construction to augment the standard hierarchy to enable efficient multi-processor query. Interestingly, this has motivated a re-examination of the Lipton-Tarjan planar point location construction [LT77] (the first point location scheme to simultaneously achieve linear preprocessing time, linear storage space and logarithmic query time) and the separator theorem's relationship with the standard subdivision hierarchy construction (cf. chapter 5).

1.6 Previous Related Work

This section surveys related work to enable the research presented within this thesis to be placed within the context of the development of computational geometry (on sequential and parallel models of computation) and related disciplines. More extensive and detailed
discussions of related research appear in the appropriate chapters. As described in the last section, the work presented here builds on the geometric hierarchy paradigm studied extensively for a variety of problems on a sequential model of computation by Dobkin and Kirkpatrick [DK83] [K83] [DK85] [DK90].

Hierarchical decompositions can be useful for dealing with complex systems and representations [S68] and have been used extensively in geometric modelling [R80] [RV82] and computer graphics particularly in the solution of the visible surface problem [SW88a] [SW88b] [FvD82]. Within their application in computer graphics, there is a dichotomy between image space hierarchies and object space hierarchies. Image space hierarchies exploit the discrete nature of picture elements which make up an image and area coherence within an image to superimpose a hierarchical organization. This approach is typified by the quadtree and octree data structures [SW88a]. The geometric hierarchy paradigm is more closely related to the concept of object space hierarchies.

Object space hierarchies exploit object coherence within a geometric model and are useful in settings requiring arbitrary precision. Clark [C76] proposed the use of an object space hierarchy to structure domain information and to limit the amount of information under consideration at any particular time. He suggested that search algorithms take the form of a recursive descent through the hierarchy. This approach was developed by several researchers including Eastman [E77], Eastman and Lividini [EL75], Reddy and Rubin [RR78], and Rubin and Whitted [RW80]. In conjunction with some of these approaches, the geometric hierarchy described in by Kirkpatrick [K83], Dobkin and Kirkpatrick [DK90] and outlined in section 1.5 has been applied to the computer graphics visible surface problem and similar problems arising in the computer modeling of room acoustics [DaKW82] [D82] [DaKW85].

It appears that the first published attempt to address the solution of computational geometry problems using a parallel model of computation in a comprehensive way was Chow's thesis
She presented algorithms for rectangle intersection, planar point location data structure (construction and search), convex hull construction (in 2- and 3-dimensions) and Voronoi diagram construction for the PRAM and cube-connected-cycles models of computation. Unfortunately, her results were not widely distributed and as a result, rather than providing the impetus to combine further computational geometry and parallel computation results, her pioneering contributions went largely unrecognized.

The first widely distributed comprehensive examination of computational geometry problems in a parallel setting appeared in the paper of Aggarwal et al [ACGOY85] [ACGOY88]. Aggarwal et al presented a number of techniques and tools which have since laid the foundation for the study of parallelism in computational geometry. Among their results were parallel solutions to such familiar geometric problems as convex hull construction (in 2 and 3 dimensions), Voronoi diagram construction (in 2 dimensions) and closest point search, and segment intersection.

Within Aggarwal et al's method for the construction of Voronoi diagrams [ACGOY85], there is a subproblem which involves point location in a planar subdivision defined by vertices and edges forming an embedded complete binary tree. Their initial solution to this subproblem was a time and space bottleneck within their algorithm and offered fertile ground for research. This resulted in the region tree algorithm of chapter 3 which produced the $O(\log^2 n)$ time–n processor Voronoi diagram construction algorithm presented there and in [DaK87a]. In their revised paper, Aggarwal et al [ACGOY88] presented an improved construction (differing from ours) which resulted in another $O(\log^2 n)$ time-n processor Voronoi diagram construction algorithm.

---

1 In fact, the 3-dimensional convex hull construction algorithm originally presented by Aggarwal et al [ACGOY85] was not as efficient as Chow's original algorithm.
Nicholas Pippenger [P87] pointed out the similarities between our algorithm for region tree preprocessing [DaK87a] and the tree contraction work of Miller and Reif [MR85]. This connection was explored by the UBC Parallel Algorithms Discussion Group and the region tree point location algorithm was developed into a general tree contraction algorithm based on a reduction to list-ranking [ADKP89]. The presentation in chapter 3 reverses the historical thread by beginning with the general tree contraction algorithm and then discussing the geometric application.

The next natural question to examine was search in general planar subdivisions, specifically to produce a parallel construction of the subdivision hierarchy of Kirkpatrick. As described in chapter 4, this was accomplished by an algorithm which given a hierarchy element constructs the next element in the sequence by removing a 'low-degree' independent set. This algorithm first identifies a subgraph S of the input planar graph in which all the vertices of S have degree bounded by some constant. This 'low-degree' subgraph is then decomposed into edge-disjoint chains and either randomization or Cole and Vishkin's deterministic coin-tossing technique [CV86a] is used to construct an independent set. This algorithm was also applied to a polyhedron convex union problem resulting in a parallel 3-d convex hull construction algorithm that is both more efficient and simpler than the ones presented by Aggarwal et al [ACGOY85] [ACGOY88].

The planar graph decomposition technique used in the subdivision hierarchy construction algorithm described above can also be applied to the problem of determining maximal (and fractional) independent sets in planar graphs [DaK87b]. Independently, Hagerup et al [HCD87] and Goldberg et al [GPS87] developed similar planar graph decomposition techniques for graph colouring and independent set applications (cf. section 4.2). These results all built on the seminal work of Cole and Vishkin [CV86a] both for their list ruling set algorithms and their application of Brent's scheduling principle [B74] to obtain optimal speedup with respect to known linear time sequential algorithms.
Atallah and Goodrich [AG86b] [G87a] developed further the parallel plane sweep algorithm, one of the techniques proposed by Aggarwal et al. Among other results, this was used to produce a planar subdivision point location data structure which can be constructed in $O(\log n)$ time using $n$ CREW processors employing $O(n \log n)$ space and can answer point location queries in $O(\log n)$ time. This data structure is constructed using a parallel fractional cascading technique which was presented in more detail as part of Goodrich's thesis [G87a]. Goodrich's thesis also described several other techniques and algorithms (notably "many-way divide and conquer" as outlined in section 1.3) which were elaborated on in several other papers [AG86a] [G85] [ACG89].

Atallah and Goodrich's work on convex polygon common tangents and separation [AG88] suggested another opportunity to explore the geometric hierarchy paradigm. They present what we call cooperative algorithms for constructing convex polygon common tangents and determining convex polygon separation which run in $O(\log^2 n/(1+\log^2 k))$ time using $k$-CREW processors. Thus, their algorithms run in constant time using a quasi-linear ($n^\alpha$ for any $\alpha > 0$) number of processors but run in $O(\log^2 n)$ time when only a constant number of processors are available. We built on these results and used the geometric hierarchy paradigm to produce the $O(\log n/(1+\log k))$ time $k$-CREW processor algorithms presented in chapter 2. A reduction from the sorted table lookup problem to these convex polygon problems combined with the lower bounds on searching on the PRAM model presented by Snir [S85] provided the matching lower bounds for the problems of constructing convex polygon mutual tangents and determining convex polygon separation.

Having produced optimal cooperative algorithms for the 2-d tangents and separation problems, the next natural question to ask was: Is this type of cooperative search possible for convex polyhedra (or planar subdivisions)? We approached the problem with three goals: i) feasibility (Is cooperative planar subdivision search possible?); ii) constructibility (Can a data structure which supports cooperative subdivision search be constructed?) and
iii) efficiency (Is there an efficient algorithm for constructing a space-efficient data structure for performing cooperative subdivision search?)

This led to the results described in chapter 5. The dynamic hierarchy data structure described there is constructed by augmenting an "ordinary" subdivision hierarchy with static hierarchies, data structures each of which is attuned to cooperative search for a particular number of processors. Interestingly, the static hierarchies (including the hierarchy used for sequential search) are built using a planar separator theorem employing a construction somewhat simpler than that presented in Lipton and Tarjan's application paper [LT77].

1.7 Thesis Outline

This thesis can be regarded as a case study in the applicability of a particular problem solving methodology, the Geometric Hierarchy paradigm, to the solution of geometric problems on a parallel model of computation. Parts of this thesis involve a direct application of the geometric hierarchy representation to these problems, others involve the use of the geometric hierarchy methodology to design algorithms and data structures specifically for use in the parallel setting.

In chapter 2, the geometric hierarchy paradigm is applied to the convex polygon separation and mutual tangent problems on a parallel model of computation. The cooperative algorithms presented for these convex polygon queries make direct use of the geometric hierarchy representation. The hierarchical paradigm is applied by employing an accelerated polygon hierarchy to produce cooperative algorithms which answer separation and separating/common tangent queries for n-vertex convex polygons in $O(\log n/(1 + \log k))$ time using $k$ CREW processors for $1 \leq k \leq n$. To address some of the implementation considerations associated with the cooperative algorithms presented, we describe a parallel algorithm which constructs
the 2-dimensional convex hull of a point set of size $n$ sorted by $x$-coordinate in $O(\log n)$ parallel time employing $n/\log n$ CREW processors\(^1\). Finally, exploiting the relationship between a linear subdivision and the boundary of a convex polygon, a reduction of the sorted table problem to convex polygon common tangents and separation and Snir's lower bounds for the sorted table lookup problem on a PRAM [S85] provide the matching lower bounds for our algorithms.

In chapter 3, the geometric hierarchy paradigm is applied to point location problems in certain restricted types of planar subdivisions which arise naturally in a number of applications. The algorithms and data structures presented in this chapter are not a direct application of the geometric hierarchy representation. However, the geometric hierarchy paradigm was pivotal in the development of a data structure described in chapter 3 which evolved into a general tree-contraction algorithm. The central result of this chapter is an effective reduction of the tree contraction problem to the list-ranking problem. Employing the list-ranking algorithms of Cole and Vishkin [CV86b] or Miller and Anderson [AM88] and an application of Brent's Scheduling Principle, the tree contraction sequence can be constructed for an $n$ node rooted ordered binary tree in $O(\log n)$ time using $n / \log n$ EREW processors.

For a geometric application of tree contraction, we define an abstraction of a class of planar subdivisions called region trees and show that tree contraction can be used to preprocess a region tree for fast point location. Several natural forms of restricted planar subdivision search can be reduced directly to region tree search. A subdivision search subproblem arising in the parallel construction of general (closest-point) Voronoi diagrams can be formulated as a region tree search problem. This results in a $O(\log^2 n)$ time algorithm employing $n$ CREW processors for the construction of general Voronoi diagrams on $n$ vertices based on the algorithm described by Aggarwal et al [ACGOY85]. Region trees can also be applied to the construction

\[^1\] This result was previously obtained using different techniques by Goodrich [G87b] and Wagener [Wa85].
of a point location data structure for furthest point Voronoi diagrams and Voronoi diagrams defined on a convex set of points.

In chapter 4, the geometric hierarchy paradigm is applied to the problem of efficiently preprocessing a planar subdivision for fast sequential search. This is achieved by giving a direct parallel construction of the geometric hierarchy representation as applied to subdivision search by Kirkpatrick [K83]. Employing deterministic coin-tossing (cf. section 1.3 and [CV86a] [CV86b]) or randomization to break symmetry, the subdivision hierarchy H(S) for an n-vertex subdivision S can be constructed in \(O(\log n \log^* n)\) (respectively, \(O(\log n)\) expected) parallel time using a deterministic (respectively, randomized) algorithm. This subdivision hierarchy uses \(O(n)\) space and supports \(O(\log n)\) time sequential point location.

We also exploit the relationship between the boundary of a convex polyhedron and a planar subdivision as described in section 1.4. Once the polyhedral hierarchy has been constructed, using essentially the same algorithm for constructing the subdivision hierarchy, it can be used to answer a number of intersection and separation queries either sequentially [DK90] or in parallel using a concurrent read facility. Aggarwal et al [ACGOY85] present a parallel algorithm for the construction of 3 dimensional convex hulls which is based on the general divide and conquer approach of Preparata and Hong [PH78]. The polyhedral hierarchy representation can be exploited (within the same basic design) to construct the 3 dimensional convex hull of an n-vertex point set in \(O(\log^2 n \log^* n)\) (respectively, \(O(\log^2 n)\) expected) time employing \(n\) CREW processors using a deterministic (respectively, randomized) algorithm. Following Dobkin and Kirkpatrick [DK90], the polyhedral hierarchies can also be employed for determining the separation of two convex polyhedra in the time of the polyhedral hierarchy construction.

In chapter 5, algorithms are presented for augmenting the (ordinary) subdivision hierarchy representation to enable cooperative algorithms to solve planar point location queries for
$n$-vertex planar subdivisions in $O(\log n/(1 + \log k))$ time using $k$ CREW processors ($1 \leq k \leq n$). This augmentation is performed in such a way that the entire data structure only requires linear space. This is described in two stages: first we demonstrate how a planar separator theorem can be used to construct a \textit{static hierarchy}, one which supports cooperative search by $k$ processors for a fixed value of $k$ ($1 \leq k \leq n$). We then demonstrate how the ordinary subdivision hierarchy can be augmented with a number of static hierarchies resulting in an efficient \textit{dynamic hierarchy}, one which uses only $O(n)$ space and supports $O(\log n/(1 + \log k))$ time point location using $k$ CREW processors for any value of $k$ ($1 \leq k \leq n$). This is not a direct application of the geometric hierarchy representation but is an extension of that structure designed specifically for cooperative search.

Again, we exploit the relationship between a planar subdivision and the boundary of a convex polyhedron in the design of algorithms which use the augmented hierarchical data structure to answer $n$-vertex polyhedron-linear subspace separation queries in $O(\log n/(1 + \log k))$ time and polyhedron-polyhedron separation queries in $O((\log n/(1 + \log k))^2)$ time.

In chapter 6, we present a high level discussion of our results and explore possible directions for further work.
Chapter 2: Linear Subdivision Search and Convex Polygon Hierarchies

2.1 Introduction and Related Work

In this chapter, the geometric hierarchy paradigm is applied to the convex polygon separation and mutual tangent problems. This is a direct application of the geometric hierarchy representation (cf. section 1.5 and [DK90]) to the solution of these convex polygon problems on a parallel model of computation.

Determining the separation of two convex polygons is a generalization of detecting their intersection; it involves the identification of a witness to the minimum distance separating them. If the polygons do intersect, the separation is zero and the witness is a point in their intersection.

A separating (respectively, common) tangent of two convex polygons P and Q is a line tangent to both polygons such that P and Q lie in different half-planes (respectively, the same half-plane). The separating tangents delimit the set of possible separating lines for P and Q. The common tangent in which P and Q both lie in the half-plane below (respectively, above) the tangent is known as their upper (respectively, lower) common tangent. Determining the common tangents of two convex polygons is a step in forming the convex hull of their union.

We present cooperative algorithms for these convex polytope queries which make implicit or explicit use of a hierarchical representation. The hierarchical paradigm is applied by exploiting the relationship between a linear subdivision and the boundary of a convex polygon to produce cooperative algorithms which answer separation and separating/common tangent
queries for \(n\)-vertex convex polygons in \(\Theta(\log n/(1 + \log k))\) time using \(k\) CREW processors for \(1 \leq k \leq n\).

These cooperative search algorithms are faster than previously known algorithms which address the same problems and provide constant time solutions when a quasi-linear \((n^\alpha\) for any \(\alpha > 0\)) number of processors is available. They also have the interesting property that as the number of processors available decreases, the time bound increases smoothly to match the best known sequential time bound with a single processor. Although these algorithms are applicable to the entire range of \(k\), occasionally minor modifications are necessary to ensure correct behavior for small values of \(k\). When these modifications are straightforward, we omit their details in the interest of an uncluttered presentation.

In solving the polygon separation and mutual tangent problems, we exploit the relationship between the exterior of a convex polygon and a linear subdivision. A \textit{linear subdivision} is a partitioning of the real line into contiguous segments. Given a real value, the linear subdivision search problem involves the identification of the segment which contains that value. Recall that the sorted table look-up problem is defined as follows: given a sorted table of \(n\) real numbers \(x_1, x_2, \ldots, x_n\) and a search key \(x_s\), return the index \(i\) such that \(x_s \in [x_i, x_{i+1}]\) (with \(x_0 = -\infty\) and \(x_{n+1} = \infty\)). Clearly, the linear subdivision search problem is equivalent to the sorted table lookup problem.

Snir [S85] has shown that in the CREW model of computation, with \(p\) processors available \((1 \leq p \leq n)\), the sorted table look-up problem has a time bound of \(\Theta(\log n/(1 + \log p))\). In the absence of concurrent read, the sorted table look-up problem has time bound \(\Theta(\log n - \log p)\). (This is a key result which demonstrates that processors with a concurrent read facility are strictly more powerful than processors without concurrent reads.) In section
2.6, we show that the sorted table lookup problem can be reduced implicitly to convex polygon common tangents and separation providing lower bounds for the problems addressed in this chapter.

To motivate our approach, consider a special case of planar subdivision search: Given an n-vertex convex polygon P and a point q, determine whether or not q lies inside P. Ignoring degeneracies, this question can be answered by placing a horizontal line L through q, finding the (0 or 2) intersections of L with P, and testing q's position on L with respect to these intersections. By decomposing P into two chains which are monotone with respect to the y-axis, we can use a natural extension of linear subdivision search to answer the line/polygon intersection (hence point-in-polygon) question in $O(\log n/(1 + \log p))$ time using p processors.

In early work, Chazelle and Dobkin [CD80] made the observation that (given appropriate representations) it was easier to detect the intersection of convex objects P and Q than to construct the intersection of P and Q. They provided sublinear algorithms for a number of polytope intersection detection problems. Building on their work, Overmars and van Leeuwen [OvL81] and Edelsbrunner [E85] have presented logarithmic time sequential algorithms for convex polygon common tangents and convex polygon separation, respectively.

Viewing the boundary of each polygon as a sequence of values, all of the algorithms cited above involve a 'binary search' approach; within each iteration, the midpoints of the remaining sequence of vertices of each polygon are compared. Then, a case analysis is applied to limit the remaining search to one half of the sequence associated with one of the polygons. The iterations continue until one of the two sequences is reduced to a constant size. Therefore, there are a logarithmic number of constant time iterations. Although these algorithms do not
seem to admit to straightforward parallel implementations in our model, their binary search approach forms the basis for the k-way search algorithms developed here.

Dobkin and Kirkpatrick [DK82][DK83][DK85][DK90] and Avis, El Gindy and Seidel [AES85] have also presented efficient sequential algorithms for a number of polygon and polyhedron separation problems. Some of the results presented here can be viewed as non-trivial parallel generalizations of their work.

On a parallel model of computation, the solutions which Atallah and Goodrich [AG88] present for the common tangents and separation problems are described for a quasi-linear number of processors. However, the natural extensions of their algorithms for n-vertex convex polygons run in \( O(\log^2 n/(1 + \log k)^2) \) parallel time using \( k \) CREW processors. Therefore, although they provide constant time algorithms (with a sublinear number of processors) for the problems which they address, they do not achieve the theoretical maximum speed-up of \( \log k \) (cf. section 2.6). In particular, since they embed a point-polygon test within each iteration of their polygon-polygon algorithms, as \( k \) the number of processors available decreases and approaches one, the natural extensions of their algorithms run in \( O(\log^2 n) \) sequential time. We extend the basic design of their algorithms using convexity, monotonicity and properties of the inner polygonal hierarchy of Dobkin and Kirkpatrick [DK82].

The results reported here unify and extend the work cited above. In section 2.2, we present details of the hierarchical representations which are used in the design of our algorithms. In sections 2.3 & 2.4, we describe algorithms which use a polygonal hierarchy representation to solve separation and mutual tangent queries for n-vertex convex polygons in \( O(\log n/(1 + \log k)) \) time using \( k \) CREW processors (\( 1 \leq k \leq n \)). In section 2.5, we illustrate
some of the implementation considerations associated with the convex polygon common
tangent algorithm by presenting an algorithm for computing the 2 dimensional convex hull of
an n-vertex sorted point set which runs in $O(\log n)$ time using $n/\log n$ CREW processors. In
section 2.6, we present a reduction of the sorted table look-up problem to the polygon
separation and mutual tangent problems which, combined with Snir's PRAM lower bounds on
the sorted table look-up problem, provides the lower bounds for these problems.

2.2 Polygon Hierarchies

All polygons considered are convex and are assumed to be presented in an array listing the
vertices in clockwise order. We note that conversion from a list to this representation is
straightforward using list ranking (cf. section 1.3). For simplicity, we assume that no three
vertices of a given polygon are collinear. We denote the set of vertices of polygon P by $V(P)$.
*Segment* $ab$ is the line segment whose endpoints are distinct vertices $a$ and $b$. *Line* $ab$ is the
line which contains segment $ab$. *Ray* $ab$ is the half-line directed from $a$ which contains segment $ab$.

A convex polygon $P$ can be defined as the intersection of a set of bounding half-planes.
Hence, we define the edges as having a direction (clockwise) around the polygon such that
each edge $ab$ (defined on distinct vertices $a$ and $b$) defines two closed bounding half-planes $H^+_{ab}$
(on the right side of the edge from $a$ to $b$) and $H^-_{ab}$ (on the left side of the edge from $a$ to $b$).
Note that within this definition, $a$ and $b$ need not be consecutive vertices of $P$. This notation is
generalized to define corresponding closed half-planes $H^+_r$ and $H^-_r$ with respect to any ray $r$.

Our algorithms make use of the geometric hierarchy representation described in section 1.5.
In two dimensions, this hierarchical representation can be motivated by a simple observation:
Given a convex polygon $P$, any subsequence of $P$'s vertices defines another convex polygon completely contained within $P$. There exist positive constants $\alpha$ and $\beta$, such that for any convex polygon $P$ with vertex set $V(P)$, a (nested) sequence of convex polygons $P^0, P^1, \ldots, P^N$ is said to be an *(inner) polygon hierarchy* of $P$ if

1. $P^0 = P$;
2. $|P^N| = 3$;
3. $V(P^{i+1}) \subseteq V(P^i)$ and $|P^{i+1}| < |P^i| / \alpha$ and
4. there is at most $\beta$ vertices of $P^i$ between any two consecutive vertices of $P^{i+1}$.

Again, conditions (ii) and (iii) ensure that the height of the hierarchy $N = O(\log n)$ and condition (iv) ensures that the vertices of $P^{i+1}$ are evenly selected from the vertices of $P^i$. A sequence of polygons $H(P) = P^0, P^1, \ldots, P^N$ that forms a polygon hierarchy is described simply: if the vertices of $P$ are $p_0, p_1, \ldots, p_{n-1}$, then the vertices of $P^i$ are taken to be those vertices of $P$ whose index is congruent to $0 \bmod 2^i$. This hierarchical representation has been used to solve a number of geometric separation problems in a sequential model of computation [DK90]. The algorithms which use this representation generally follow our basic paradigm: Having answered the separation query for hierarchy element $P^i$, step to element $P^{i-1}$ and use a constant number of tests to answer the query for $P^{i-1}$.

One of our goals is to demonstrate that this hierarchical representation for polygons is adapted easily for use in cooperative search queries. This is accomplished by relating the values $\alpha$ and $\beta$ in conditions (iii) and (iv) of the polygon hierarchy definition to $k$, the number of processors we have available. A sequence of polygons $H_k(P) = P^0_k, P^1_k, \ldots, P^M_k$ that forms
an accelerated polygon hierarchy attuned to the value \( k \ (2 \leq k \leq n) \) is described accordingly: if the vertices of \( P \) are \( p_0, p_1, \ldots, p_{n-1} \), then the vertices of \( P_k^1 \) are taken to be those vertices of \( P \) whose index is congruent to \( 0 \ \text{MOD} \ k \). Therefore, this accelerated polygon hierarchy has \( O(\log n/(1 + \log k)) \) elements.

Viewed explicitly in this way, this hierarchical representation can be used by \( k \) processors to solve a point-polygon separation query. Each iteration of the algorithm deals with an element of the polygon hierarchy which is a "current approximation" of the initial polygon. When a point which realizes the separation property of interest is localized, the algorithm steps to the preceding element of the hierarchy which is a refinement of the current polygon approximation. Solving a polygon-polygon separation query is analogous, the algorithm deals with a current approximation of each polygon; an iteration of the algorithm steps to the preceding element of (at least) one of the polygon hierarchies.

We maintain the invariant that when examining an element of the polygon hierarchy, only a portion of size \( O(k) \) need be considered. The algorithm initializes this invariant by examining the last element of the sequence; this element forms the coarsest approximation of \( P \) and is of size \( O(k) \). Given \( P_k^{i+1} \), from the portion of size \( O(k) \) under consideration, the algorithm identifies a constant size portion which is guaranteed to contain a point which realizes the separation property of interest. This expands into a portion of size \( O(k) \) in \( P_k^i \), and maintains the invariant.

---

1 The hierarchy used for \( k = 1 \) will be a simulation of the hierarchy for \( k = 2 \).

2 It is worth noting that Atallah and Goodrich's approach [AG88] is an informal variant of this paradigm. Their algorithms take a step by refining the region of interest dynamically using a \( k \)-way subdivision.
When a polygon \( P \) is presented in an array, all elements of the hierarchy (for all values of \( k \)) are available implicitly by indexing.\(^3\) To avoid excess notation, we denote the current hierarchy element \( P^k \) by \( \widetilde{P} \) (or, more commonly, \( \bar{P} \) when the context is clear or unimportant). At any given point within an algorithm, a vertex may have two different labels, its index in the original polygon \( P \) and its index in the current hierarchy element \( \bar{P} \). To avoid possible ambiguities, we adopt some notational conventions: The vertex \( p_i \) is the \( i \)th vertex of polygon \( P \), and the vertex \( \bar{p}_i \) is the \( i \)th vertex of the current hierarchy element \( \bar{P} \). The sequence of vertices of polygon \( P \) between \( p_s \) and \( p_t \) inclusive is denoted by \( P[p_s, p_t] \), consecutive subscripts refer to consecutive vertices on \( P \). The sequence of \( m \) vertices of interest within hierarchy element \( \bar{P} \) between \( \bar{p}_s \) (\( = \bar{p}_1 \)) and \( \bar{p}_t \) (\( = \bar{p}_m \)) inclusive is denoted by \( \bar{P}[\bar{p}_1, \bar{p}_m] \), consecutive subscripts refer to consecutive vertices on \( \bar{P} \). For example, \( \bar{P}[\bar{p}_i, \bar{p}_{i+1}] \) refers to the complete sequence of vertices on polygon \( P \) between two vertices \( \bar{p}_i \) and \( \bar{p}_{i+1} \) inclusive which are consecutive on the current hierarchy element \( \bar{P} \). In the discussion of polygon separation where the separation may be realized by a point which is not necessarily a vertex, we occasionally abuse this notation by using \( \bar{P}[\bar{p}_i, \bar{p}_{i+1}] \) to refer to the complete sequence of points on the boundary of \( P \) between two vertices \( \bar{p}_i \) and \( \bar{p}_{i+1} \) inclusive.

Given hierarchy element \( \bar{P} \), the convexity of \( P \) imposes implicit restrictions on how far \( P \) can extend beyond the edges of \( \bar{P} \): Let \( a, b \) and \( c \) be three arbitrary vertices of convex polygon \( P \) appearing in that order and consider the two chords \( ab \) and \( bc \) of \( P \). Consider the two half-planes \( H^+_{ab} \) and \( H^-_{ab} \) (respectively, \( H^+_{bc} \) and \( H^-_{bc} \)) defined by ray \( ab \) (respectively, ray \( bc \)).

**Observation 1:** \( P[a,b] \subset H^-_{ab} \) and \( P[b,a] \subset H^+_{ab} \).

---

\(^3\) In three dimensions, the corresponding polyhedral hierarchy must be explicitly constructed in a preprocessing phase (cf. chapter 5).
Observation 2: $P[c,a] \subseteq H_{ab}^+ \cap H_{bc}^+ \text{ and } H_{ab}^- \cap H_{bc}^- \cap P = \emptyset$.

Applying these observations about chords of $P$ to the edges of an element of the polygon hierarchy $\hat{P}$ implicitly defines $\hat{P}$: the star-shaped (possibly unbounded) polygon which is the union of all possible convex polygons which could have $\hat{P}$ as a hierarchy element. The boundary of the polygon $P$ must lie within the envelope of points between any hierarchy element $\hat{P}$ and its corresponding $\hat{P}$ (see figure 2.1).

The key to the efficient parallel use of the hierarchy element approximations is the judicious assignment of processors. Intuitively, this can be thought of as a direct analogue of parallel search in a sorted table [S85]; each iteration of the k-way search corresponds to a step through the implicit polygon hierarchy. This correspondence, formalized by the reduction in section 2.6, allows the design of k-processor convex polygon algorithms which are natural extensions of k-way search. Within each iteration of the natural k-way search algorithm for the sorted table lookup problem, the search key's containment in a particular subsequence can be determined by a single processor in constant time. This process is slightly more involved for the polygon tangents/separation problems but can be done efficiently by employing tests which exploit convexity and properties of a polygon hierarchy.
Within each iteration of the algorithms for polygon tangents and separation, a solution is found for the current approximations (hierarchy elements) of the polygons and then the appropriate case analysis is applied to reduce the region of interest. This can be thought of as 'electing' an active processor (the one that witnesses the hierarchy element/subsequence solution) which then applies the case reduction. The active processor then writes the reduced ranges into a predetermined location and the remaining processors use the concurrent read facility to update their information. Since only one processor is assigned to the pair realizing the hierarchy element solution, this election process avoids write contention.

As an example of our general approach, consider the following k-processor algorithm for finding $p^d$, the extremum of an n-vertex convex polygon $P$ in the direction of ray $d$: Initialize $\tilde{P}$ to be the last element of $P$'s polygon hierarchy and repeat the following until $\tilde{P}$ is reduced to a single point. Assign a processor to each vertex of the current hierarchy element $\tilde{P}^t$. Vertex $\tilde{p}_i$ realizes the extremum of $\tilde{P}^t$ in the direction of ray $d$ iff there exists a line $L$ perpendicular to $d$
such that $\tilde{p}_i \in L \cap \tilde{P}^t$, $\tilde{P}^t \subset H^t_L$ and $d$ extends to infinity in $H^t_L$. This can be determined in constant time by comparing the slopes of segments $\tilde{p}_{i-1}\tilde{p}_i$ and $\tilde{p}_i\tilde{p}_{i+1}$ with ray $d$. Given such a vertex $\tilde{p}_i$, then by Observation 2, all points of $P$ clockwise between $\tilde{p}_{i+1}$ and $\tilde{p}_{i-1}$ lie in the wedge formed by rays $\tilde{p}_i\tilde{p}_{i+1}$ and $\tilde{p}_{i-1}\tilde{p}_{i+1}$. Therefore, $p^d$ must lie in $P[\tilde{p}_{i-1}, \tilde{p}_{i+1}]$. Replace $\tilde{P}$ with the portion of hierarchy element $\tilde{P}^{t-1}$ between vertices $\tilde{p}_{i-1}$ and $\tilde{p}_{i+1}$.

This procedure uses $k$ processors to perform $O(k)$ tests in constant time for each element of the hierarchy. Essentially, this algorithm identifies the vertex of $P$ which realizes the separation between $P$ and a line $L$ perpendicular to direction $d$ placed at infinity in direction $d$. Therefore, we have immediately:

**Lemma 2.2.1:** An $n$-vertex convex polygon's extremum in a given direction $d$ can be identified in $O(\log n/(1 + \log k))$ time using $k$ CREW processors.

For the problems we consider, we present a uniform treatment and simplify aspects of our algorithms by adding an initial preprocessing step to reduce a convex polygon under consideration to a constant number of polygonal chains each of which delimits a convex region of the plane. From Dobkin and Kirkpatrick [DK83], we recall the definition of a Monotone Polygonal Sector (see figure 2.2): The boundary of a convex polygon is decomposed into two monotone polygonal chains of edges by cutting at its highest and lowest $y$ coordinates. This yields two sequences of vertices and edges in order of increasing $y$-coordinate. By convexity, any such chain will be either left- or right-oriented. Semi-infinite rays are attached to the beginning and end of each chain and the interior is included to form two (convex) monotone polygonal sectors (MPS). These rays run parallel to the $x$-axis towards $-\infty$ if left-oriented or $+\infty$ if right-oriented and define the area contained by the MPS. By Lemma 2.2.1, a convex
n-vertex polygon \( P \) can be decomposed into its left MPS \( P_L \) and its right MPS \( P_R \) in \( O(\log n/(1 + \log k)) \) time using \( k \) CREW processors.

We maintain \( MPS \ P_L \) with its vertices indexed by increasing y-coordinate as \( P_L[p_{lower_p}, p_{upper_p}] \). This denotes the convex region delimited by the sequence of edges of \( P \) from vertex \( p_{lower_p} \) to vertex \( p_{upper_p} \) augmented by infinite rays at the upper and lower bounds. Initially, the upper and lower bounds correspond to the maximum and minimum y-coordinates respectively. Within the general design of our algorithms, each iteration reduces the size of a polygonal chain under consideration by adjusting the upper and lower bounds of a MPS. When the upper and lower bounds are adjusted, the resulting MPS has infinite rays extending from the adjusted bounds. The polygon hierarchy extends in the natural way to an MPS: the MPS \( \tilde{P}^t_L[\tilde{p}_{lower_p}, \tilde{p}_{upper_p}] \) consists of the portion of hierarchy element \( \tilde{P}^t \) between (and including) \( \tilde{p}_{lower_p} \) and \( \tilde{p}_{upper_p} \). When the context is clear, we omit the range specification and hierarchy element index by referring to \( P_L[p_{lower_p}, p_{upper_p}] \) (respectively, \( \tilde{P}_L[\tilde{p}_{lower_p}, \tilde{p}_{upper_p}] \)) as \( P_L \) (respectively, \( \tilde{P}_L \)).
Figure 2.2: Monotone Polygonal Sectors.
2.3 Polygon Separation

For our polygon separation algorithms, we assume that each input convex polygon $P$ has been decomposed into its left and right MPSs $P_L$ and $P_R$. We state without proof several useful separation properties of monotone polygonal sectors [DK82]:

**MPS Property 1:** Convex polygons $P$ and $Q$ intersect iff $P_L$ intersects $Q_R$ and $P_R$ intersects $Q_L$.

**MPS Property 2:** If $P$ and $Q$ do not intersect then their separation is realized either by the separation between $P_L$ and $Q_R$ or the separation between $Q_L$ and $P_R$.

**MPS Property 3:** Ignoring degeneracies, the boundaries of two oppositely-oriented MPSs can intersect in either 0 or 2 points.

First, consider the problem of finding the separation of an $n$-vertex convex polygon $P$ and a line segment $r$. By MPS Properties 1 and 2, it is sufficient to describe an algorithm for MPS/line segment separation. For simplicity, we assume that the MPS and line segment do not intersect; the algorithm can be trivially modified to report an intersection if one is detected.

Without loss of generality, we consider the problem of determining the separation of MPS $P_L$ and line segment $r$. The following $k$ processor algorithm to find a point $p^*$ of $P$ which realizes the separation of $P_L$ and $r$ is essentially the algorithm presented by Atallah and Goodrich [AG88]: Initialize $\bar{P}_L$ to be the last element of $P_L$'s polygon hierarchy and repeat the following. If $\bar{P}_L^t$ is reduced to a constant number of segments then determine exhaustively a point $p^*$ of $\bar{P}_L^t$ (not necessarily a vertex) which realizes the separation with $r$. Otherwise, assign

---

Note that a line segment can be considered a degenerate polygon and hence these properties can be applied to polygon/line segment separation.
a processor to each vertex $\tilde{p}_i$ of $\tilde{P}_L$. In constant time, determine if vertex $\tilde{p}_i$ is closer to segment $r$ than vertices $\tilde{p}_{i-1}$ or $\tilde{p}_{i+1}$. If so, then by Observation 2, all points of $P_L$ in $P_L[\tilde{p}_{\text{lower}_P}, \tilde{p}_{i-1}]$ or in $P_L[\tilde{p}_{i+1}, \tilde{p}_{\text{upper}_P}]$ lie interior to the wedge formed by rays $\tilde{p}_i\tilde{p}_{i+1}$ and $\tilde{p}_i\tilde{p}_{i-1}$. Therefore, $p^*$ must lie in $P_L[\tilde{p}_{i-1}, \tilde{p}_{i+1}]$. Replace $\tilde{P}_L^t$ with the portion of hierarchy element $\tilde{P}_L^{t-1}$ between vertices $\tilde{p}_{i-1}$ and $\tilde{p}_{i+1}$.

Each iteration of this procedure takes a single step through the polygon hierarchy while maintaining the invariant that the remaining range of edges contains a point which realizes the desired separation. This algorithm can be used to find line/polygon separation by regarding the line as a segment with its endpoints extended to infinity. Similarly, this algorithm can be used to find point/polygon separation by regarding the point as a degenerate segment with zero length.\(^5\) This last application can be thought of as a table-lookup routine: finding the minimum in a virtual table of distances from a fixed point. We summarize these results as:

**Lemma 2.3.1:** The separation of an $n$-vertex convex polygon with a line-segment, a line or a point can be determined in $O(\log n/(1 + \log k))$ time using $k$ CREW processors.

We turn to the problem of finding the separation of arbitrary convex polygons $P$ and $Q$. If one of the polygons, say $Q$, has a constant number of facets then using Lemma 2.3.1 it is possible to determine the separation of $P$ and $Q$ by testing each facet of $Q$ against $P$. If $P$ is an $n$-vertex convex polygon, this would determine the polygon/polygon separation in $O(\log n/(1 + \log k))$ time using $k$ CREW processors. However, if both $P$ and $Q$ are $n$-vertex polygons then the naive approach of testing each facet of $Q$ against $P$ would not produce an optimal algorithm.

\(^5\) Note that due to the non-unimodality of distances in a convex polygon [E85], this technique cannot be used directly for the farthest point problem.
Therefore, consider the case in which both P and Q have n-vertices. The MPS properties simplify several aspects of determining polygon/polygon separation. By Properties 1 and 2, it suffices to compute MPS separations; without loss of generality, we describe an algorithm to determine the separation of \( P_L \) and \( Q_R \). By Property 3, if the polygons P and Q do intersect, then their corresponding (oppositely-oriented) MPS boundaries have a constant number of intersections and a witness to the intersection of P and Q can be determined from the corresponding MPS intersections.

Within each iteration of the algorithm, we maintain the invariant that the MPSs \( \tilde{P}_L[\tilde{p}_{\text{lower}, P}, \tilde{p}_{\text{upper}, P}] \) and \( \tilde{Q}_R[\tilde{q}_{\text{lower}, Q}, \tilde{q}_{\text{upper}, Q}] \) contain either a pair of points realizing the minimum distance separating \( P_L \) and \( Q_R \) or a point in the intersection of \( P_L \) and \( Q_R \). Thus, the algorithm can be viewed as a search routine to identify \( p^* \) and \( q^* \), points of \( P_L \) and \( Q_R \) respectively, which realize the MPS separation. Each iteration of the algorithm finds the separation of the current hierarchy elements \( \tilde{P}_L \) and \( \tilde{Q}_R \). If \( \tilde{P}_L \) and \( \tilde{Q}_R \) intersect then a witness to their intersection is returned as a witness to the intersection of \( P_L \) and \( Q_R \). Alternatively, if \( \tilde{P}_L \) and \( \tilde{Q}_R \) do not intersect, their separation is realized by a vertex-vertex or vertex-segment pair.

We maintain the segment T with endpoints \( \tilde{p}_i \) and \( \tilde{q}_j \) defining the separation between hierarchy elements \( \tilde{P}_L \) and \( \tilde{Q}_R \). For ease of exposition in the vertex-segment case, we assume the existence of a pseudo-vertex on the segment in question which realizes the separation and assume that it has been labeled to incorporate it into the vertex ordering. This reduces the vertex-segment case to the vertex-vertex case.

Given \( \tilde{P}_L \) and \( \tilde{Q}_R \), hierarchy element MPSs of \( P_L \) and \( Q_R \) respectively, and the points \( \tilde{p}_i \) and \( \tilde{q}_j \) which realize the separation of \( \tilde{P}_L \) and \( \tilde{Q}_R \), the following two lemmas show how the range of
vertices under consideration for realizing the separation of $P_L$ and $Q_R$ can be reduced so that a step can be taken in one of their polygon hierarchies. This is done by demonstrating that there exist portions of $P_L$ and $Q_R$ which can be replaced by horizontal rays such that points $p^*$ and $q^*$ realizing the separation are retained within the respective reduced MPSs.

**Lemma 2.3.2** (The kitty-corner lemma): Given hierarchy element MPSs $P_L$ and $Q_R$ and vertices $p_i$ and $q_j$ realizing their separation, then

(i) either $p^* \in P_L[p_{i-2}^+, p_{upper_p}]$ or $q^* \in Q_R[q_{lower_Q}, q_{j+2}^-]$

and

(ii) either $p^* \in P_L[p_{lower_P}^-, p_{i+2}^-]$ or $q^* \in Q_R[q_{j-2}^-, q_{upper_Q}].$

Furthermore, the valid reduced range in each of (i) and (ii) can be determined by a single processor in constant time.

**Proof:** If one of the MPSs, say $Q_R[q_{lower_Q}, q_{upper_Q}]$, is a single vertex\(^6\) then, by Observation 2, $p^* \in P_L[p_{i-1}^-, p_{i+1}^+]$. As well, if $p_{i-2}^-$ or $q_{j+2}$ (respectively, $p_{i+2}$ or $q_{j-2}$) do not exist then (i) (respectively, (ii)) is true trivially.

Otherwise, each of statements (i) and (ii) offers a possibility of two MPS reductions. We ignore for the moment the question of determining which reduction is applicable, and begin by showing that statements stronger than (i) and (ii) respectively are true. That is, we prove that the separation cannot be realized by a point in $P_L[p_{lower_P}^-, p_{i-1}^-]$ and a point in $Q_R[q_{j+1}^+, q_{upper_Q}]$ (respectively, a point in $P_L[p_{i+1}^+, p_{upper_p}]$ and a point in $Q_R[q_{lower_Q}^-, q_{j-1}^-]$).

Observe that the "kitty-corner" MPSs $P_L[p_{lower_P}^-, p_{i-1}^-]$ and $Q_R[q_{j+1}^+, q_{upper_Q}]$ (respectively, $P_L[p_{i+1}^+, p_{upper_p}]$ and $Q_R[q_{lower_Q}, q_{j-1}^-]$) are contained in halfplanes on opposite sides of the

---

\(^6\) That is $q_{lower_Q}^- = q_{upper_Q}^+$. 


parallel lines of support passing through \( \tilde{p}_i \) and \( \tilde{q}_j \) perpendicular to the separation segment \( \tilde{p}_i\tilde{q}_j \). It follows that their minimum separation must be greater than the distance between \( \tilde{p}_i \) and \( \tilde{q}_j \) and hence (at least) one of them can be discarded while retaining a pair of points which realize the separation of \( P_L \) and \( Q_R \).

The difficulty lies in the determination of which portions of \( P_L \) and \( Q_R \) can be discarded while retaining a pair of points which realize the true separation. To ensure that a correct reduced range can be determined by a single processor in constant time, it seems necessary to restrict consideration to which of \( P_L[\tilde{p}_{lower_P}, \tilde{p}_{i-2}] \) and \( Q_R[\tilde{q}_{j+2}, \tilde{q}_{upper_Q}] \) (respectively, which of \( P_L[\tilde{p}_{i+2}, \tilde{p}_{upper_P}] \) and \( Q_R[\tilde{q}_{lower_Q}, \tilde{q}_{j-2}] \)) can be replaced by a single ray.

We turn to the determination of a correct reduced range corresponding to statement (i). Consider the line \( \tilde{p}_{i-1}\tilde{q}_{j+1} \). At least one of (a), (b) and (c) must occur:

(a) Vertex \( \tilde{q}_{j+2} \in H^+_{\tilde{p}_{i-1}\tilde{q}_{j+1}} \) (see figure 2.3). Note that the case condition implies that \( \tilde{q}_{j+1} \in H^-_{\tilde{p}_{i-1}\tilde{q}_{j+1}} \) and equivalently, \( \tilde{p}_{i-1} \in H^+_{\tilde{q}_{j+1}\tilde{q}_{j+2}} \). As well, by the monotonicity of \( Q_R \), the y-coordinate of \( \tilde{q}_{j+1} \) is less than the y-coordinate of \( \tilde{q}_{j+2} \), therefore the case condition implies that the y-coordinate of \( \tilde{p}_{i-1} \) is less than the y-coordinate of \( \tilde{q}_{j+1} \).

Let \( w \) be the horizontal ray originating at \( \tilde{q}_{j+2} \) and extending to positive infinity. Let \( v \) be the ray originating at \( \tilde{q}_{j+2} \), at right angles to \( \tilde{q}_{j+1}\tilde{q}_{j+2} \), such that \( v \) does not intersect the interior of \( Q_R \). Denote by \( p' \) and \( q' \) points which realize the separation of \( P_L \) and \( Q_R[\tilde{q}_{j+2}, \tilde{q}_{upper_Q}] \) respectively. By Observation 1 and monotonicity, \( Q_R[\tilde{q}_{j+2}, \tilde{q}_{upper_Q}] \) is contained in the wedge \( W_Q = H^+_{\tilde{q}_{j+2}\tilde{q}_{j+2}} \cap H^-_w \) and therefore \( q' \in W_Q \). The following case analysis demonstrates that

---

7 It is possible that both (a) and (b) occur simultaneously.
wherever \( p' \) is located, there must exist a point on \( w \) which realizes the separation of \( P_L \) and \( Q_R[\tilde{q}_{j+2}, \tilde{q}_{\text{upper}_Q}] \):

1) \( p' \in H_w^+ \cup H_v^- \). Any circle centred at \( p' \) which contains a point in \( Q_R[\tilde{q}_{j+2}, \tilde{q}_{\text{upper}_Q}] \) must also contain a point of the ray \( w \). That is, if \( q' \in w \) then it is possible to find a point \( q_w \in w \) such that the angle \( \angle p'q_wq' \geq \frac{\pi}{2} \) and therefore, the distance between \( p' \) and \( q_w \) must be less than the distance between \( p' \) and \( q' \). Therefore, the separation of \( p' \) with \( Q_R[\tilde{q}_{j+2}, \tilde{q}_{\text{upper}_Q}] \) must be realized by a point on \( w \) and hence in \( Q_R[\tilde{q}_{\text{lower}_Q}, \tilde{q}_{j+2}] \).

![Figure 2.3: Kitty Corner/Same Side Lemma Cases (a) and (b).](image-url)
2) \( p' \in H_+^t \cap H_w^- \). Since the y-coordinate of \( \tilde{p}_{i-1} \) is less than the y-coordinate of \( \tilde{q}_{j+1} \) then \( \tilde{p}_{i-1} \notin H_+^v \cap H_w^- \). If \( p' \in H_+^v \), then the segment \( \tilde{p}_{i-1}p' \) must intersect \( w \) and a witness to the intersection must exist in \( QR[\tilde{q}_{\text{lower}_Q}, \tilde{q}_{j+2}] \). Otherwise, \( p' \in H_+^{-}\tilde{q}_{j+2} \). Denote by \( p'' \) the intersection of segment \( \tilde{p}_{i-1}p' \) and ray \( v \) and note that by Case 1, \( p' \neq p'' \). Assume that \( q' \notin w \). If the distance between \( p' \) and \( q' \) were less than the distance between \( p'' \) and \( \tilde{q}_{j+2} \), then ray \( p''p' \) would have to intersect ray \( \tilde{q}_{j+2}q' \). But ray \( p''p' \subset \tilde{p}_{i-1}p' \subset H_+^{-}\tilde{q}_{j+2} \) and ray \( \tilde{q}_{j+2}q' \subset W_Q \), and clearly \( H_+^{-}\tilde{q}_{j+2} \) and \( W_Q \) are disjoint except for the vertex \( \tilde{q}_{j+2} \). Therefore, the distance between \( p' \) and \( q' \) must be greater than the distance between \( p'' \) and \( \tilde{q}_{j+2} \) and the separation of \( PL \) with \( QR[\tilde{q}_{j+2}, \tilde{q}_{\text{upper}_Q}] \) must be realized by a point in \( QR[\tilde{q}_{\text{lower}_Q}, \tilde{q}_{j+2}] \).

These cases are exhaustive and therefore \( q^* \in QR[\tilde{q}_{\text{lower}_Q}, \tilde{q}_{j+2}] \). Note that the analysis does not depend on the structure of \( PL \) or the location of \( \tilde{p}_{i-1} \) other than what is implied by the initial case condition that \( \tilde{q}_{j+2} \in H_+^{+}\tilde{q}_{j+1} \). This allows us to reuse this argument in the proof of Lemma 2.3.3.

(b) Vertex \( p_{i-2} \in H_+^{+}\tilde{q}_{j+1} \). By symmetry with (a), \( p^* \in PL[\tilde{p}_{i-2}, \tilde{q}_{\text{upper}_p}] \).

c) Vertex \( \tilde{q}_{j+2} \in H_+^{-}\tilde{q}_{j+1} \) and vertex \( p_{i-2} \in H_+^{+}\tilde{q}_{j+1} \) (see figure 2.4). Denote line \( \tilde{p}_{i-1}\tilde{p}_{i-2} \) by \( L \) and line \( \tilde{q}_{j+1}\tilde{q}_{j+2} \) by \( R \). Without loss of generality, assume that the intersection of \( L \) and \( R \) lies in \( H_+^{-}\tilde{q}_{j+1} \) and therefore, that the horizontal separation of \( L \) and \( R \) in \( H_+^{-}\tilde{q}_{j+1} \) increases with increasing y-coordinate. By the case condition, \( \tilde{q}_{j+2} \in H_+^{-}\tilde{q}_{j+1} \) (respectively, \( \tilde{p}_{i-2} \in H_+^{+}\tilde{q}_{j+1} \)), the intersection of \( L \) and \( R \) must lie below \( \tilde{p}_{i-1} \) (respectively, below \( \tilde{q}_{j+1} \)).
Figure 2.4: Kitty Corner Lemma Case (c).

Define rays $w$ and $v$ as in (a). Let $u$ be the horizontal ray originating at $\tilde{p}_{i-1}$ and extending to negative infinity. Again, denote by $p'$ and $q'$ points which realize the separation of $PL$ and $QR[q_{j+2}, \tilde{q}_{j+1}]$ respectively and, as in (a), note that $q' \in W_Q = H^+_{q_{j+2}, q_{j+1}} \cap H^*_w$. Recall that the separation of $PL$ and $QR$ cannot be realized by a point in $P_L[\tilde{p}_{lower-P}, \tilde{p}_{i-1}]$ and a point
in $Q_R[\tilde{q}_{j+1}, \tilde{q}_{\text{upper}_Q}]$ and therefore we assume that $p' \in P_L[\tilde{p}_{i-1}, \tilde{p}_{\text{upper}_Q}]$. By Observation 1 and monotonicity, $P_L[\tilde{p}_{i-1}, \tilde{p}_{\text{upper}_Q}]$ is contained in the wedge $W_P = H_{\tilde{p}_{i-1}}^- \cap H_{\tilde{p}_{i-1}}^+$ and therefore $p' \in W_P$. Since the intersection of $L$ and $R$ lies below $\tilde{p}_{i-1}$ and $\tilde{q}_{j+1}$, the wedges $W_P$ and $W_Q$ are disjoint.

Let $\alpha = \angle \tilde{q}_{j+1}\tilde{q}_{j+2}\tilde{p}_{i-1}$ and $\beta = \angle \tilde{q}_j\tilde{q}_{j+1}\tilde{p}_{i-1}$. Before proceeding with a case analysis, we show that $\alpha < \frac{\pi}{2}$. Since $\tilde{q}_{j+2} \in H_{\tilde{q}_j \tilde{q}_{j+1}}^+$ (by Observation 1) and $\tilde{q}_{j+2} \in H_{\tilde{p}_{i-1} \tilde{q}_{j+1}}^-$ (by the case condition), therefore $\alpha < \beta$. We assume that $\alpha \geq \frac{\pi}{2}$ (and, therefore $\beta > \frac{\pi}{2}$) and derive a contradiction. If $\beta > \frac{\pi}{2}$, then $\tilde{p}_1 \in H_{\tilde{p}_{i-1} \tilde{q}_{j+1}}^+$ (since $\angle \tilde{p}_1 \tilde{q}_j \tilde{q}_{j+1} \geq \frac{\pi}{2}$). Therefore, $\tilde{p}_{i-2}$ must be below the horizontal line through $\tilde{p}_{i-1}$ (by monotonicity), $\tilde{p}_{i-2} \in H_{\tilde{p}_{i-1} \tilde{q}_{j+1}}^+$ (by Observation 1), and $\tilde{p}_{i-2} \in H_{\tilde{p}_{i-1} \tilde{q}_{j+1}}^{-}$ (by the case condition). But these three regions are disjoint (except for vertex $\tilde{p}_{i-1}$). Therefore, $\beta \leq \frac{\pi}{2}$ and hence $\alpha < \frac{\pi}{2}$ and $\tilde{p}_{i-1} \in H_{\tilde{v}}^-$.

The following case analysis demonstrates that wherever $p'$ is located there must exist a point on $w$ which realizes the separation of $P_L$ and $Q_R[\tilde{q}_{j+2}, \tilde{q}_{\text{upper}_Q}]$:

1) $p' \in H_{\tilde{q}_j \tilde{q}_{j+2}}^+ \cap H_{w}^- \ (= W_Q)$. As noted above, $p' \in W_P$ and $W_P \cap W_Q = \emptyset$.

Therefore this case cannot occur.

2) $p' \in H_{w}^+ \cup H_{\tilde{v}}^-$. As in (a) Case 1, any circle centred at $p'$ which contains a point in $Q_R[\tilde{q}_{j+2}, \tilde{q}_{\text{upper}_Q}]$ must also contain a point of the ray $w$. Therefore, the separation of $p'$ with $Q_R[\tilde{q}_{j+2}, \tilde{q}_{\text{upper}_Q}]$ must be realized by a point in $Q_R[\tilde{q}_{\text{lower}_Q}, \tilde{q}_{j+2}]$.

3) $p' \in H_{\tilde{v}}^+ \cap H_{\tilde{q}_{j+2} \tilde{q}_{j+2}}^-$. Since $\tilde{p}_{i-1} \in H_{\tilde{v}}^-$, $p'$ and $\tilde{p}_{i-1}$ lie on opposite sides of ray $v$.

Denote by $p''$ the intersection of segment $\tilde{p}_{i-1}p'$ and ray $v$ and note that by Case 1, $p' \neq p''$. Assume that $q' \in w$. If the distance between $p'$ and $q'$ were less than the distance between $p''$ and $\tilde{q}_{j+2}$, then ray $p''p'$ would have to intersect ray $\tilde{q}_{j+2}q'$. But ray $p''p' \subset \text{ray } \tilde{p}_{i-1}p' \subset W_P$ and ray $\tilde{q}_{j+2}q' \subset W_Q$, and $W_P \cap W_Q = \emptyset$. Therefore,
the distance between \( p' \) and \( q' \) must be greater than the distance between \( p'' \) and \( \tilde{q}_{j+2} \)
and the separation of \( PL \) with \( QR[\tilde{q}_{j+2}, \tilde{q}_{upper_Q}] \) must be realized by a point in
\( QR[\tilde{q}_{lower_Q}, \tilde{q}_{j+2}] \).

These cases are exhaustive and therefore \( q^* \in QR[\tilde{q}_{lower_Q}, \tilde{q}_{j+2}] \). Similarly, if the
intersection of \( L \) and \( R \) lies in \( H_{\tilde{p}\tilde{q}} \) then \( p^* \in PL[\tilde{p}_{i-2}, \tilde{p}_{upper_p}] \). If \( L \) and \( R \) are parallel, then
\( q^* \in QR[\tilde{q}_{lower_Q}, \tilde{q}_{j+2}] \) and \( p^* \in PL[\tilde{p}_{i-2}, \tilde{p}_{upper_p}] \).

The arguments used in (a), (b) and (c) prove (i); using symmetry in the y direction the
same arguments prove (ii). The tests employed above to determine correct reduced ranges are
line intersection and point containments in half-planes. These tests can be easily implemented to
run in constant time using a single processor.

To take a step in one of the polygon hierarchies within a given iteration, it is necessary to
ensure that one of the \( PL \) and \( QR \) subsequences under consideration be reduced to a constant
length. This is not guaranteed by Lemma 2.3.2. It is possible, for example, that only the
upper portion of \( PL \) and the upper portion of \( QR \) are eliminated. The following Lemma shows
that this reduction is always possible for at least one of the hierarchy element approximations.

**Lemma 2.3.3** (The same side lemma): Given hierarchy element MPSs \( PL \) and \( QR \) and
vertices \( \tilde{p}_i \) and \( \tilde{q}_j \) realizing their separation, then

(i) either \( p^* \in PL[\tilde{p}_{lower_p}, \tilde{p}_{i+2}] \) or \( q^* \in QR[\tilde{q}_{lower_Q}, \tilde{q}_{j+2}] \)

and (ii) either \( p^* \in PL[\tilde{p}_{i-2}, \tilde{p}_{upper_p}] \) or \( q^* \in QR[\tilde{q}_{j-2}, \tilde{q}_{upper_Q}] \).

Furthermore, the valid reduced range in each of (i) and (ii) can be determined by a single
processor in constant time.
Proof: If one of the MPSs, say $Q_R[\tilde{q}_{lower_Q}, \tilde{q}_{upper_Q}]$, is a single vertex then, by Observation 2, $p^* \in P_L[\tilde{p}_{i-1}, \tilde{p}_{i+1}]$. As well, if $\tilde{p}_{i+2}$ or $\tilde{q}_{j+2}$ (respectively, $\tilde{p}_{i-2}$ or $\tilde{q}_{j-2}$) do not exist then (i) (respectively, (ii)) is true trivially.

Otherwise, as in Lemma 2.3.2, we ignore for the moment the question of determining which reduction is applicable, and begin by showing that statements stronger than (i) and (ii) respectively are true. Observe that the "same side" MPSs $P_L[\tilde{p}_{i-1}, \tilde{p}_{i}+1]$ and $Q_R[\tilde{q}_{j+1}, \tilde{q}_{upper_Q}]$ (respectively, $P_L[\tilde{p}_{lower_P}, \tilde{p}_{i-1}]$ and $Q_R[\tilde{q}_{lower_Q}, \tilde{q}_{j-1}]$) are contained in halfplanes on opposite sides of the parallel lines of support passing through $\tilde{p}_i$ and $\tilde{q}_j$ perpendicular to the separation segment $\tilde{p}_i\tilde{q}_j$. It follows that their minimum separation must be greater than or equal to the distance between $\tilde{p}_i$ and $\tilde{q}_j$ and that (at least) one of them can be discarded while retaining a pair of points which realize the separation of $P_L$ and $Q_R$.

Again, it seems necessary to restrict consideration to which of $P_L[\tilde{p}_{lower_P}, \tilde{p}_{i-2}]$ and $Q_R[\tilde{q}_{lower_Q}, \tilde{q}_{j-2}]$ (respectively, which of $P_L[\tilde{p}_{i+2}, \tilde{p}_{upper_P}]$ and $Q_R[\tilde{q}_{j+2}, \tilde{q}_{upper_Q}]$) can be discarded to ensure that a correct reduced range can be determined by a single processor in constant time.

We turn to the determination of a correct reduced range corresponding to statement (i). Consider line $\tilde{p}_{i+1}\tilde{q}_{j+1}$. At least one of (a), (b) and (c) must occur:

(a) Vertex $\tilde{q}_{j+2} \in H^+_{\tilde{p}_{i+1}\tilde{q}_{i}}$. Substituting $\tilde{p}_{i+1}$ for $\tilde{p}_{i-1}$, the same case analysis used in Lemma 2.3.2 (a) shows that $q^* \in Q_R[\tilde{q}_{lower_Q}, \tilde{q}_{j+2}]$.

(b) Vertex $\tilde{p}_{i+2} \in H^+_{\tilde{p}_{i+1}\tilde{q}_{i}}$. By symmetry with (a), $p^* \in P_L[\tilde{p}_{lower_P}, \tilde{p}_{i+2}]$.

---

8 It is possible that both (a) and (b) occur simultaneously.
(c) Vertex \( \bar{q}_{j+2} \in H_{p_{i+1}q_{j+1}}^- \) and vertex \( \bar{p}_{i+2} \in H_{p_{i+1}q_{j+1}}^- \) (see figure 2.5). Define rays \( w \) and \( v \) as in Lemma 2.3.2 (a).

Consider the angles \( \alpha = \angle \bar{p}_{i+1}\bar{p}_{i+2}\bar{q}_{j+2} \) and \( \beta = \angle \bar{q}_{j+1}\bar{q}_{j+2}\bar{p}_{i+2} \). Note that \( \alpha + \beta < \pi \) (otherwise, rays \( \bar{p}_{i+1}\bar{p}_{i+2} \) and \( \bar{q}_{j+1}\bar{q}_{j+2} \) would intersect or would be parallel) and therefore at least one of the angles \( \alpha \) or \( \beta \) must be less than \( \frac{\pi}{2} \). Without loss of generality, assume that \( \beta < \frac{\pi}{2} \) and therefore that \( \bar{p}_{i+2} \in H_v^- \).

Again, denote by \( p' \) and \( q' \) points which realize the separation of \( P_L \) and \( Q_R[\bar{q}_{j+2}, \bar{q}_{\text{upper}_Q}] \) respectively and, as in Lemma 2.3.2 (a), note that \( q' \in W_Q = H_{q_{i+2}q_{j+1}}^- \cap H_w^- \). We demonstrate by a case analysis that wherever \( p' \) is located there must exist a point on \( w \) which realizes the separation of \( P_L \) and \( Q_R[\bar{q}_{j+2}, \bar{q}_{\text{upper}_Q}] \):

1) \( p' \in H_{q_{i+2}q_{j+2}}^- \cap H_w^- \) (= \( W_Q \)). In the halfplane \( H_{p_{i+1}q_{j+1}}^- \), \( P_L \subseteq H_{p_{i+1}q_{j+1}}^- \) (by Observation 1), \( W_Q \subseteq H_{q_{i+2}q_{j+1}}^+ \) (by Observation 1), and rays \( \bar{q}_{j+1}\bar{q}_{j+2} \) and \( \bar{q}_{j+1}\bar{q}_{j+2} \) do not intersect. In the halfplane \( H_{p_{i+1}q_{j+1}}^+ \), \( P_L \subseteq H_{p_{i+1}q_{j+1}}^+ \) (by Observation 1), \( W_Q \subseteq H_w^- \) and rays \( \bar{p}_{i+2}\bar{p}_{i+1} \) and \( w \) do not intersect. Therefore, \( P_L \) and \( W_Q \) are disjoint and this case cannot occur.

2) \( p' \in H_w^+ \cup H_v^- \). As in Lemma 2.3.2 (a) Case 1, any circle centred at \( p' \) which contains a point in \( Q_R[\bar{q}_{j+2}, \bar{q}_{\text{upper}_Q}] \) must also contain a point of the ray \( w \). Therefore, the separation of \( p' \) with \( Q_R[\bar{q}_{j+2}, \bar{q}_{\text{upper}_Q}] \) must be realized by a point in \( Q_R[\bar{q}_{\text{lower}_Q}, \bar{q}_{j+2}] \).

3) \( p' \in H_w^+ \cap H_{q_{i+2}q_{j+2}}^- \). Since \( \beta < \frac{\pi}{2} \), then \( \bar{p}_{i+2} \in H_v^- \) and \( p' \in P_L[\bar{p}_{i+2}, \bar{q}_{\text{upper}_P}] \). But, as noted above, the separation of \( P_L \) and \( Q_R \) cannot be realized by a point in \( P_L[\bar{p}_{i+1}, \bar{q}_{\text{upper}_P}] \) and a point in \( Q_R[\bar{q}_{j+1}, \bar{q}_{\text{upper}_Q}] \). Therefore, no \( q' \) arising in this case can be a point realizing the minimum separation of \( P_L \) and \( Q_R \).
Figure 2.5: Same Side Lemma Case (c).

These cases are exhaustive and therefore \( q^* \in Q_R[\tilde{q}_{lower_Q}, \tilde{q}_{j+2}] \). Similarly, if \( \alpha < \frac{\pi}{2} \) then \( p^* \in P_L[\tilde{p}_{lower_P}, \tilde{p}_{i+2}] \).

The arguments used in (a), (b) and (c) prove (i); using symmetry in the y direction the same arguments prove (ii). The tests employed above to determine the valid reduced ranges are point containments in half-planes and angle comparisons. These tests can be implemented to run in constant time using a single processor.
Without loss of generality, assume that \( k \geq 25 \). We outline a \( k \) CREW processor algorithm for determining the separation between \( P_L \) and \( Q_R \) which follows from Lemmas 2.3.2 and 2.3.3 above: Initialize \( \tilde{P}_L \) and \( \tilde{Q}_R \) to be the last elements of the \( \sqrt{k} \)-polygon hierarchies\(^9\) of \( P_L \) and \( Q_R \) respectively and repeat the following. If \( \tilde{P}_L^t \) or \( \tilde{Q}_R^s \) are of constant size then use the polygon/segment separation algorithm to determine their separation and halt. Otherwise, assign a processor to each pairing of a vertex from \( \tilde{P}_L^t \) with a vertex from \( \tilde{Q}_R^s \). The processor which determines that the edges incident on its pair of vertices realize the separation of the current hierarchy elements then restricts the sequence of vertices under consideration according to Lemmas 2.3.2 and 2.3.3 by adjusting lower_\( P \) and upper_\( P \), or lower_\( Q \) and upper_\( Q \). Depending on which MPS bounds are adjusted, replace \( \tilde{P}_L^t \) (respectively, \( \tilde{Q}_R^s \)) with the portion of \( \tilde{P}_L^{t-1} \) (respectively, \( \tilde{Q}_R^{s-1} \)) between vertices \( \tilde{p}_{\text{lower}_P} \) and \( \tilde{p}_{\text{upper}_P} \) (respectively, \( \tilde{q}_{\text{lower}_Q} \) and \( \tilde{q}_{\text{upper}_Q} \)).

**Theorem 2.3.4:** The separation of two convex n-vertex polygons \( P \) and \( Q \) can be determined in \( O(\log n/(1 + \log k)) \) time using \( k \) CREW processors.

**Proof:** Given two convex n-vertex polygons, the MPS preprocessing is performed in \( O(\log n/(1 + \log k)) \) time using \( k \) CREW processors. Within each iteration, the vertices realizing the separation of \( \tilde{P}_L \) and \( \tilde{Q}_R \) (or \( \tilde{P}_R \) and \( \tilde{Q}_L \)) can be determined in constant time. Therefore by Lemmas 2.3.2 and 2.3.3, each iteration takes constant time and results in a step in (at least) one of the polygon hierarchies. Therefore, a total of \( O(\log n/(1 + \log k)) \) iterations

\(^9\) Note that an integer within a constant factor of \( \sqrt{k} \) will suffice for the subdivision size and that with \( k \) processors an integral approximation of \( \sqrt{k} \) can be computed in constant time. Each processor squares its predecessor's, its successor's and its own index. The unique processor whose squared index is closest to \( k \), writes its index into a predetermined location and the other processors read it using the concurrent read facility.
will suffice to determine MPS separation. Running the algorithm twice for the two MPS pairs will suffice to determine the convex polygon separation.

The use of the $\sqrt{k}$-polygon hierarchy (an instance of the $\sqrt{k}$-subdivision technique [KR88] [G87a]) provides a logarithmic sequential algorithm and a constant time algorithm using a quasi-linear number of processors for computing convex polygon separation. The polygon/polygon separation case analysis provided in Edelsbrunner [E85] produces a sequential $O(\log n)$ time separation algorithm. If concurrent write is available, it is possible to use Edelsbrunner's case analysis to design a polygon/polygon separation algorithm which runs in $O(\log n/(1 + \log k))$ time using $k$ CRCW processors. In the absence of a concurrent write facility, it is not straightforward to coordinate the $\sqrt{k}$ case results at each vertex of the hierarchy elements under consideration. Our algorithm avoids this coordination problem by "electing" the processor which corresponds to the hierarchy element minimum separation to apply the case analysis.
2.4 Polygon Separating and Common Tangents

For the problem of computing convex polygon tangents, we add a preprocessing step. To ensure that the polygons in question do not intersect, we apply the appropriate separation algorithm presented in the last section. With a pair of points realizing the separation, we construct a separating line $S$ through the midpoint of and perpendicular to the segment realizing the separation. Without loss of generality, we assume that this separating line $S$ is oriented horizontally with polygon $P$ above and polygon $Q$ below. Given this orientation, it is sufficient to consider MPS common/separating tangents: The two separating tangents of $P$ and $Q$ can be constructed from oppositely-oriented MPSs $P_L$ and $Q_R$, or $P_R$ and $Q_L$. The two common tangents can be constructed from similarly-oriented MPSs $P_R$ and $Q_R$ or $P_L$ and $Q_L$.

Therefore, we assume that each polygon $P$ has been preprocessed into MPSs $P_L$ and $P_R$ which are monotone with respect to the perpendicular of the separating line $S$ as constructed above. Note that the rays augmenting the left and right monotone polygonal chains of $P$ are parallel to $S$.

We address the problem of finding separating tangents. For a convex polygon and vertex pair, the separating tangents are the same as the common tangents. For two convex polygons, minor modifications to the separating tangents algorithm produce an algorithm for finding common tangents; we discuss the necessary modifications at the end of the section.

We first consider the problem of constructing the tangent of an $n$-vertex MPS $P_L$ passing through a vertex $r$ outside $P_L$. We assume that $P_L$ and $r$ are presented with a horizontally-oriented separating line $S$ with $P_L$ above $S$ and $r$ below $S$. The following $k$ processor algorithm to find a point $p^*$ of $P$ which realizes the tangent of $P_L$ passing through $r$ is
essentially the algorithm presented by Atallah and Goodrich [AG88]: Initialize $\tilde{P}_L$ to be the last element of $P_L$'s polygon hierarchy and repeat the following. If $\tilde{P}_L^t$ is reduced to a constant number of segments then determine exhaustively a vertex $p^*$ of $\tilde{P}_L^t$ which realizes the tangent passing through $r$. Otherwise, assign a processor to each vertex $\tilde{p}_i$ of $\tilde{P}_L^t$. In constant time, for vertex $\tilde{p}_i$ determine if $\tilde{p}_{i-1}$ and $\tilde{p}_{i+1}$, the neighbours of $\tilde{p}_i$ on $\tilde{P}_L^t$, both lie on the same side of line $r\tilde{p}_i$. If so, then by Observation 2, all points of $P_L$ in $P_L[\tilde{P}_{\text{lower}_P}, \tilde{p}_{i-1}]$ or in $P_L[\tilde{p}_{i+1}, \tilde{P}_{\text{upper}_P}]$ lie interior to the wedge formed by rays $\tilde{p}_i\tilde{p}_{i+1}$ and $\tilde{p}_{i+1}\tilde{p}_i$. Therefore, $p^*$ must lie in $P_L[\tilde{p}_{i-1}, \tilde{p}_{i+1}]$. Set $\tilde{P}_{\text{lower}_P}$ to $\tilde{p}_{i+1}$ and $\tilde{P}_{\text{upper}_P}$ to $\tilde{p}_{i-1}$ and replace $\tilde{P}_L^t$ with the portion of hierarchy element $\tilde{P}_L^{t-1}$ between vertices $\tilde{p}_{i-1}$ and $\tilde{p}_{i+1}$.

Each iteration of this procedure takes a single step through the polygon hierarchy while maintaining the invariant that the range of edges under consideration contains a point which realizes the desired tangent. Therefore, the tangent of $P_L$ passing through $r$ can be constructed using $k$ CREW processors in $O(\log n/(1 + \log k))$ time. The separation and MPS preprocessing are performed within the same resource bounds.

Next we consider the problem of constructing the separating tangents of $n$-vertex convex polygons $P$ and $Q$. As noted above, it suffices to construct the separating tangents of oppositely-oriented MPSs; without loss of generality, we describe an algorithm to construct the separating tangent of $P_L$ and $Q_R$. We assume that $P_L$ and $Q_R$ have been preprocessed with respect to the horizontally-oriented separating line $S$ as described above and that they are presented with $P_L$ above $S$ and $Q_R$ below $S$. Since this preprocessing step uses the separating line perpendicular to the minimum separation of $P$ and $Q$, the vertex with minimum $y$-coordinate on $P_L$ and the vertex with the maximum $y$-coordinate on $Q_R$ have the same $x$-coordinate. Therefore, we are searching for the separating tangent of maximum positive slope.
Denote the vertices of $P_L$ and $Q_R$ which realize the separating tangent as $p^*$ and $q^*$ respectively; recall that the vertices are indexed in order of increasing $y$-coordinate. Given $\tilde{P}_L$ and $\tilde{Q}_R$, hierarchy element MPSs of $P_L$ and $Q_R$ respectively, and the current separating tangent line $T$ defined by vertices $\tilde{p}_i$ and $\tilde{q}_j$ of $\tilde{P}_L$ and $\tilde{Q}_R$ respectively, the following two lemmas show how the range of vertices under consideration for realizing the separating tangent of $P_L$ and $Q_R$ can be reduced so that a step can be taken in one of their polygon hierarchies. This is done by demonstrating that there exist portions of $\tilde{P}_L$ and $\tilde{Q}_R$ which can be replaced by horizontal rays such that points $p^*$ and $q^*$ realizing the separating tangent are retained within the respective reduced MPSs.

**Lemma 2.4.1** (see figure 2.6): Given hierarchy element MPSs $\tilde{P}_L$ and $\tilde{Q}_R$ with vertices $\tilde{p}_i$ and $\tilde{q}_j$ realizing their separating tangent $T$ then $p^* \in P_L[\tilde{p}_{lower_p}, \tilde{p}_{i+1}]$ and $q^* \in Q_R[\tilde{q}_{j-1}, \tilde{q}_{upper_Q}]$.

**Proof:** If one of the MPSs, say $Q_R[\tilde{q}_{lower_Q}, \tilde{q}_{upper_Q}]$, is a single vertex then, by Observation 2, $p^* \in P_L[\tilde{p}_{i-1}, \tilde{p}_{i+1}]$.

Otherwise, $p^* \in P_L[\tilde{p}_{i-1}, \tilde{p}_{upper_p}]$ iff a vertex in that range lies in $H^+_{\tilde{q}_{j-1}}$. By Observation 1, all vertices in $P_L[\tilde{p}_{i+1}, \tilde{p}_{upper_p}]$ lie in $H^-_{\tilde{q}_{j-1}}$ and hence in $H^-_{\tilde{q}_{j-1}}$. Therefore, $p^* \in P_L[\tilde{p}_{lower_p}, \tilde{p}_{i+1}]$ and, by a symmetric argument, $q^* \in Q_R[\tilde{q}_{j-1}, \tilde{q}_{upper_Q}]$. 

**Lemma 2.4.2** (see figure 2.6): Given hierarchy element MPSs $\tilde{P}_L$ and $\tilde{Q}_R$ with vertices $\tilde{p}_i$ and $\tilde{q}_j$ realizing their separating tangent $T$ then either $p^* \in P_L[\tilde{p}_{i-2}, \tilde{p}_{i+1}]$ or $q^* \in Q_R[\tilde{q}_{j-1}, \tilde{q}_{j+2}]$. Furthermore, the correct containment can be determined in constant time using a single processor.
Proof: If one of the MPSs, say $Q_R[\tilde{q}_{\text{lower}_Q}, \tilde{q}_{\text{upper}_Q}]$, is a single vertex then, by Observation 2, $p^* \in P_L[\tilde{p}_{i-1}, \tilde{p}_{i+1}]$. As well, if $\tilde{p}_{i-2}$ or $\tilde{q}_{j+2}$ do not exist then the result follows trivially from Lemma 2.4.1.

Otherwise, denote line $\tilde{p}_{i-2}\tilde{p}_{i-1}$ by L, and line $\tilde{q}_{j+1}\tilde{q}_{j+2}$ by R. Note that the slopes of L and R are both positive and both less than the slope of T. Without loss of generality, assume that L and R intersect below S and denote line $\tilde{p}_{i}\tilde{p}_{i-1}$ by $L_2$. Note that $L_2$ must also intersect R below S since the slope of $L_2$ lies between the slopes of L and T. If $q^* \in Q_R[\tilde{q}_{j+2}, \tilde{q}_{\text{upper}_Q}]$ then part of $P_L$ above S must extend below R. By Observation 1, all vertices on $P_L$ which are outside $P_L[\tilde{p}_{i-2}, \tilde{p}_{i-1}]$ (respectively, outside $P_L[\tilde{p}_{i-1}, \tilde{p}_{i}]$) are above L (respectively, above $L_2$). Therefore, no part of $P_L$ below T can extend below R and therefore, with the above argument, $q^* \in Q_R[\tilde{q}_{j-1}, \tilde{q}_{j+2}]^1$. Similarly, if L intersects R above S then $p^* \in P_L[\tilde{p}_{i-2}, \tilde{p}_{i+1}]$.

This test can be easily implemented to run in constant time using a single processor. §

The polygon/polygon separating tangent algorithm uses the $\sqrt{k}$-subdivision technique (in the $\sqrt{k}$-polygon hierarchy) in the same way as the polygon separation algorithm described in section 2.3. Within each iteration the separating tangent between the current hierarchy elements is determined and the case analysis described in the above lemmas enables the algorithm to take a step in (at least) one of the polygon hierarchies.

---

1 Note that if line $q_{i}q_{j+1}$ intersects L above S then it is possible that $q^* \in [\tilde{q}_{j+1}, \tilde{q}_{j+2}]$. 
Theorem 2.4.3: The separating tangents of two convex $n$-vertex polygons $P$ and $Q$ can be determined in $O(\log n/(1 + \log k))$ time using $k$ CREW processors.

Proof: The separation preprocessing and decomposition into monotone polygonal sectors is performed within the same resource bounds. Similar to Theorem 2.3.4, by Lemmas 2.4.1 and
2.4.2, each iteration takes constant time and results in a step through one of the polygon hierarchies. Therefore, a total of $O(\log n/(1 + \log k))$ iterations will suffice to determine MPS separating tangents. Running the algorithm twice with the appropriate (symmetric) changes will produce both separating tangents.

The case analysis presented here is similar in spirit to that provided by Overmars and Van Leeuwen [OvL81]. They describe an analysis for a sequential $O(\log n)$ time algorithm to find the upper common tangents of two $n$-vertex convex polygons; their analysis can be modified to provide an equivalent separating tangent result. Although the analysis for their common tangent algorithm is sufficient for the sequential result, it only considers consecutive vertices and thus does not immediately provide a test which will enable the algorithm to take a step in one of the polygon hierarchies.

The separating tangents algorithm can be modified easily to compute common tangents within the same bounds. For the separating tangent case analysis presented in Lemma 2.4.2, the case reduction is predicated on whether $R$ and $L$ intersect above or below the separating line $S$. In fact, within Lemma 2.4.2, the same case reduction could have been predicated on whether $R$ and $L$ intersect to the left or to the right of the current separating tangent $T$. However, the analog of Lemma 2.4.2 used in the common tangent algorithm requires a separating line for its case reduction (as does the case reduction of Overmars and Van Leeuwen [OvL81]). Therefore, as for the separating tangents algorithm, given two convex polygons $P$ and $Q$, we apply the appropriate separation algorithm, construct a line $S$ between them perpendicular to their separation and decompose $P$ and $Q$ into their respective MPSs with respect to line $S$. 
We describe the modifications to find the common tangent of $P_L$ and $Q_L$ (see figure 2.7); the common tangent algorithm for $P_R$ and $Q_R$ is symmetric. The $\sqrt{k}$-hierarchies technique is used for each of $P_L$ and $Q_L$. Within each iteration, the tangent of $\tilde{P}_L$ and $\tilde{Q}_L$ is found and analyses analogous to those in Lemmas 2.4.1 and 2.4.2 are applied to take a step in one of $\tilde{P}_L$'s or $\tilde{Q}_L$'s polygon hierarchies.

Denote the vertices of $P_L$ and $Q_L$ which realize the common tangent as $p^*$ and $q^*$ respectively. Assume that we have vertices $p_i$ and $q_j$ which realize the common tangent $T$ of $\tilde{P}_L[p_{\text{lower}_P}, p_{\text{upper}_P}]$ and $\tilde{Q}_L[q_{\text{lower}_Q}, q_{\text{upper}_Q}]$, hierarchy elements of $P_L$ and $Q_L$ respectively.

For the analog of Lemma 2.4.1, consider the horizontal slab delimited by the y-coordinate of $p_i$ above and the y-coordinate of $q_j$ below. The true common tangent support vertices $p^*$ and $q^*$ must lie on or to the right of where the current common tangent segment $p_iq_j$ intersects this slab. Note that $\tilde{p}_{i-1}$ and $\tilde{q}_{j+1}$ lie to the left of the current common tangent. By Observation 1, all vertices on $P_L$ below $p_{i-1}$ lie to the left of line $\tilde{p}_{i-1}\tilde{p}_i$ and therefore lie to the left of the current common tangent. Therefore, $p^*$ cannot be below $\tilde{p}_{i-1}$ on $P_L$. Similarly, $q^*$ cannot be above $\tilde{q}_{j+1}$ on $Q_L$. 
Figure 2.7: Common tangents.

For the analog of Lemma 2.4.2, denote ray $\tilde{p}_{i+2}\tilde{p}_{i+1}$ by $L$, and ray $\tilde{q}_{j-2}\tilde{q}_{j-1}$ by $R$. Denote ray $\tilde{p}_{i+1}\tilde{p}_i$ by $L_2$ and ray $\tilde{q}_{j-1}\tilde{q}_j$ by $R_2$. Without loss of generality, assume that $L$ and $R$ intersect below the given separator $S$. Note that $L_2$ must also intersect $R$ below $S$ since $L_2$ lies
to the left of L. For q* to exist below qj on Q, part of PL must extend to the right of R. By Observation 1, all vertices on PL which are outside PL[πi+2, πi+1] (respectively, outside PL[πi+1, πi]) are to the left of L (respectively, to the left of L2) and above S. Therefore, no part of P can extend to the right of R and therefore, combined with the above reduction q* ∈ Q[qi-2, qi+1].2 Similarly, if L intersects R above the given separator S then p* ∈ PL[πi-1, πi+2].

Therefore, using the √k-polygon hierarchies and the case analysis above, we have immediately:

**Theorem 2.4.4:** The common tangents of two convex n-vertex polygons P and Q can be determined in O(log n/(1 + log k)) time using k CREW processors.

This technique provides a logarithmic sequential algorithm and a constant time algorithm using a quasi-linear number of processors for computing upper common tangents. As noted by Atallah and Goodrich [AG88], with n processors using standard recursive subdivision, the common tangents algorithm can be incorporated into another optimal 2-d convex hull algorithm (cf. section 2.6). Earlier optimal 2-d convex hull algorithms used a √k-subdivision for the divide and conquer step [ACGOY88] [AG86a]; this can be thought of as using the power of this technique within a different level of the algorithm. This √k-subdivision technique has also been used for list merging and other applications [KR88]. Interestingly, Cole & Goodrich [CG88] use a cascading routine based on Cole's parallel merge sort [C86b] to construct the convex hull without using the √k-subdivision technique.

---

2 Note that if R2 intersects L and L2 above S then it is possible that q* ∈ [qi-1, qi-2].
2.5 A Sorted Point Set Convex Hull Algorithm

In this section, we address some of the implementation considerations associated with the convex polygon bridge finding algorithm presented in section 2.4. To motivate this discussion, we describe a parallel algorithm which constructs the convex hull of a sorted point set of size $n$ in $O(\log n)$ parallel time using $n/\log n$ CREW processors. This problem was considered by Goodrich [G87a] [G87b] who also presents an $O(\log n)$ time algorithm which employs $n/\log n$ CREW processors\(^3\). The algorithm presented here is based on a novel scheduling technique [KP90] and the parallel bridge finding algorithm described in section 2.4.

We assume that the $n$ input vertices are sorted by $x$-coordinate in non-decreasing order and that we are constructing the upper convex hull. We superimpose a complete binary tree on the point set and associate with each internal node the convex hull of its descendants. The levels of the tree are numbered from 0 (the leaves) to $\log n$ (the root). The final goal of the algorithm is to construct the convex hull at the root (the entire point set) by constructing the hulls associated with various internal nodes. The algorithm proceeds in three phases:

(i) Level 0 to level $(\log \log n)$: The sorted point set is split into $n/\log n$ contiguous blocks of size $\log n$. A single processor is assigned to each block and constructs the convex hull of its block using a linear time sequential algorithm [Me87]. At the end of phase (i), there are $n/\log n$ hulls of size $O(\log n)$, therefore for the next phase we assign one processor per hull.

(ii) Level $(1 + \log \log n)$ to level $(2 \log \log n)$: The hull associated with each internal node is constructed from the hulls associated with its two children by using a logarithmic time sequential bridge-finding algorithm (cf. section 2.4, [OvL81] and [DK90]). The tree is

---

\(^3\) Apparently, Wagener [Wa85] has derived similar results which are unpublished.
computed in parallel one level at a time. At the end of phase (ii), there are $n/\log^2 n$ hulls of size $O(\log^2 n)$ each, therefore for the next phase we assign $\log n$ processors per hull.

(iii) Level $(1 + 2\log \log n)$ to level $(\log n)$: Again, the hull associated with each internal node is constructed from the hulls associated with its two children, this time using the cooperative bridge-finding algorithm of section 2.4.

We examine the time requirements of the algorithm. In phase (i), each processor uses a sequential algorithm to construct the hull of $\log n$ vertices in $O(\log n)$ time. Within each level of phase (ii), each processor uses a 'binary search' type bridge finding routine to find their combined hulls in time logarithmic in their size. Therefore, constructing the hulls on any single level in phase (ii) can be done in $O(\log \log n)$ time. Since there are $O(\log \log n)$ levels in phase (ii), phase (ii) can be completed in $O((\log \log n)^2)$ time. In phase (iii), the number of processors available for each bridge-finding routine, is quasi-linear ($n^\alpha$ for some $\alpha > 0$) in the size of the hulls to be combined. Since there are fewer than $\log n$ levels in phase (iii) and the hulls on each level can be constructed in constant time, phase (iii) can be completed in $O(\log n)$ time. Therefore the three phases can be completed in $O(\log n)$ time.

We have shown that the computation involved with constructing the hull of a sorted point set of size $n$ can be performed in $O(\log n)$ time. However, we must show that all the data structure manipulations necessary to support this computation can also be performed within this time bound. Specifically, the bridge-finding algorithms used in phases (ii) and (iii) assume that the points are presented in contiguous array locations, therefore our data structure must support random access. However, as the bridges are constructed and sequences of vertices are removed, we may not have sufficiently many processors to perform a full compaction (so that
each hull is represented by contiguous locations in a single array) within the required time bounds.

We solve this problem by employing a two-level array structure. This structure does not simulate a single array and therefore does not allow full random access. However it enables us to perform our binary (and k-ary) searches in two stages and, by incorporating pointers, it allows us to perform the necessary compaction within the required time-processor bounds.

We describe the data structure and its usage in more detail. At the end of phase (i), there are \( \frac{n}{\log n} \) hulls of size \( O(\log n) \) each. Each of these is placed into a hull array, an array of length \( \log n \). The header array, an array of \( \frac{n}{\log n} \) entries, is initialized so that each entry is associated with a hull array. Each header array entry consists of a single representative vertex (say, the hull vertex with smallest x-coordinate) from its hull array, and two pointers into its hull array, to the beginning and end of the hull in the array respectively. Throughout the algorithm, we maintain the invariant that each hull is specified by a consecutive sequence of header array entries, each of which points to a non-empty hull array. Our invariant is initialized with the header array at the end of phase (i). We assign one processor to each header array entry which (irrespective of its other duties) is always responsible for moving the location of its entry in a compaction step to maintain our invariant.

In constructing the convex union of two hulls using a bridge-finding routine, contiguous portions of each hull are discarded, leaving contiguous portions of each hull to be combined with a bridge. After the bridge for two hulls is constructed, the header array entries which point to 'surviving' portions of the hull are moved to consecutive array locations within the header array. Using the compaction processor assignment mentioned above and employing concurrent read, this compaction can be performed in constant time maintaining our invariant.
In performing the bridge-finding, the representative vertices associated with the entries in the header array are used to approximate the convex hull. Specifically, these representative vertices form a convex polygon contained within the overall hull. In phases (ii) and (iii), the bridges are initially found for the convex polygons described by the representative vertices in the header arrays. Once this has been done the bridges for the full convex hulls can be found by examining a constant number of hull arrays. Intuitively, this bridge finding corresponds to performing binary (or k-ary) search in a sorted table.

We summarize the main result of this section as:

**Theorem 2.5.1:** The convex hull of a sorted set of $n$ points can be constructed in $O(\log n)$ parallel time using $n/\log n$ CREW processors.

We note in passing that this sorted point set convex hull algorithm can be modified to find the convex hull of a simple polygonal chain within the same time and processor bounds.

### 2.6 Lower Bounds

We have repeatedly stressed the link between the common tangents/separation problems and sorted table lookup. We make this precise with a reduction.

Recall that the sorted table look-up problem is defined as follows: given a sorted table of $n$ real numbers $x_1, x_2, \ldots, x_n$ and a search key $x_s$, return the index $i$ such that $x_s \in [x_i, x_{i+1}]$ (with $x_0 = -\infty$ and $x_{n+1} = \infty$). To deal simply with the extreme table values, we consider the normalized sorted table look-up problem where $x_1 = 0$, $x_n = 1$, and $x_s \in (0, 1)$. 
**Theorem 2.6.1:** The sorted table-lookup problem can be reduced to convex polygon common tangents and separation.

**Proof:** Given a sorted table $X = \{x_1, x_2, \ldots, x_n\}$ such that $x_1 = 0$ and $x_n = 1$, and a search key $x_s \in (0, 1)$, associate with each $x_i$ the coordinate $P(x_i) = (x_i, -x_i^2)$. This associates with each table entry a vertex on a parabola opening downwards. Therefore, $X$ defines the vertices of a convex polygon $P(X)$. Consider the key $x_s$; $P(x_s)$ is also a vertex on this parabola. It is easy to see that,

(i) $x_s \in [x_k, x_{k+1}]$ iff $P(x_s)$ has common tangents with $P(x_k)$ and $P(x_{k+1})$; and

(ii) $x_s \in [x_k, x_{k+1}]$ iff the separation of $P(x_s)$ with $P(X)$ is realized by a point on the segment $P(x_k)P(x_{k+1})$.

Therefore, a solution to either of these problems provides an equivalent solution to the sorted table-lookup problem.

Snir [S85] has shown that in the CREW PRAM model of computation, with $k$ processors available, the sorted table look-up problem has a time bound of $\Theta(\log n/(1 + \log k))$. In the absence of concurrent read, the sorted table look-up problem has time bound $\Theta(1 + \log n - \log k)$. This is a key result which demonstrates that processors with a concurrent read facility are strictly more powerful than processors without concurrent reads. Snir’s lower bound results are proven for a powerful model of computation called an *ideal PRAM* [KR88] in which each processor can do any amount of local processing or transfer any amount of information in unit time. These lower bounds together with Theorems 2.3.4, 2.4.3, 2.4.4 and 2.6.1 immediately yield:

**Corollary 2.6.2:** Finding the common or separating tangents of a vertex with an $n$-vertex convex polygon with $k$ CREW processors has time complexity $\Theta(\log n/(1 + \log k))$. 
Corollary 2.6.3: Finding the separation of a vertex with an n-vertex convex polygon with k CREW processors has time complexity $\Theta(\log n/(1 + \log k))$.

The algorithms presented in Sections 3 and 4 demonstrate that the tight bounds stated in Corollaries 2.6.2 and 2.6.3 for the vertex/polygon problems in the CREW model extend to the polygon/polygon versions of those problems\textsuperscript{4}. The reduction of Theorem 2.6.1 has implications for the EREW model as well:

Corollary 2.6.4: Finding the common or separating tangents of a vertex with an n-vertex convex polygon with k EREW processors has time complexity $\Theta(1 + \log n - \log k)$.

Corollary 2.6.5: Finding the separation of a vertex with an n-vertex convex polygon with k EREW processors has time complexity $\Theta(1 + \log n - \log k)$.

The algorithms providing the upper bound for Corollaries 2.6.4 and 2.6.5 are the obvious extensions of the EREW k-way search algorithm: the k processors are used in the first step to reduce the length of the input sequence by a factor of k and the (unique) winning processor whose subsequence contains the solution applies the sequential algorithm to obtain the final solution.

However, it is not clear how to design an $O(1 + \log n - \log k)$ algorithm for the polygon/polygon versions of those problems using the EREW model of computation. In the CREW model, the $\sqrt{k}$-subdivision technique introduces a constant factor which does not affect the asymptotic complexity of the solution. This technique does not seem to be useful for these

---

\textsuperscript{4} Note that within the vertex-polygon version of the algorithms described a full concurrent read facility is not necessary, a facility which allows one processor to broadcast to all other processors would suffice.
problems in the EREW model. In fact, given two \( n \)-vertex polygons, it is not clear how to obtain an asymptotic speedup for \( n^\alpha \) EREW processors for any \( \alpha \leq 1 \).
Chapter 3: Parallel Search in Restricted Subdivisions

3.1 Introduction and Related Work

In this chapter, the geometric hierarchy paradigm is applied to point location problems in certain restricted types of planar subdivisions which arise naturally in a number of applications. These are subdivisions which are sufficiently restricted that a very simple hierarchical structure to allow $O(\log n)$ sequential point location can be constructed in $O(\log n)$ time using $n/\log n$ processors. Examples are star-shaped polygons, furthest point and convex point set Voronoi diagrams and what we call Voronoi fringes.

The algorithms and data structures presented in this chapter are not a direct application of the geometric hierarchy representation (cf. section 1.5 and [DK90]). However, the geometric hierarchy paradigm was pivotal in the development of a data structure described in this chapter which evolved into a general tree-contraction algorithm.

In the last chapter we exploited the relationship between a linear subdivision and the boundary of a convex polygon to develop polygon separation and tangent algorithms. To motivate this relationship we described a point in (convex) polygon algorithm which was essentially a sorted-table lookup algorithm. We use the same point in convex polygon containment problem to motivate the discussion of searching in restricted planar subdivisions.

Given a convex polygon $P$ on $n$ vertices presented in order in an array, we recall from the last chapter that a sequence of polygons $H(P) = P^0, P^1, ..., P^N$ that forms a polygon hierarchy is described simply: if the vertices of $P$ are $p_0, p_1, ..., p_{n-1}$, then the vertices of $P^t$ are taken to be those vertices of $P$ whose index is congruent to $0 \mod 2^t$ and the boundary of $P^t$ is defined
by the chords (and edges) of \( P \) connecting consecutive vertices of \( P^l \). This produces a sequence of \( O(\log n) \) convex polygons (with the original polygon \( P \) being first and a constant size polygon/triangle \( P^N \) being last). If we superimpose the polygons in this sequence, we obtain a decomposition of the interior of \( P \). Aggarwal et al [ACGOY88] use this kind of decomposition and refer to it as a *recursive triangulation* of a convex polygon.

Given a query point \( q \), we can use this superimposed structure to determine whether or not the polygon \( P \) contains \( q \) in \( O(\log n) \) sequential time. The recursive triangulation of the polygon implicitly defines a binary tree structure on the triangle in the decomposition by associating a tree node with each triangle and connecting two nodes if they share a mutual chord. The search algorithm begins by testing \( q \) for containment within the innermost polygon/triangle \( P^N \) (the root of the tree). To borrow the half-plane notation of chapter 2, if \( q \) is contained within the current triangle \( abc \) (that is, \( q \in H_{ab}^+ \cap H_{bc}^+ \cap H_{ca}^+ \)) then \( q \) is contained within \( P \). Otherwise, if \( q \in H_{ab}^- \) and \( ab \) is an edge of \( P \) or if \( q \) is contained in exactly two of \( H_{ab}^- \), \( H_{bc}^- \) and \( H_{ca}^- \) then, by convexity, \( q \) cannot be contained within \( P \) (cf. section 2.2).

Otherwise, there is a unique triangle/tree node to step to and test against. Since the maximum distance between any two nodes of the tree is \( O(\log n) \), this search algorithm can determine containment in \( O(\log n) \) sequential time. This 'tree search' algorithm can be recast in our hierarchical search paradigm; each time we step to a new tree node, we are actually taking a step in the polygon hierarchy.

The recursively triangulated decomposition defined by the polygon hierarchy above can be viewed as a bottom-up construction formed by recursively removing triangles, effectively creating levels of a search tree. An equivalent structure can be built in a top-down fashion: Select a distinguished vertex \( v \) and insert a chord between \( v \) and the vertex which is the \( n/2^{nd} \) vertex from \( v \). This subdivides the polygon into two polygons on either side of the inserted
chord both containing the vertex v. Recursively subdivide the polygons produced until every vertex has a chord connecting it to v. This defines a hierarchical structure which can be used to answer a point in polygon query in logarithmic time. This also defines a recursive triangulation of the polygon. Using this structure, the point in polygon query can be answered in $O(\log n)$ sequential time by performing binary search on the inserted chords.

Although we have described an explicit construction for this subdivision scheme, if the polygons vertices are presented in order in an array then this scheme can be applied implicitly without preprocessing. As well this top-down scheme can be extended implicitly to cooperative k-way search in a natural way by selecting a distinguished vertex and considering $(k - 1)$ evenly-spaced chords at each decision point.

There are other easily-defined subdivisions which contain enough structure to superimpose a simple hierarchy which can be used for answering point location queries. A planar subdivision defined by a set of parallel lines, or by rays emanating from a point can use this kind of tree structure for search. This last example can be modified to produce a point in star-shaped polygon algorithm by selecting a point in the kernel and placing a ray through each of the vertices in the polygon. Once the two consecutive rays bounding the region have been determined, a final half-plane test against the edge contained in that region suffices to answer the point in polygon question.

The common ingredient in the point in polygon schema described above is the ability to superimpose a binary search structure on a decomposition formed from an ordering of the vertices. If the vertex ordering is not specified as part of the input, but element adjacency is

---

1 Cole & Goodrich [CG88] present an $O(\log n)$ time-$n/\log n$ processor algorithm for constructing the kernel of an $n$-vertex starshaped polygon.
defined then it is possible to use list-ranking to combine the adjacencies into a total ordering. The remainder of this chapter is devoted to a particular kind of restricted subdivision called a region tree. A region tree can be preprocessed for sequential point location in $O(\log n)$ time using $n/\log n$ EREW processors. Although this is not an asymptotic improvement over currently available techniques for parallel construction of a data structure for planar point location (cf. descriptions of [CZ87] and [TamV89] in section 4.1), its simplicity and its relationship to tree contraction justify its examination.

In section 3.2, we present a general tree contraction algorithm based on a reduction to list-ranking using the Euler Tour technique. The design of this tree contraction algorithm is based on the geometric hierarchy paradigm. In section 3.3, tree contraction is used to preprocess an abstraction of a class of planar subdivisions called region trees for efficient point location. We show that searching in certain restricted (but natural) types of planar subdivisions can be reduced directly to region tree search.

### 3.2 Tree Contraction

The tree contraction problem is to reduce a rooted ordered tree to its root by a sequence of independent vertex removals. As discussed in section 1.3, the tree contraction problem can be viewed as an instantiation of the expression evaluation problem. We describe an efficient parallel algorithm for tree contraction based on a simple reduction to list ranking. The reduction for trees with $n$ nodes can be implemented on an exclusive read–exclusive write (EREW) PRAM in $O(\log n)$ time and using $n/\log n$ processors. This reduction of tree contraction to list ranking allows us to construct a simple tree contraction algorithm which runs in the cost of the chosen list ranking algorithm. As a consequence, we have a solution to parallel tree contraction which demonstrates optimal speedup with respect to the linear sequential time algorithms.
outlined in section 1.3 using the list ranking results of Cole and Vishkin [CV86b] or Anderson and Miller [AM88].

Parallel algorithms for tree contraction have been the subject of much research in recent years. Most of the earlier results have been described for the stronger concurrent read/exclusive write (CREW) model though they do not appear to make any essential use of concurrent reads. Miller and Reif [MR85] describe a deterministic algorithm which runs in $O(\log n)$ time with $n$ processors. They present their algorithm in the context of dynamic expression evaluation for an expression presented in the form of a parse tree. A single step of their algorithm converts a current binary parse tree to a simpler one by removing in parallel all leaves (the RAKE operation) and compressing maximal chains of nodes with only one child using pointer-jumping (the COMPRESS operation). They show that after $O(\log n)$ such steps a given tree is reduced to its root. They also present a randomized version of their algorithm which uses only $n/\log n$ processors.

Miller and Reif apply their method to construct parallel algorithms for problems which can be reduced to computation on trees. They give a randomized algorithm for testing isomorphism of trees, a deterministic algorithm for constructing a tree of 3-connected components of a planar graph, and other algorithms.

Among the important contributions of Miller and Reif's work is their abstraction of the problem of tree contraction. This leads to a separation of the problem from its familiar applications (notably, dynamic expression evaluation) and places it, along with list ranking, among the fundamental problems of parallel computation.

He [H86] defines the binary tree algebraic computation (BTAC) problem and applies Miller and Reif's technique to obtain a parallel algorithm for this problem. Roughly speaking
the BTAC problem is to compute the value of an algebraic expression given in the form of a
parse tree under the assumption that the algebra in which the computation is performed has a
finite carrier.

Gibbons and Rytter [GR86] present an optimal algorithm for dynamic evaluation of
algebraic expressions. Their algorithm runs in $O(\log n)$ time using $n/\log n$ processors. They
assume, however, that the input (the string representing the expression to be computed) is
given in an array and is hence preordered. They apply the algorithm of Bar-On and Vishkin
[BV85] to obtain a parse tree. The leaves of the parse tree are numbered from left to right on
the basis of their index in the input array. This numbering makes it possible to group leaves in
blocks of $O(\log n)$ size and then to perform computation in each block in parallel. As a result
the parse tree is contracted to a tree of size $O(n/\log n)$, at which point the algorithm of Reif and
Miller is used to compute the tree contraction using $n/\log n$ processors. Similarly to He, they
prove that the algorithm can be applied to compute an algebraic expression in any algebra with
finite carrier.

Cole and Vishkin [CV88] propose an alternative method for computation on trees. They
solve the tree contraction problem by parallel reduction to the list ranking problem. In this
respect their approach is similar to ours. Our reduction, an abstraction of the technique used
originally for the parallel preprocessing of region trees for fast (sequential) subdivision search
[DaK87a], is simpler and more explicit than that of Cole and Vishkin; in particular, it
completely avoids the centroid decomposition techniques that lie at the heart of their reduction.

Cole and Vishkin's tree contraction algorithm uses the same two basic parallel operations as
ours. The first stage of Cole and Vishkin's algorithm is to compute for each tree vertex $v$ the
function $\text{SIZE}(v)$, which is equal to the number of nodes in the subtree rooted at $v$. This step is
done using the Euler tour technique [TV84]. Then, using the SIZE function, the tree is partitioned into the so-called centroid paths. The bypass operation can be applied only to a node which does not have a non-centroid child. The next step of the algorithm (called the scheduling stage) is to compute the order in which the prune and bypass operations should be performed. In the last stage, called the evaluation stage, the prune and bypass operations are performed according to the order computed in the previous stage. In the evaluation stage some work has to be done in order to assign processors to the jobs.

In our algorithm the reduction to list ranking is done in a simpler way. The only information which is used to schedule the order of performing bypass operations is the initial numbering of the leaves. Consequently we do not need a separate scheduling phase, which in Cole and Vishkin's algorithm is quite complicated. As well, processor assignment is simpler in our approach.

Kosaraju and Delcher [KD88] have presented a tree-contraction algorithm which uses the same basic operations as the algorithm presented here. They view this algorithm to be a simplification of the Gibbons and Rytter algorithm and do not explicitly discuss a reduction to list ranking. They describe how to evaluate all subexpressions of an arithmetic expression of size $n$ in $O(\log n)$ parallel time using $n / \log n$ EREW processors.

### 3.3 Reduction to list ranking

We assume that trees are presented as an (unordered) array of vertices each of which has associated with it a parent pointer and a doubly-linked list of children. There are techniques for efficiently converting to or emulating such a representation from other more primitive
representations (cf. [ADKP87b]). (Successive) vertices on any list of children are said to be (immediate) siblings. Let $T$ be any binary tree with vertex set $V(T)$. We cast our approach to tree contraction within the hierarchical paradigm: A sequence of trees $T_1, T_2, \ldots, T_N$ is said to be a $\text{tree contraction sequence}$ of length $k$ for $T$ if,

(i) $T_1 = T$;
(ii) $|V(T_N)| \leq 3$;
(iii) $V(T_i) \subseteq V(T_{i+1})$; and
(iv) if $v \in V(T_{i+1}) - V(T_i)$ then either
   (a) $v$ is a leaf of $T_{i+1}$, or
   (b) $v$ has exactly one child, $x$, in $T_{i+1}$, $x \in V(T_i)$, and the parent of $v$ in $T_{i+1}$ is the parent of $x$ in $T_i$.

![Prune and Bypass operations](image)

Figure 3.1: Prune and Bypass operations
It is clear from the definition that successive trees in a tree contraction sequence are formed by "independent" executions of the following two fundamental operations (see figure 3.1):

Prune(v) — leaf v is removed from the current tree; and Bypass(v) — non-root node v with exactly one child x is removed from the current tree, and the parent w of v becomes the new parent of x (with x replacing v as a child of w).

By "independent" we mean that if v is pruned or bypassed then its parent is not bypassed. In this way tree modifications are ensured to be local and executable in parallel.

It is clear that every binary tree with n nodes admits a contraction sequence of length n; simply prune n − 1 times in succession until the tree has been reduced to its root. On the other hand, since just over half of the nodes can be removed in any one step, every contraction sequence must have length at least log(n) − 1. We say that a contraction sequence is optimal if it has length O(log n). (Note, it may not be immediately obvious that every binary tree has an optimal contraction sequence, let alone that such a sequence can be constructed efficiently.)

The tree contraction problem is to construct, for an arbitrary binary tree T, an optimal tree contraction sequence for T. It is not necessary to construct the sequence explicitly; it suffices to associate with each node v the index i of the highest indexed tree containing v in the sequence, together with pointers to the parent and child (if any) of v in T.

It is clear that if T has height h then T admits a contraction sequence of length h consisting entirely of prunes but when T is unbalanced such a sequence is far from optimal. Miller and Reif [MR85] introduce the idea of path compression — a familiar operation in the context of list ranking — to cope with trees, or parts thereof, that have become so unbalanced that they have degenerated into paths. Miller and Reif show that by interleaving global pruning (that is,
the removal of all leaves) with global path compression (the compression of all paths of length greater than one) it is possible to construct an $O(\log n)$ contraction sequence in $O(\log n)$ time using $O(n)$ processors. The complications of Miller and Reif's approach, especially with regard to processor reduction, arise in the compression step. Our approach differs in that we define the primitive contraction operations at a more elementary level. By scheduling the order in which leaves are eliminated it is possible to completely eliminate the difficulties associated with path compression. Indeed, paths of length greater than two never arise as our algorithm contracts an (initially regular) binary tree. Another interesting feature of our algorithm is that a leaf remains a leaf until it is removed and an internal node remains an internal node until it is removed.

The pair of operations prune($v$) followed by bypass(parent($v$)) (where $v$ is any leaf) form a conceptual unit in our algorithm. The algorithm proceeds in phases, each of which consists of a batch of these basic contractions performed in parallel. The independence of the underlying operations is guaranteed by a simple global schedule for leaf removal. Let the leaves be numbered in left to right order. A leaf is removed in phase $t$ if the rightmost 1 in its leaf index is in position $t$.

Our binary tree contraction algorithm has the following simple description:
procedure contract (T)
begin
(* Assign leaf indices from 0 to \( n - 1 \) *)
for each leaf \( v \) in parallel
    index\((v)\) ← left_to_right leaf index of \( v \)
(* Contraction iterations. *)
repeat \([\log n]\) - 1 times
    for each leaf \( v \) in parallel
        \( w \leftarrow \) parent \((v)\)
        if index\((v)\) is odd and \( w \neq \) root
            then if \( v \) is a left child
                then prune \((v)\)
                bypass \((w)\)
                if \( v \) is a right child
                then prune \((v)\)
                bypass \((w)\)
            else index\((v)\) ← index\((v)\) / 2
end;

Note that the innermost if statements, though they have opposite conditions, are intended to be executed in sequence, with appropriate synchronization in between. Thus, each iteration of the repeat loop has four slots in which prune or bypass operations may be executed. Accordingly, we associate four elements of the tree contraction sequence with each iteration of the repeat loop, describing the tree after each of the four slots. It is also helpful to view the behaviour of the algorithm at two other levels. It is immediate from the description that each prune operation is immediately followed by a bypass. Hence in each successive pair of slots a number of composite prune-bypass operations are executed in parallel. Each pair of these composite slots (making up an entire iteration of the repeat loop) serves to eliminate all of the leaves with odd parity in the current tree together with their parents.

Lemma 3.3.1. Procedure contract constructs an optimal tree contraction sequence.
Proof: It suffices to demonstrate that the prunes and bypasses are performed independently. Since prunes and bypasses are never executed simultaneously, it need only be demonstrated that no vertex \( v \) and its parent \( w \) are ever bypassed simultaneously. Suppose this is not the case. Without loss of generality, \( v \) is the right child of \( w \). Since they are bypassed simultaneously, they must both have leaves as left children. But since these leaves are adjacent in the left to right order and since the index array maintains each leaf's left-to-right rank in the current contracted tree, \( v \) and \( w \) must have indices of opposite parity, a contradiction.

Theorem 3.3.2. Procedure contract provides an \( O(\log n) \) time and \( n / \log n \) processor deterministic reduction of tree contraction to list ranking.

Proof: First note that there is a straightforward reduction of the leaf ranking problem, which arises in the first step of procedure contract, to the list ranking problem. The method is called the Euler Tour technique [TV84]. Each node \( v \) of \( T \) is split into three nodes \( v_T, v_L \) and \( v_R \). For each of the resulting nodes we define a next field as follows. If \( v \) is a leaf then \( v_T\.next = v_L \) and \( v_L\.next = v_R \). If \( w \) is the right child of \( v \) then \( v_L\.next = w_T \) and \( w_R\.next = v_R \). If \( w \) is the left child of \( v \) then \( v_T\.next = w_T \) and \( w_R\.next = v_L \). What results is a list that starts at root_T and ends at root_R and traverses each edge of \( T \) once in each direction. If we assign weight(\( z \)) = 1 if \( z = v \) and \( v \) is a leaf and weight(\( z \)) = 0 otherwise, then the weighted rank of each node \( v_T \), where \( v \) is a leaf, gives the leaf index of \( v \) in \( T \).

As a consequence of the above, it suffices to prove that the iterated contraction step of procedure contract can be implemented in \( O(\log n) \) time using \( n/\log n \) processors. An \( O(\log n) \) time, \( n \) processor implementation is immediate; if one processor is devoted to each leaf then each phase can be carried out in \( O(1) \) time. On the other hand if each of \( n / \log n \) processors is assigned to a block of \( \log n \) successive leaves, the obvious simulation of the \( n \)
processor implementation incurs an additional additive overhead of only $O(\log n)$ time (cf. \[B74\], Lemma 2).

It follows from Theorem 3.3.2 that results for list ranking carry over directly to tree contraction. We summarize the most important implication in the following:

**Corollary 3.3.3.** The tree contraction problem can be solved deterministically in $O(\log n)$ time using $n / \log n$ processors.

**Proof:** This is an immediate consequence of the $O(\log n)$ time $n / \log n$ processor deterministic list ranking algorithms due to Cole and Vishkin [CV86b] and Anderson and Miller [AM88]. Alternatively, there exist practical randomized parallel list ranking algorithms that achieve the asymptotically optimal bounds [MR85] [ADKP87a].

### 3.4 Geometric Applications of Tree Contraction: Region Trees

We now examine a geometric application of tree contraction. We use tree contraction to construct a data structure which is then available for individual queries. Although the tree contraction sequence can be constructed using exclusive write, it seems natural in this context to allow concurrency in the query processing, hence we describe this application using a CREW model of computation.

We define an abstraction of a class of planar subdivisions called region trees and show that tree contraction can be used to preprocess region trees for fast point location. We then show that searching in certain restricted types of planar subdivisions can be reduced directly to region tree search.
A region tree is an embedded binary tree with an assigned region associated with each node in the tree such that

1) the assigned region associated with a leaf of the tree is empty,

2) the assigned region associated with an internal node of the tree is non-empty and contains all assigned regions associated with all its descendent nodes,

3) given two nodes of a region tree, if neither node is a descendant of the other then their assigned regions are disjoint, and

4) a query point's containment within a node's assigned region can be determined in constant time.

Given a region tree and a query point, a containing node is a vertex whose assigned region contains the query point. The minimal containing node is the vertex whose assigned region contains the query point but none of whose descendants do. The region tree point location problem is the problem of identifying the minimal containing node of the region tree for the query point.

Tree contraction can be used to preprocess a region tree into a sequence of increasingly simple region trees (a region tree hierarchy) effectively defining a binary search structure on the tree.

Let R be a region tree as defined above. A sequence of region trees $H(R) = R_1, \ldots, R_k$ is said to be a region tree hierarchy of R if

i) $R_1 = R$;

ii) $R_1 \ldots R_k$ is the tree contraction sequence of R.
Theorem 3.4.1: Given a region tree $R$ on $n$ vertices and $O(n)$ query points, the minimal containing node for each of the query points can be identified in $O(\log n)$ parallel time using $n$ processors.

Proof: By Corollary 3.3.3, the tree sequence, $H(R)$, can be constructed in $O(\log n)$ parallel time. A processor is assigned to each query point. Locating the minimal containing node for a query point then follows the standard search method associated with the geometric hierarchy paradigm: The root of the last element of the tree sequence $R_k$ is set as the minimal containing node for that element. Knowing the location of a point in a given element $R_{i+1}$ of the sequence allows a single processor to determine its location in the preceding element $R_i$ in $O(1)$ time. Since there are $O(\log n)$ elements in the tree sequence, a query point's minimal containing node for the first element of the tree sequence (the original region tree) $R_1$ can be determined sequentially in $O(\log n)$ time.

It is possible to solve subdivision search for certain restricted subdivisions by direct reduction to region tree search:

Theorem 3.4.2: Given a planar subdivision on $n$ vertices and $O(n)$ query points, if it is possible to form a region tree from the subdivision such that the Assigned Region associated with a node minus the Assigned Regions associated with its descendants intersects a constant number of subdivision regions then the subdivision search problem can be solved in $O(\log n)$ time using $n$ processors.

Proof: By Theorem 3.4.1, the minimal containing nodes for the query points can be found in $O(\log n)$ parallel time. Once the minimal containing node has been determined for a query point, only a constant amount of further work is necessary to determine its containing subdivision region.
Several natural forms of restricted planar subdivision search can be reduced directly to region tree search. Following Aggarwal et al [ACGOY85] [ACGOY88], we outline a divide and conquer Voronoi diagram construction algorithm whose merge step involves determining which regions of the diagram are incident on the merge contour. The algorithm is derived from the divide and conquer approach of Shamos and Hoey [SH75]. In that sequential algorithm the point set is sorted, and subdivided into two linearly separable halves. The Voronoi diagram of each half is constructed recursively and the results are combined by tracing the contour segments dividing them.

Aggarwal et al noted that two separable Voronoi diagrams could be merged in parallel by predetermining which regions of each were incident on the contour. This can be done by locating where the contour 'slices' each of the Voronoi diagrams and thus discovering the incident regions.

For completeness, we present a high level description of the algorithm of Aggarwal et al. Let S be the input point set (of size \( n \)) and V(S) its Voronoi diagram:

1. Sort S by x-coordinate using the \( O(\log n) \) time - n processor algorithm of Cole [C86b]. In the following recursive description of the algorithm, we assume that we have two point sets P and Q separated by a line L, and their Voronoi diagrams V(P) and V(Q) which we are to merge into V(S).

2. Determine which regions are incident on the contour by locating the Voronoi points beyond the fringe of V(P) in Q's regions (and vice-versa).

3. Given the sequences of regions in P and Q contributing to the contour, compute the actual edges which constitute the contour in \( O(\log n) \) time.

4. Update the V(P) and V(Q) structures with the determined contour edges to produce V(S) using \( O(\log n) \) parallel time.

Their original algorithm [ACGOY85] erred in the implicit assumption that the contour would not intersect the convex hull of either point set. In fact, the contour may intersect either
hull repeatedly. However, for a Voronoi diagram and a separating line, as would be available in the merge step of the divide and conquer algorithm described above, we can determine quickly and describe simply the extent to which a contour can penetrate the Voronoi diagram. (This is similar to the approach taken by Aggarwal et al in their revised paper [ACGOY88].)

Given a Voronoi diagram and a separating line, the Voronoi fringe chain is defined as a chain of segments which delimits the extent to which the contour may penetrate the point set; the Voronoi fringe is defined as the region beyond the fringe chain. We will elaborate on step [2] of the algorithm and describe in more detail the structure of the fringe.

Consider a Voronoi point v which is equidistant from three data points q_1, q_2, and q_3 of Q. Note that if the circle centred at v which passes through q_1, q_2, and q_3 contains no points of P then v will be a Voronoi point in the merged structure. Otherwise, the contour 'slices' the vertex v off.

![Figure 3.2 Slicing off a Voronoi point](image-url)
This suggests a way of computing the fringe chain of a Voronoi diagram. Any Voronoi edge in P's Voronoi Diagram which has one endpoint closer to a point in Q than its closest point in P is a candidate for being sliced by the contour. Denote by L the line separating the P and Q point sets. The farthest influence the points in Q can have is the line L. Every point on a Voronoi edge in P's diagram is equidistant from two points in P. Thus, a fringe segment endpoint is a point on a Voronoi edge which is equidistant from its two contributing points of P and a point on the separating line L.

Assume that a processor has been assigned to each Voronoi edge. Each can determine in constant time whether or not its edge contains 0, 1, or 2 fringe segment endpoints. A Voronoi edge e will have a pointer to the two data vertices a and b that it separates; given the separating line L, consider the two circles through a and b which have the separating line L as a tangent. The centres of these circles lie on the line through the edge e. If the centre actually lies on the edge e, it defines a fringe segment endpoint. Therefore in constant time a single processor can determine whether or not the edge e contains 0, 1, or 2 fringe segment endpoints. If it does, it can also determine in constant time what its fringe segment neighbours are. List ranking will allow each point to determine its index in the sequence of fringe points in logarithmic time. Note that the sequence of segments defining the fringe chain is monotone with respect to the separating line L.

As noted by Aggarwal et al [ACGOY85] [ACGOY88], if the data points of S are in general position then the structure extending past a convex hull edge has the form of a complete binary tree with the bisector of the two hull points extending from the root to infinity. The regions within this tree (like all Voronoi regions) are convex. Since the fringe will span a number of hull edges then a number of trees will extend into the fringe region with their leaves on the
fringe chain. Given a query point, binary search on the rays corresponding to the (infinite) hull point bisectors will determine which 2 fringe trees to search in.

**Corollary 3.4.3:** Point location in a Voronoi fringe can be reduced to region tree search. Thus a search structure for a Voronoi fringe (permitting $O(\log n)$ sequential query time) can be constructed in $O(\log n)$ parallel time using $n/\log n$ processors.

**Proof:** We omit the details of the region assignment but note that each Voronoi Fringe tree can be augmented such that it can be associated with a region tree such that its component regions satisfy the requirements of Theorem 3.4.1. §

Therefore, using this result in the Voronoi diagram construction algorithm due to Aggarwal et al [ACGOY88], we have immediately:

**Corollary 3.4.4:** The two-dimensional Voronoi diagram for a set of n points can be constructed in $O(\log^2 n)$ parallel time using $n$ CREW processors and $O(n)$ space.

For the following two corollaries, we note (as in the case of Voronoi fringes described above) that a farthest point Voronoi diagram and a Voronoi diagram constructed on the vertices of a convex polygon each describe an embedded binary tree. We omit the details of a region assignment necessary to fully specify the reductions but simply state the following results:

**Corollary 3.4.5:** Point location in a furthest point Voronoi diagram on $n$ points can be reduced to region tree search. Therefore, a point location search structure for a furthest point Voronoi diagram (with $O(\log n)$ sequential query time) can be constructed in $O(\log n)$ parallel time using $n/\log n$ CREW processors.

**Corollary 3.4.6:** Point location in a Voronoi diagram constructed on the $n$ vertices of a convex polygon can be reduced to region tree search. Therefore, a point location search
structure for the Voronoi diagram constructed on the $n$ vertices of a convex polygon can be constructed in $O(\log n)$ parallel time using $n/\log n$ CREW processors. Furthermore, this data structure supports $O(\log n)$ sequential query time.
Chapter 4: Parallel Processing for Search in Planar Subdivisions

4.1 Introduction and Related Work

Having addressed the question of parallel preprocessing of restricted planar subdivisions for efficient point location, it is natural to turn our attention to general planar subdivisions. In this chapter, the geometric hierarchy paradigm is applied to the problem of efficiently preprocessing a planar subdivision in parallel for fast sequential search. This is achieved by giving a direct parallel construction of the geometric hierarchy representation as applied to subdivision search by Kirkpatrick [K83].

Subdivision search in the sequential setting has been well-studied; the descriptions of the various approaches to planar subdivision search are presented in roughly chronological order. Dobkin and Lipton [DL76] [PS85] argued that the key to an efficient solution to the planar subdivision search problem was devising some way of performing binary search on the elements in the subdivision. They applied this principle to obtain their algorithm referred to as the slab method by Preparata and Shamos [PS85]. This involves defining horizontal lines through each of the vertices in the subdivision thus forming horizontal slabs, each of which consists of a horizontal sequence of (possibly degenerate) trapezoids. Storing the slabs and the arrangements of regions with each slab allows two applications of binary search to determine the region containing a query point. For an n-vertex subdivision, this method requires $O(n^2)$ space and $O(n^2 \log n)$ preprocessing time but allows $O(\log n)$ query time. The preprocessing time can be reduced to $O(n^2)$ by employing a plane-sweep technique to construct the slabs [PS85].
In an alternate approach which Preparata and Shamos [PS85] call the *chain method*, Lee and Preparata [LP77] demonstrated how to decompose a (possibly augmented) planar subdivision into a set of monotone chains. The associated point location algorithm performs binary search on the chains using another instance of binary search to discriminate against any individual chain. For an n-vertex planar subdivision, this method requires $O(n)$ space and $O(n \log n)$ preprocessing time while allowing $O(\log^2 n)$ point location.

Subsequently, Lipton and Tarjan [LT77] produced a construction which demonstrated that it was possible to build a data structure to answer a point location query in an n-vertex planar subdivision in $O(\log n)$ time while requiring only $O(n)$ space and $O(n)$ preprocessing time. Their algorithm makes use of a planar separator theorem [LT79] in a fairly complicated construction. Their paper was noteworthy for demonstrating the existence of a technique optimal in time and space but their algorithm was not advocated as a practical solution to the point location problem. The question of using planar separators in the preprocessing of a planar subdivision for efficient point location is addressed further in chapter 5.

Kirkpatrick [K83] building on the Lipton and Tarjan results [LT77] and the batching approach used in the graph colouring algorithm of Lipton and Miller [LM78] presented an optimal subdivision search algorithm which is an instance of the geometric hierarchy paradigm explored here. Also known as *triangular refinement* [PS85], this algorithm is the basis of the Subdivision Hierarchy construction algorithms presented in this chapter and is described in greater detail in the next section (4.2).

Another point location algorithm of interest is the *trapezoid method* due to Preparata [P81] [PS85]. It is an extension of the slab method which uses the observation that edges which span a number of slabs are useful for reorganizing the search structure to reduce the space and preprocessing bounds. The preprocessing builds a hierarchical binary search structure for the planar subdivision by recursively subdividing trapezoids. The initial trapezoid is initialized by
placing vertical lines at positive and negative infinity and using the maximum and minimum y-coordinates as the top and bottom. Given a trapezoid, it is subdivided using the horizontal line through the median y-coordinate of the vertices within the trapezoid and edges which span (one of) the resulting upper and lower trapezoids. The resulting trapezoids are recursively subdivided. Within each recursive subdivision, a final balancing operation ensures a balanced search structure. For an n-vertex planar subdivision, this construction takes \(O(n \log n)\) time and \(O(n \log n)\) space and allows \(O(\log n)\) query time.

Building on the chain method, Edelsbrunner, Guibas and Stolfi [EGS86] made the observation that in Lee and Preparata's algorithm, information gathered during the test of a query point against a given chain is not exploited in testing against other chains. They use a cascaded sampling technique in the design of an elegant data structure called a layered dag (directed acyclic graph). Conceptually, the data structure is a binary tree of catalogs: each internal node corresponds to a monotone chain and each leaf node corresponds to one of the subdivision facets. An entry in the catalog associated with an internal node has pointers to entries in that node's children. The query point is tested against the chain/catalog associated with the root node in logarithmic time. Given the query point's location in the catalog of a given internal tree node \(v\), the pointer structure allows the query point to be located in the catalog of \(v\)'s child node in constant time. For an n-vertex subdivision, the layered dag can be constructed in \(O(n)\) time using \(O(n)\) space and allows \(O(\log n)\) point location.

Cole [C86a] and Sarnak and Tarjan [ST86] both build on the slab method of Dobkin and Lipton [DL76]. Their common abstraction deals with a planar subdivision described as a sequence of lists derived from a vertical line sweeping horizontally across the subdivision. As a vertex is encountered, the vertical line finishes sweeping some of the regions incident on that vertex and begins sweeping others. Their respective data structuring techniques are applied to construct directed acyclic graph search structures for the sequence of lists corresponding to an
n-vertex planar subdivision in $O(n \log n)$ time, using $O(n)$ space and allowing $O(\log n)$ point location.

Recently, Seidel [S90] has described an algorithm for planar subdivision search which is based on a simple modification of the subdivision hierarchy approach. It can also be used to construct a search structure for an n-vertex planar subdivision in $O(n)$ time, using $O(n)$ space which allows $O(\log n)$ point location. It constructs a hierarchical search structure consisting of a sequence of subdivisions based on the batched removal of sets of independent edges (where independence is based on vertical visibility). Seidel claims that the constants associated with this method are smaller than those associated with the subdivision hierarchy and that the performance of this method is comparable to the fractional cascading methods [EGS86] [C86a][ST86] outlined above.

On parallel models of computation, previous work has concentrated on parallel preprocessing for fast sequential point location algorithms. Most of this research has concentrated on designing parallel implementations of the sequential techniques described above. In the following, we restrict discussion to results on the PRAM model of computation.

In her thesis, Chow [C80] describes a parallel implementation of a data structure based on the slab method of Dobkin and Lipton [DL76] with some modifications derived from Preparata's trapezoid method [P81] outlined above. For an n-vertex planar subdivision, Chow's data structure can be constructed by $n$ CREW processors in $O(\log^2 n \log \log n)$ time, uses $O(n \log n)$ space and allows $O(\log^2 n)$ sequential time point location.

Aggarwal et al [ACGOY88] present an algorithm for constructing the trapezoidal decomposition of an n-vertex planar graph in $O(\log^2 n)$ parallel time using $n$ CREW processors

---

1 This preprocessing time can be reduced to $O(\log^2 n)$ by employing the $O(\log n)$ sorting algorithm due to Cole [C86b].
and $O(n \log n)$ space. Although not originally designed for the planar point location problem, they note in passing that this structure does support $O(\log^2 n)$ sequential time point location (matching Chow's results). The parallel plane sweep aspect of their approach was improved on and applied to a variety of problems by Atallah and Goodrich [AG86b]. They demonstrated how to construct an augmented plane sweep tree for an $n$-vertex planar subdivision with $n$ CREW processors in $O(\log n \log \log n)$ time using $O(n \log n)$ space and allowing $O(\log n)$ sequential time point location. This construction was further improved by Atallah, Cole and Goodrich [ACG89] using parallel fractional cascading to reduce the parallel construction time of the plane sweep tree data structure to $O(\log n)$.

Building on the work of Preparata and Tamassia [PT88], Tamassia and Vitter [TamV89] demonstrate how to construct topological orderings derived from a *planar st-graph* $G$. Employing the parallel fractional cascading technique of Atallah, Cole and Goodrich [ACG89], Tamassia and Vitter extend these results to give a parallel implementation of the layered dag structure of Edelsbrunner, Guibas and Stolfi [EGS86]. For an $n$-vertex planar graph, this structure can be constructed by an $n$ processor EREW PRAM in $O(\log n)$ time, using $O(n)$ space and allowing $O(\log n)$ sequential query time. For monotone subdivisions, this construction can be performed using $n/\log$ processors in the same time and space bounds.

In this chapter, we give a parallel construction for subdivision hierarchies [K83]. As in the sequential setting, subdivision hierarchies are constructed by identifying and removing large independent sets of low-degree vertices to produce a sequence of progressively simpler subdivisions. Thus, much of our attention in this chapter is devoted to parallel algorithms for the identification of large independent sets in certain restricted graphs.

The subdivision hierarchy approach to the point location problem developed in this chapter was proposed by Aggarwal *et al* [ACGOY88] but was eventually abandoned in favour of a less general solution. However, several researchers use techniques similar to those presented here
to solve a variety of problems on planar and/or bounded degree graphs. These results are described at the end of section 4.2.

Section 4.2 is concerned with the Subdivision Hierarchy Construction problem. Through a series of problem reductions, we examine the relative complexities of different instantiations of the Subdivision Hierarchy Construction problem and relate it to the problem of identifying a Fractional Independent Set within a planar graph. This approach is used to describe a parallel construction for subdivision hierarchies. A subdivision hierarchy for an $n$-vertex subdivision $S$ which allows $O(\log n)$ time sequential search and employs $O(n)$ space can be constructed in $O(\log n \log^* n)$ (respectively, $O(\log n)$ expected) parallel time using a deterministic (respectively, randomized) algorithm.

In section 4.3, we relate our subdivision hierarchy algorithms to the problem of constructing a hierarchical representation of a convex polyhedron. This hierarchical representation, in turn, is applied to the solution of certain geometric intersection and separation problems. Among the results is an $O(\log^2 n \log^* n)$ parallel time algorithm for the construction of 3-dimensional convex hulls.

4.2 Subdivision Hierarchies and Fractional Independent Sets

Suppose $S$ is a planar subdivision and $V(S)$ is the vertex set of $S$. We denote by $G_S$ the associated embedded planar graph. A subdivision hierarchy representation of $S$ is a sequence of increasingly simpler descriptions of $S$. The first element of the sequence, $S_1$, is a triangulation of $S$. Each subsequent element of the sequence, $S_i$, is a triangulated subdivision whose size is some fixed fraction less than the size of its predecessor $S_{i-1}$ and each of whose regions intersects at most a constant number of regions of $S_{i-1}$. The last element of the sequence, $S_N$, is a subdivision with at most some fixed number of vertices. Since the sizes of successive subdivisions form a geometrically decreasing sequence, the number of elements in
the sequence is logarithmic in the size of the original graph and the overall space requirement is linear in n.

Casting this more formally into our hierarchical framework, there exist positive constants $\alpha > 1$ and $\beta$ such that for any triangulated planar subdivision $S$ with vertex set $V(S)$, there is a sequence of triangulated subdivisions $H(S) = S^1, \ldots, S^N$ (called a subdivision hierarchy of $S$) satisfying:

i) $S^1 = S$;

ii) $|S^N| = 3$;

iii) $V(S^{i+1}) \subseteq V(S^i)$ and $|S^{i+1}| < |S^i| / \alpha$; and

iv) for any vertex $v \in S^i$ either $v \in S^{i+1}$ or degree($v$) in $S^i$ is less than $\beta$ and all of $v$'s neighbors in $S^i$ are in $S^{i+1}$.

One of the implications of condition (iv) for a subdivision hierarchy is that each region $R$ of $S^{i+1}$ has associated with it at most $\beta$ regions of $S^i$ whose union includes $R$. This is the property which allows a point location query in $S^i$ to be answered in constant time given the points location in $S^{i+1}$.

We first review the sequential algorithm for constructing a subdivision hierarchy from a given $n$-vertex subdivision presented by Kirkpatrick [K83]. The original subdivision $S$ is fully triangulated to produce the first element of the sequence, $S^1$. Each subsequent element $S^i$ is derived from its predecessor $S^{i-1}$ by identifying, and removing a set of low-degree independent vertices and retriangulating the resulting subdivision. This continues until a subdivision $S^N$ with three vertices has been produced.

For any connected planar graph $G$ with $v$ vertices, $e$ edges and $f$ faces, Euler's formula [BM76] states that $v - e + f = 2$. A direct consequence of Euler's formula is that $e \leq 3v - 6$ and therefore that every planar subdivision has an average degree of less than 6 which implies
that fewer than half the vertices have degree exceeding 11. From the set of vertices \( V \) of degree at most 11, an independent set of size at least \( |V|/12 \geq n/24 \) can be identified quickly. Using this, Kirkpatrick [K83] shows that every subdivision has an associated subdivision hierarchy with \( \alpha = 24/23 \) and \( \beta = 11 \) which can be constructed in linear time exclusive of the initial triangulation. In another context, Lipton and Miller [LM78] (and subsequently Edahiro et al [EKA84]) showed that large independent sets of vertices of degree at most 6 can be easily identified. Accordingly, we say that a vertex has low-degree if it has degree less than or equal to 6.

Given a query point \( q \) and a subdivision hierarchy \( H(S) \) of a subdivision \( S \) on \( n \) vertices, the straightforward subdivision search algorithm falls into our general search paradigm. Since \( S^N \) is of constant size, the region of \( S^N \) containing \( q \) is identified in constant time. For each face \( f \) of \( S^{i+1} \) either \( f \) is a face of \( S^i \) or \( f \) was produced when a low-degree vertex of \( S^i \) was removed. Therefore, given the face of \( S^{i+1} \) containing \( q \), the face of \( S^i \) containing \( q \) can be determined in constant time. By stepping through the sequence in this way (each step taking constant time), the face of \( S^1 \) (hence \( S \)) containing \( q \) can be determined in \( O(\log n) \) time.

For completeness, we recall some implementation details relevant for both the sequential and parallel implementations. We assume as input the Doubly Connected Edge List (DCEL) [MP78] [DK90] in which the graph \( G_S \) underlying a planar subdivision \( S \) is represented as \( E \), an array of edges. Each edge \( e_1 \) in \( E \) has associated with it six pointers, one each to its originating and terminating vertices \( v_1 \) and \( v_2 \), one each to its two incident faces \( f_1 \) and \( f_2 \), a pointer to the edge \( e_2 \) incident on \( v_1 \) which is first counter-clockwise from \( e \) and a pointer to the edge \( e_3 \) incident on \( v_2 \) which is first counter-clockwise from \( e \).

This is modified slightly to represent the subdivision hierarchy. Each face \( f \) in \( S^{i+1} \) has a pointer to the (low-degree) vertex \( v \) in \( S^i \) (if one exists) whose removal caused the formation of \( f \). In the construction, when a vertex \( v \) is removed its neighbourhood is retriangulated and each
of the resulting faces points to v. An edge (v, w) has its corresponding pointer in w's edge ring removed. In the search, if f is the face of $S^{i+1}$ containing a query point q, there are only a constant number of (triangular) faces to search in order to locate q in $S^i$. The pointers from faces of $S^{i+1}$ to vertices of $S^i$ are used to 'bridge' consecutive elements (subdivisions) of the hierarchy and implement the directed acyclic graph structure used for the subdivision search. We note that the pointer space requirement for any particular subdivision of the sequence is dominated by the space requirement for that subdivision.

We define the **Subdivision Hierarchy Construction Problem** (SHCP) as the problem of constructing a subdivision hierarchy for an arbitrary subdivision on $n$ vertices. We will also use SHCP(n) (and variants) to refer to the parallel complexity of solving the Subdivision Hierarchy Construction Problem (and variants). The special cases when $S$ is a **Convex Subdivision** (it is bounded by a convex polygon and each of its interior faces is convex) – which we denote C-SHCP(n) – or when $S$ is a **Triangular Subdivision** (it is bounded by a triangle and each of its interior faces is a triangle) – which we denote T-SHCP(n) – are of independent interest.

As we have seen, it suffices at each stage to identify and remove a set of low-degree independent vertices which constitutes a fixed fraction of the entire vertex set. Therefore, we define the **Fractional Independent Set Problem** (FISP(n)) as the problem of identifying for an arbitrary planar graph G with $n$ vertices an independent set I of low-degree vertices in G such that $|I| > n/c$ for some fixed constant c. Special cases of this problem, BD-FISP(n) and L-FISP(n), concern the restriction to bounded degree graphs, and list graphs (digraphs whose vertices have in-degree and out-degree bounded by 1) respectively.

It is possible to relate the parallel complexities of variants of the SHCP and FISP by means of some straightforward reductions:
Lemma 4.2.1:

(i) \( \text{SHCP}(n) \leq O(\log n) + T-\text{SHCP}(n) \)

(ii) \( \text{C-SHCP}(n) \leq O(\log n) + T-\text{SHCP}(n) \)

(iii) \( T-\text{SHCP}(n) \leq O(\log n) \text{ FISP}(n) \)

Proof:

(i) This involves embedding the given planar subdivision within a triangular subdivision. A bounding rectangle can be determined for the given subdivision in \( O(\log n) \) time by finding the minimum and maximum \( x \) and \( y \) values among the vertices of the subdivision. From the bounding rectangle, a bounding triangle can be determined in constant time. Goodrich [G85] presents an algorithm which can be used to triangulate the interior polygons in \( O(\log n) \) parallel time using \( n \) processors.

(ii) This involves embedding the given convex subdivision within a triangular subdivision. First, a containing triangle is determined in \( O(\log n) \) time as in (i) above.

Next, the vertices on the boundary of the convex polygon containing the subdivision must be connected to the vertices of the containing triangle such that the region between the bounding polygon and the containing triangle is triangulated. In general this can be done in any of several different ways. The ring of edges defining the bounding polygon will be available in the DCEL; it is the edge ring corresponding to the external face. Using this ring, a vertex on the boundary can determine in constant time which (and how many) of the containing triangle vertices are visible by using its incident edges. Hereafter, it is straightforward to construct the desired triangulation using only local information.

Finally, the interior convex regions must be triangulated. Within a convex polygon, each vertex is visible from every other vertex (by the definition of convexity). Thus, the only problem to be solved in triangulating a convex polygon in parallel is making sure the inserted edges don't cross. Each edge can learn its rank (relative to a lowest numbered edge) in both of
its incident faces by optimal list ranking techniques [CV86a] [AM88]. Since this also ranks the vertices, it is straightforward to form a triangulation by connecting each vertex in the face to the lowest numbered vertex. Each convex region bounded by \( t \) edges can be triangulated in \( O(\log t) \) time using \( t / \log t \) processors.

(iii) This can be demonstrated by simply mimicking the sequential algorithm. For each element \( S^i \) in the sequence, construct \( S^{i+1} \) by identifying a fractional independent set of \( S^i \), removing this low degree independent set and retriangulating in constant time. The triangulation guarantees that any two consecutive edges incident on a given vertex \( v \) are two sides of a triangle, hence the three vertices involved are not independent. Therefore, although some vertex \( w \) incident to \( v \) may be high-degree, no consecutive edges in \( w \)'s edge ring will be removed in any particular iteration (i.e. all updates are local and can be performed in constant time). This is a by-product of the fact that we have an fully triangulated embedding; without such an embedding there may be many consecutive edges on a vertex's edge ring which are to be removed. The dominant cost of each of the \( O(\log n) \) iterations is the identification of the fractional independent set.

We note in passing that the problem of updating the edge rings of high degree vertices mentioned in the proof of (iii) above arises in an algorithm to accumulate a Maximal Independent Set of an (unembedded) planar graph by repeatedly finding fractional independent sets. In an EREW model, a 'clean-up' phase which contracts the edge lists associated with each vertex can be incorporated into each round [DaK87b] and an induction argument can be used to show that an overall \( O(\log n \log^*n) \) parallel time bound still holds. Alternately, an auxiliary graph can be used to avoid read/write conflicts [HCD89].

Lemma 4.2.2: \( \text{FISP}(n) \leq O(1) + \text{BD-FISP}(n) \)

Proof: Again we mimic the sequential algorithm. Since the low degree vertices form a fixed fraction of the vertices of a planar graph, it suffices to identify a subgraph of the input graph
such that all vertices have degree at most $b$. In the following description, the edge 'near end' and 'far end' locations are used to avoid read/write conflicts.

Each vertex processor marks its vertex high degree (as a default). It then counts its incident edges up to a maximum of $b + 1$. Any processor which counted up to $b + 1$ edges sits out and the others mark themselves low-degree. Each low-degree vertex marks the 'far end' of its incident edges High Degree and then marks the 'near end' of its incident edges Low Degree. It then reads the 'far end' of its incident edges and removes from its edge list the ones which are marked High Degree. In this way, all low degree vertices identify themselves, their low degree neighbours and the resulting induced graph in $O(1)$ parallel time.

The following pseudo-code procedure, which is executed in parallel by each of the vertex processors, describes in more detail the procedure to identify the graph induced on the low degree vertices.
procedure LowDegreeSubgraph;
begin
Mark vertex high-degree;
degree := 0;
test_edge := first_edge;
counted_all := false;

(* Count edges to identify low-degree vertices. *)
for k := 1 to (b + 1) do
  if (not counted_all) then
    begin
      degree := degree + 1;
test_edge := test_edge.next;
counted_all := (test_edge = first_edge)
    end;

(* Remove edges which are incident on high-degree vertices. *)
if degree ≤ b then
  begin
    Mark vertex low-degree;
    for j := 1 to b do if (edge j <> null) then Mark 'far end' of edge j high-degree;
    for j := 1 to b do if (edge j <> null) then Mark 'near end' of edge j low-degree;
    for j := 1 to b do if (edge j <> null) then
      if 'far end' of edge j is marked high-degree
        then remove edge j from edge list
  end.

Lemma 4.2.3: BD-FISP(n) ≤ O(I) + c L-FISP(n)

Proof: Assume we are given as input a graph G each of whose vertices has degree at most b. Each vertex is assigned a processor. We first show how the edge set E of the graph G can be decomposed, in constant time, into a constant number t < 2b of sets E_i (1 ≤ i ≤ t) where each set E_i defines a list graph.

The sets E_i (1 ≤ i ≤ t) will be formed in t rounds. A set of chains is identified in parallel by allowing each vertex to determine at most one incoming edge and at most one outgoing edge from its remaining incident edges. As an edge is chosen, it is marked as belonging to chain i and is removed from consideration. Each round is divided into b subrounds. In a subround, a
vertex which has not yet had a proposal accepted proposes to a new neighbour (if one is available). If a vertex receives any proposals, it will accept exactly one and ignore all others. A vertex has all of its proposals ignored in a round only if all of its neighbours have accepted other proposals in this round. Hence after $2b$ rounds, all of a vertex's neighbours must have accepted its proposal.

We then find a Fractional Independent set in $E_1$ and mark them as survivors. Among the set of survivors in $E_1$ we find a Fractional Independent set in $E_{i+1}$. The set of survivors in $E_t$ will form a fixed fraction of the vertices of $G$. Therefore a constant number of iterations of the Fractional Independent Set Problem for list graphs suffices to solve the Fractional Independent Set Problem for bounded degree graphs.

The following pseudo-code procedure, which is executed in parallel by each of the vertex processors, describes in more detail the procedure to decompose a degree bounded graph into a set of list graphs.
procedure DecomposeIntoChains;
begin

(* Each iteration identifies 1 outgoing edge and 1 incoming edge. *)

for i := 1 to t do
begin
    in_mated := false;
    out_mated := false;

    for j := 1 to b do
    begin
        (* Propose to a neighbour *)
        if (not out_mated) and (edge j <> null) then
            Mark 'Far End' of edge j propose;

        (* Check proposals from neighbours *)
        for k := 1 to b do
            if (not in_mated) and (edge k <> null) and
               ('near end' of edge k is marked propose) then
                begin
                    Mark 'near end' of edge k accept;
                    in_mated := true;
                    in_edge := k
                    end;

        (* See if proposal was accepted *)
        if (not out_mated) and (edge j <> null) and
           ('far end' of edge j is marked accept) then
             begin
                out_mated := true;
                out_edge := j
                end;

    (* Record incident edges for chain i *)
    if in_mated then
        begin
            Mark 'near end' of edge in_edge in-chain (i);
            Remove edge in_edge from current edge ring
        end;

    if out_mated then
        begin
            Mark 'near end' of edge out_edge out-chain (i);
            Remove edge out_edge from current edge ring
        end;

    end
end.
Note that the Fractional Independent Set Problem for \( n \) vertex list graphs can be solved by standard list ranking in \( O(\log n) \) parallel time using \( O(n) \) EREW processors [W79]. However, full list ranking is unnecessary.

**Lemma 4.2.4:** \( \text{L-FISP}(n) \) can be solved in \( O(\log^* n) \) parallel time using a deterministic algorithm.

**Proof:** Cole and Vishkin [CV86a][CV86b] define an \( r \)-ruling set on a list graph \( L \) on \( n \) vertices to be a subset \( U \) of the vertices of \( L \) such that: i) No two vertices of \( U \) are adjacent; and ii) For each vertex \( v \) in \( L \) there is a directed path from \( v \) to some vertex in \( U \) whose edge length is at most \( r \). Cole and Vishkin show how to find a 2-ruling set in \( O(\log^* n) \) time using a technique which they call deterministic coin tossing. As discussed in section 1.3, a 2-ruling set is a a 3-colouring and forms a maximal independent set for its list.

**Lemma 4.2.5:** \( \text{L-FISP}(n) \) can be solved with probability \( 1 - O(cn) \), for some \( c < 1 \), in \( O(1) \) parallel time using a randomized algorithm.

**Proof:** Assign a processor to each vertex of the list. Each processor flips a 0 or a 1 with equal probability. A vertex is chosen if its processor flips a 1 and either it has no successor or its successor flips a 0. With probability at least 1/4 an arbitrary vertex is chosen. However, these probabilities are not independent. Nevertheless, every second list element is chosen independently with a probability of at least 1/4. Thus applying the Chernoff bound (cf. [PB85, p. 464]), we find that the probability that fewer than 1/8 of the even positioned list elements are chosen is at most \( cn \), where \( c < 0.98 \).

**Theorem 4.2.6:** The Subdivision Hierarchy Construction Problem for a convex subdivision on \( n \) vertices can be solved in \( O(\log n) \) expected parallel time using a randomized algorithm or \( O(\log n \log^* n) \) parallel time using a deterministic algorithm.
Proof: The deterministic result is immediate from Lemmas 4.2.1, 4.2.2, 4.2.3, and 4.2.4. The randomized algorithm exploits Lemma 4.2.5 to find (and remove) low degree independent sets for $O(\log n)$ phases. At this point, the resulting subdivision has $O(\log n)$ vertices, with overwhelming probability, and $O(\log n)$ additional steps of the sequential algorithm suffice to complete the subdivision hierarchy. If this is not the case then the entire computation can be restarted and the expected time remains $O(\log n)$.

Corollary 4.2.7: $\text{SHCP}(n) \leq O(\log n)$

Once the subdivision hierarchy structure is constructed, the algorithm for subdivision search is identical to that presented in [K83]. The subdivision hierarchy technique uses $O(\log n \log^* n)$ parallel preprocessing, $O(n)$ space and $O(\log n)$ sequential query time.

Other researchers have used similar approaches to solve a variety of planar graph colouring and independent set problems. Hagerup, Chrobak and Diks [HCD89] describe an algorithm for computing a 5-colouring of a planar graph in $O(\log n \log^* n)$ time using $n$ EREW processors. This is accomplished by identifying the vertices of a planar graph which meet a specific reducibility criterion: they can be removed and subsequently replaced in such a way as to extend an existing 5-colouring. The subgraph induced on the reducible vertices of a planar graph forms a bounded degree graph. A large independent set of reducible vertices is constructed in $O(\log^* n)$ using a fractional independent set similar to that presented in this section (4.2).

As part of their construction, Hagerup et al outline requirements for applying the processor reduction technique of Cole and Vishkin [CV86a] based on Brent's scheduling principle [B74]. They use this to reduce their processor requirement to $n/(\log n \log^* n)$ EREW processors. We note that these apply to any $n$-processor algorithm consisting of $O(\log n)$ stages where i) each stage consists of some constant time computation plus a constant number of applications of
Cole and Vishkin's 2-ruling set algorithm; ii) the number of active processors in the ith stage is half the number of active processors in the (i-1)st stage and iii) once a processor becomes inactive it remains so. Since the deterministic version of our algorithm meets these requirements, it can be modified to produce an algorithm running in $O(\log n \log^* n)$ time employing $n/(\log n \log^* n)$ processors.

Goldberg, Plotkin and Shannon [GPS87] describe an $O(\log^* n)$ time algorithm using $O(n)$ EREW processors for colouring bounded degree graphs. By iterating through the colours, this algorithm can be used to find a maximal independent set in an $n$-vertex bounded degree graph in an additional $O(1)$ parallel time. They further describe how to 7-colour a planar graph in $O(\log n \log^* n)$ time using $n$ CRCW processors by decomposing it into a sequence of bounded degree graphs. They use this 7-colouring to find a maximal independent in an $n$-vertex planar graph. On an EREW model their 7-colouring algorithm may require $O(\log^2 n)$ parallel time.

Cole and Zajicek [CZ87] have addressed the problem of constructing subdivision hierarchies. They employ the Duration Unknown Task Scheduling technique (DUTS) to obtain an $O(\log n)$ time algorithm using $n/\log n$ processors. This is the same technique which was originally used to obtain the first list-ranking algorithm running in $O(\log n)$ time and using $n/\log n$ EREW processors [CV86b]. However, the DUTS technique uses expander graphs to perform its scheduling and hence has rather large constant factors.

As described in [DK90], the subdivision hierarchy can be used for 3-dimensional applications which seem to be beyond the scope of the other approaches to the point location problem (such as the parallel plane sweep technique or Tamassia and Vitter's data structure). We expand on this in the next section.
4.3 Polyhedral Applications

As in Dobkin and Kirkpatrick's presentations [DK82] [DK85] and [DK90], we exploit the fact that the surface of a convex polyhedron is isomorphic to some bounded planar subdivision and define an hierarchical representation for convex polytopes similar to the hierarchical representation for planar subdivisions:

For completeness, we repeat the definition of the geometric hierarchy as applied to convex polyhedra: There exist positive constants $\alpha$ and $\beta$ such that for any $n$ vertex convex polytope $P$ with vertex set $V(P)$, there is a sequence of convex polytopes $H(P) = P^1, \ldots, P^N$ (called a polyhedral hierarchy of $P$) satisfying:

i) $P^1 = P$;

ii) $|P^N|$ is bounded by a constant;

iii) $V(P^{i+1}) \subset V(P^i)$ and $|P^{i+1}| < |P^i|/\alpha$; and

iv) for any vertex $p_j \in P^i$ either $p_j \in P^{i+1}$ or degree$(p_j)$ in $P^i$ is less than $\beta$ and all of $p_j$'s neighbors in $P^i$ are in $P^{i+1}$.

The Doubly Connected Edge List can be used to implement the hierarchical representation of convex polyhedra in the same way as for Subdivision Hierarchies.

Corollary 4.3.1: Given a convex polyhedron on $n$ vertices, a hierarchical representation with $O(\log n)$ elements can be constructed in $O(\log n)$ expected parallel time using a randomized algorithm or $O(\log n \log^* n)$ parallel time using a deterministic algorithm.

Proof: This follows from Theorem 4.2.6 and the fact that the faces of a convex polyhedron can be triangulated in $O(\log n)$ time.
This hierarchical representation can be used to answer many different intersection and separation queries involving polyhedra. Convex polyhedron intersection and separation queries with points, lines and planes can be answered in \(O(\log n)\) sequential time [DK82] [DK83] [DK90].

A line query, as defined by Aggarwal \textit{et al} [ACGOY85], poses the following problem: Given a convex polyhedron \(P\) and a line \(L\) in 3-space, determine whether or not \(L\) intersects \(P\) and, if not, give the two planes through \(L\) tangent to \(P\).

\textbf{Lemma 4.3.2:} Given the hierarchical representation of a convex polyhedron \(H(P)\) and a line \(L\) in 3-space, a line query can be answered in \(O(\log n)\) sequential time.

\textbf{Proof:} Dobkin and Kirkpatrick [DK82] [DK83] [DK90] describe an \(O(\log n)\) sequential time algorithm for detecting the intersection of a line \(L\) and an hierarchically described polyhedron \(P\) and for constructing the intersection when it is non-empty (some of the implementation details are further described by Edelsbrunner and Maurer [EM85]). By a straightforward modification of the same techniques, it is possible to construct the tangent planes through \(L\) when the intersection is empty. \(\S\)

\textbf{Corollary 4.3.3:} Given the hierarchical representations of two separated convex polyhedra \(H(P)\) and \(H(Q)\), their convex union can be constructed in \(O(\log n)\) parallel time using \(n\) CREW processors.

\textbf{Proof:} By Lemma 4.3.2, given the subdivision hierarchy of a convex object \(O\) with \(n\) vertices and a line \(L\), the hierarchy can be used to answer a line query in \(O(\log n)\) time with a single processor. Thus, to construct the convex union in logarithmic time a processor is assigned to each edge in both \(P\) and \(Q\). The subdivision hierarchy is used to determine the two supporting planes of the opposite convex polyhedron through each edge. As a result each edge (and incident faces) can be classified as being in or out of the convex union. It remains to update the
edge rings of all vertices to reflect adjacencies on the convex union. The details, which involve list ranking and merging, are described by Aggarwal et al [ACGOY88].

**Corollary 4.3.4:** The 3-d convex hull of \( n \) vertices can be constructed in \( O(\log^2 n) \) expected parallel time using a randomized algorithm or \( O(\log^2 n \log^* n) \) parallel time using a deterministic algorithm employing \( n \) CREW processors.

**Proof:** The algorithm proceeds in a manner similar to the divide and conquer algorithm of Preparata and Hong [PH78] and Aggarwal et al [ACGOY88]. Using the parallel merge sort of Cole [C86b], the vertex set is lexicographically sorted in \( O(\log n) \) time using \( n \) processors. Then the vertex set is recursively divided into separable sets. The hull of each of the sets is found recursively and the Polyhedral Hierarchy is constructed for each. The separable Convex Union algorithm of Corollary 4.3.3 is used as the merge step. Constructing the subdivision hierarchy is the dominant cost of the recursive step.

The 3-d hull construction algorithm described in Aggarwal et al [ACGOY85] runs in \( O(\log^4 n) \) parallel time. This was improved to \( O(\log^3 n) \) parallel time in [ACGOY88]. The use of the hierarchical representation makes our approach significantly simpler as well as more efficient.

**Corollary 4.3.5:** The 3-d Convex Polyhedron Separation problem (determining the separation of two convex objects in \( \mathbb{R}^3 \)) can be solved in \( O(\log n) \) expected parallel time using a randomized algorithm or \( O(\log n \log^* n) \) parallel time using a deterministic algorithm.

**Proof:** The separation of two convex polyhedra will be realized by a vertex-vertex pair, an edge-vertex pair or a edge-edge pair. Once the Polyhedral Hierarchy for both convex objects is constructed, a processor is assigned to each vertex and edge. Each will determine its separation from the opposite convex polyhedron in \( O(\log n) \) time [DK90]. Then a minimum
operation can be performed to determine the polyhedral separation in \( O(\log n) \) time.

Constructing the polyhedral hierarchy is the dominant cost.

This is a very crude separation algorithm, in that its time-processor product is \( O(n \log n) \).

If the two polyhedra are presented with their polyhedral hierarchies, a single processor can determine their separation in \( O(\log^2 n) \) parallel time. The next chapter deals with more effective processor utilization in presenting cooperative algorithms for computing separation.

As a final observation, we note that the solution to the Fractional Independent Set Problem can be extended to find a Maximal Independent Set in an \( n \)-vertex planar graph in \( O(\log n \log* n) \) (respectively \( O(\log n) \) expected) parallel time using a deterministic (respectively randomized) algorithm on \( n \) EREW processors as described in [DaK87a] and [DaK87b]. The maximal independent set algorithm identifies and removes fractional independent sets in rounds in the same way as the subdivision hierarchy construction algorithm of section 4.2. In this way, the maximal independent set is accumulated from the fractional independent sets removed in each round.

A problem arises in updating high-degree vertices: When a vertex is identified as a member of a fractional independent set and it is added to the accumulated maximal independent set, its neighbours are no longer candidates for the maximal independent set. However, since some of those neighbours may be high-degree and we are assuming the EREW model, it is not possible to immediately mark those neighbouring vertices as ineligible for the maximal independent set. Rather, the edges incident on a removed vertex are marked and a phase for contracting the edge-rings of remaining vertices is added to each round. An induction argument [DaK87b] proves that \( O(\log n) \) rounds suffices to identify a maximal independent set confirming the result stated above. As noted earlier, problems within the maximal independent set algorithm which arise in updating high-degree vertices do not arise in the subdivision hierarchy construction algorithm because of the availability of an embedding.
Chapter 5: Cooperative Search in Planar Subdivisions

5.1 Introduction and Related Work

In this chapter, we show how to augment the standard subdivision hierarchy (as described in the last chapter) to enable cooperative algorithms to solve planar point location queries for n-vertex planar subdivisions in \( O(\log n / (1 + \log k)) \) time using \( k \) CREW processors (\( 1 \leq k \leq n \)). This augmentation is performed in such a way that the entire data structure only requires linear space. This is not a direct application of the geometric hierarchy representation (cf. section 1.5 and [K83]) but is an extension of that structure designed specifically for cooperative search.

Previous research into parallel algorithms for subdivision search has concentrated on parallel preprocessing of a planar subdivision to facilitate sequential subdivision search. In Chapter 4, we show that given a planar subdivision on \( n \) vertices, an \( O(n) \) space hierarchical representation (which we call an ordinary subdivision hierarchy) forming a sequence of \( O(\log n) \) subdivision elements can be constructed in \( O(\log n) \) expected parallel time using a randomized algorithm or \( O(\log n \log^* n) \) parallel time using a deterministic algorithm. We recall that the standard doubly connected edge list representation is used to implement each of the subdivisions in the ordinary hierarchy with additional pointers used to 'bridge' consecutive subdivisions. As discussed in Chapter 4 and elsewhere [DK82] [DK83] [DK90] [DaK87a] [DaK89a], the ordinary hierarchy when interpreted as a polyhedral hierarchy can be used to answer many different intersection and separation queries involving polyhedra.

Exploiting a subdivision hierarchy for cooperative search algorithms is not as simple as the k-ary search algorithms applied to linear subdivision search or, equivalently, the use of
accelerated polygonal hierarchies in 2 dimensions (cf. chapter 2). An accelerated subdivision hierarchy is not implicit within the representation; the hierarchy must be explicitly constructed within a preprocessing phase. Additional pointer information must be incorporated to allow "jumping through" levels of the hierarchy, applying parallelism to accelerate stepping through the hierarchy; an accelerated hierarchy is available within the 2-dimensional structure simply by using indexing. Localizing the search within the hierarchy by finding descendants of a particular element is more complicated; this could be done within the linear subdivision (or polygonal hierarchy) by limiting the index range under consideration. However, it is possible to apply the geometric hierarchy paradigm to the design and use of a data structure which can be used for cooperative subdivision search queries.

5.2 A Static Subdivision Hierarchy

The first issue to address is the existence of a space-efficient static hierarchy structure – one which will admit to $O(\log n/(1 + \log k))$-time $k$ CREW processor point location for a fixed value of $k$ such that $1 \leq k \leq n$ and only requires linear space. It is worth noting that attempting to construct a static hierarchy in the obvious way – by collapsing multiple levels of the ordinary hierarchy – will introduce an unacceptable space cost. The accumulated pointer overhead becomes the dominant space cost. According to the subdivision hierarchy definition (cf. section 4.2), the number of vertices (hence the size of the subdivision) of element $S^i$ decreases by a factor of $\alpha$ to produce $S^{i+1}$ but each facet of $S^i$ may have $\beta$ descendants. In collapsing $d$ levels of the hierarchy, the size of the subdivision decreases by a factor of $\alpha^d$ but each facet may have $\beta^d$ descendants. Therefore the space requirement for $S^{i+d}$ is given by

$$|S^{i+d}| \leq \frac{|S^i|}{\alpha^d} + \frac{|S^i| \beta^d}{\alpha^d}$$
where the first term is the size of the subdivision itself and the second term is the associated pointer cost. Since $\beta$ is typically much larger than $\alpha$, the geometric sequence argument used to obtain the $O(n)$ space bound in the ordinary hierarchy no longer applies. As suggested by Edelsbrunner et al [EGS86], a directed acyclic graph structure seems to be necessary to combine an efficient search structure with efficient space utilization. Therefore, it is necessary to use a strategy which is more sophisticated than simply collapsing levels and composing pointers – a scheme which allows 'pointer-sharing' between levels to reduce the space cost.

An $f(n)$-separator theorem for a class of graphs $\mathcal{S}$ is a theorem which states that for any $n$-vertex graph $G$ in $\mathcal{S}$, there exist constants $\sigma < 1$ and $\delta > 0$ such that it is possible to partition $G$ into sets $A$, $B$ and $C$ such that no edge joins a vertex of $A$ with a vertex of $B$, $|A| \leq \sigma n$, $|B| \leq \sigma n$, and $|C| \leq \delta f(n)$. The existence of a separator theorem for a class of graphs can be useful in the design of divide-and-conquer algorithms for that class. There has been a large body of work on algorithms to identify planar separators for sequential and parallel models of computation [LT79] [DJ82] [M86] [GM87]. Lipton and Tarjan provide an algorithm for the following separator theorem for planar graphs:

**Theorem 5.2.1** [LT79]: Let $G$ be any $n$-vertex planar graph with non-negative vertex costs summing to no more than one. Then the vertices of $G$ can be partitioned into three sets $A$, $B$, $C$, such that no edge joins a vertex in $A$ with a vertex in $B$, neither $A$ nor $B$ has total vertex cost exceeding $\frac{2}{3}$, and $C$ contains no more than $\sqrt{8n}$ vertices. Furthermore, $A$, $B$, $C$ can be found in $O(n)$ time.

In a follow-up paper [LT77] [LT80], Lipton and Tarjan present various algorithms for solving problems on planar graphs which made use of a planar separator theorem. One of
these was a complicated construction for preprocessing an n-vertex planar subdivision in $O(n)$ time utilizing $O(n)$ space to answer point location queries in $O(\log n)$ time\(^1\).

Lipton and Tarjan's approximation algorithm for the maximum independent set problem makes use of the fact that a planar separator algorithm can be applied recursively to further decompose a planar graph. We recall the statement of Lipton and Tarjan's \textit{iterated planar separator theorem}:

\textbf{Theorem 5.2.2} [LT80]: Let $G$ be an $n$-vertex planar graph with nonnegative vertex costs summing to no more than one. Given $\epsilon$ such that $0 < \epsilon \leq 1$, there is some set $C$ of $O(\sqrt{n/\epsilon})$ vertices whose removal leaves $G$ with no connected component of cost exceeding $\epsilon$. Furthermore, the set $C$ can be identified in $O(n \log n)$ time.

As described by Lipton and Tarjan [LT80], this set $C$ is identified through iterated application of their planar separator theorem [LT79]. The parallel simple cycle separator algorithm due to Miller [M86] can be employed to identify an equivalent set $C$ in polylogarithmic time using a polynomial number of processors.

Iterated application of a planar separator theorem can be used to construct a space-efficient subdivision hierarchy for cooperative point location by $k$ processors. This is achieved by employing the iterated planar separator algorithm to decompose a planar subdivision into isolated components each of which can be searched by $k$ processors in constant time.

Therefore, in this \textit{static hierarchy}, the value of $k$ is built directly into the data structure.

---

\(^1\) The point location application was discussed in the original conference paper and was not addressed in the subsequent journal version [LT80].
**Theorem 5.2.3:** Given a planar subdivision $G$ on $n$ vertices and an integer $k$ for $1 \leq k \leq n$, it is possible to construct an $O(n)$-space data structure which will allow $k$ processors to perform a point-location query in $O(\log n / (1 + \log k))$ parallel time.

**Proof:** We assume the subdivision is triangulated. By Theorem 5.2.2, for a vertex cost assignment summing to no more than one and a given $\varepsilon$ such that $0 \geq \varepsilon \geq 1$, a small separator $C$ can be identified whose removal will leave no component of size exceeding $\varepsilon$. The key to using an iterated planar separator algorithm to preprocess a planar subdivision for cooperative search by $k$ processors is to relate $k$ to $\varepsilon$, the size of the isolated components.

This can be used within the geometric hierarchy paradigm to construct a sequence of subdivisions which can be used for cooperative point location by $k$ processors. The first element of the sequence is the initial triangulated subdivision. Each subsequent element (subdivision) of the sequence is derived from its predecessor by identifying a separator which decomposes the subdivision into isolated components, removing those components and retriangulating the separator. This continues until the final element of the sequence is small enough to be searched by $k$ processors in constant time.

This approach can be used to construct a data structure with the desired query time and storage costs with an appropriate non-negative vertex cost assignment, and an appropriate choice of $\varepsilon$.

It is necessary to control the size of the triangulation which results from the removal of a component to ensure that any of the isolated components can be searched by $k$ processors in constant time. This requires relating the weight of a component to the "size of the hole" remaining when it is removed. For a planar graph on $n$ vertices, Euler's formula implies that there are at most $(3n - 6)$ edges [BM76] and that the sum of the degrees of all vertices is
(6n − 12). By assigning each vertex v the weight \( \frac{\text{degree}(v)}{(6n - 12)} \), the sum of all vertex weights is equal to one and we can guarantee that any component of weight \( \frac{k}{n} \) has at most \( k \) edges incident on the boundary of the component. Therefore, when this component is removed the resulting retriangulation will have at most \( k \) faces. Using this cost assignment, a component of weight \( \frac{k}{n} \) might be realized by anything between a single vertex of degree \( k \) and \( O(k) \) vertices of degree bounded by a constant.

The static hierarchy is customized for a given \( k \), therefore it is necessary to address the structure of the hierarchy for small values of \( k \) by refining the definition of the component weight \( \epsilon \). Appealing to Theorem 5.2.2, if \( \epsilon \leq \frac{1}{n} \), \(|C| = n\) and the theorem holds trivially but no components can be removed. Since \(|C| = O(\sqrt{n/\epsilon})\), there exist constants \( c \) and \( n_0 \) such that for all \( n \geq n_0 \), \(|C| \leq c\sqrt{n/\epsilon}\). We are interested in values of \( \epsilon \) for which \( c\sqrt{n/\epsilon} < n \). Therefore, \( \epsilon > \frac{c^2}{n} \). (The construction for Lipton and Tarjan's original separator theorem [LT79] gives \( c < 16 \); Venkatesan [Ve87] improves this to \( c < 5 \).)

Therefore, Theorem 5.2.2 can be used to construct a static hierarchy for \( k \) such that \( 1 \leq k \leq n \) by choosing

\[
\epsilon = \begin{cases} 
\frac{c^2}{n} & \text{if } 1 \leq k \leq c^2 \\
\frac{k}{n} & \text{if } c^2 < k \leq n.
\end{cases}
\]

For a small value of \( k \), each component has constant size and can be searched by a single processor in constant time. Therefore for all \( k \) \( (1 \leq k \leq n) \), \( k \) processors can search any component of weight \( \epsilon \) in constant time.
Using the algorithm associated with Theorem 5.2.2 on G with the given weight scheme on the vertices and the choice of ε specified above, it is possible to identify a separating set C of size \( O(n/\sqrt{k}) \) such that the remaining connected components have size \( O(k) \). Removing the connected components and retriangulating produces a new planar subdivision \( G_1 \) of size \( O(n/\sqrt{k}) \).

This entire process can be repeated to produce a sequence of m subdivisions, \( G = G_0, G_1 \) to \( G_m \) such that the size of \( G_m \) is \( O(k) \). Since the size of \( G_0 \) is \( n \) and each subdivision is a \( \sqrt{k} \) factor smaller than its predecessor, \( m \) is \( \log_{\sqrt{k}}(n) \) or \( O(\log n / (1 + \log k)) \).

Consider the space requirement of this static hierarchy. Each facet f which is removed from subdivision \( G_i \) keeps one pointer to the list of facets in subdivision \( G_{i+1} \) which resulted when f's component was removed and retriangulated. Thus, each component of size \( O(k) \) which is removed within a subdivision \( G_i \) keeps one list of the faces in its corresponding retriangulation in \( G_{i+1} \). This "sharing of pointers" from level to level only requires a linear number of pointers to be associated with a given level (see figure 5.1). Since the size of the levels is geometrically decreasing, the overall storage is linear. We note in passing that the data structure implementing the 'bridge' between the retriangulated component in \( G_{i+1} \) and its corresponding component in \( G_i \) must be able to support \( k \) processor allocation in constant time; an array of pointers to the (triangular) faces of the removed component is sufficient.

The associated search algorithm is a simple extension of the search algorithm associated with the ordinary subdivision hierarchy. The search begins at the last level \( G_m \). Within an iteration, the \( k \) processors search the \( O(k) \) target faces in the current level \( G_i \) in constant time. A unique "winner" is determined which writes its index into a predetermined location. The \( k \) processors use the concurrent read facility to determine which processor was successful and
reallocate themselves to the associated sequence of $O(k)$ faces in level $G_{i-1}$. This continues until the query point is located in $G_0 = G$. A constant amount of time is required for each level, and therefore using $k$ CREW processors, the overall query time is directly proportional to the number of levels, $O(\log n / (1 + \log k))$. §
The static hierarchy constructed using the algorithm associated with Theorem 5.2.2 can be considered a straightforward generalization of the ordinary subdivision hierarchy. In the ordinary hierarchy, the removable component weights are held to be constant and the components to be removed are restricted to being single vertices. The fact that an independent set of low-degree vertices which forms a fixed fraction of the size of the subdivision can be identified quickly is used in place of the (more complicated) separator theorem. There is an interesting dichotomy between the two hierarchies: the ordinary hierarchy is built bottom up by identifying the components to be removed; the static hierarchy is built top-down by identifying the vertices in the separator. Note that this is a simpler application of Lipton and Tarjan's separator result to subdivision search than that originally presented in [LT77].

5.3 A Dynamic Subdivision Hierarchy

The next issue to address is the existence of a space-efficient dynamic hierarchy structure—one which will admit to $O(\log n/(1 + \log k))$-time $k$-processor point location for all values of $k$ such that $1 \leq k \leq n$ and only requires linear space.

One way of building a (space-inefficient) structure is to simply construct separate static hierarchies for $2^i$ processors ($0 \leq i \leq \log(n)$). This structure consists of $O(\log n)$ static hierarchies, each of which requires $O(n)$ space, resulting in an $O(n \log n)$ space requirement overall. Given a value of $k$ in the valid range, one of the available static hierarchies will be within a constant factor (2) of $k$.

\footnote{For simplicity in this discussion, we ignore problems in dealing with small values of $k$.}
This first attempt at a dynamic hierarchy can be improved by noting that the speed-up provided by a static hierarchy is in \( \log(k) \). Therefore, it would suffice to construct separate static hierarchies for \( 2^{2^i} \) processors \((1 \leq i \leq \log\log(n))\) resulting in an \( O(n \log \log n) \)-space structure. Given a value of \( k \) in the valid range, it is possible to use the static hierarchy for the largest \( 2^{2^i} \leq k \) and still achieve \( O(\log n/(1 + \log k)) \)-time point location employing \( k \) CREW processors.

In both of the attempts at a dynamic hierarchy above, the space cost of each of the component static hierarchies is dominated by its largest subdivision; in all cases above, the original subdivision. A space-efficient structure can be constructed by using static hierarchies to augment an ordinary hierarchy. In this scheme, the static hierarchies are 'rooted' at different subdivisions within the ordinary hierarchy:

**Theorem 5.3.1:** Given a planar subdivision \( G \) on \( n \) vertices, it is possible to construct an \( O(n) \)-space data structure which will allow \( k \) processors to perform a point-location query in \( O(\log n / (1 + \log k)) \) parallel time for any integer \( 1 \leq k \leq n \).

**Proof:** The *dynamic augmented hierarchy* is an ordinary subdivision hierarchy which has been augmented to allow different entry points for different specific values of \( k \). The larger the value of \( k \), the 'higher' the entry point within the ordinary hierarchy. Recall that an ordinary hierarchy for a planar subdivision \( G \) on \( n \) vertices is a sequence of \( w \) subdivisions, \( G = G_0, G_1 \) to \( G_w \) such that for positive constants \( \alpha > 1 \) and \( \beta > 1 \), (i) \( |G_{i+1}| = |G_i| / \alpha \); (ii) \( |G_w| \) is \( O(1) \) and therefore \( w = \log_\alpha n \), and (iii) each region of \( G_{i+1} \) overlaps at most \( \beta \) regions in \( G_i \).

For simplicity, we assume that \( n = 2^{2^t} \) (and therefore, \( t = \log\log n \)) for some integer \( t \). We assume that we have an ordinary subdivision hierarchy constructed for \( G \) as described above.
We augment each of the elements $G_{2^t}$ for $1 \leq i \leq \log \log n$ with a static hierarchy attuned to $2^{2^i}$ processors (see figure 5.2).

Figure 5.2  A dynamic hierarchy

Consider performing a point location query with $k$ processors such that $1 \leq k \leq n$. We can assume that $k = 2^{2^i}$ for $1 \leq i \leq \log \log n$, otherwise as described above we can use the
largest $2^i \leq k$. Our structure should enable $k$ processors to answer a point location query in $G$ in $O(\log n / (1 + \log k)) = O(2^{t-i})$ steps. Therefore, we use the $k$ processors to perform a cooperative search query using the static hierarchy associated with element $G_{2^i}$ in $O(2^{t-i})$ steps. Given the query point's location in $G_{2^i}$, a single processor can employ the ordinary hierarchy to identify the face containing the query point in another $2^{t-i}$ steps, each of which require constant time. Therefore, this dynamic hierarchy enables $k$ processors to perform a point location query in $G$ in $O(\log n / (1 + \log k))$ time.

We show that this dynamic hierarchy uses $O(n)$ space. The ordinary hierarchy only requires $O(n)$ space. Denote the size of the augmented structure associated with ordinary hierarchy element $G_{2^i}$ by $S_{2^i}$. Let $m = |G_{2^i}| = n / \alpha^{2^i}$ for $\alpha > 1$ such that $\alpha$ is the reduction factor associated with the ordinary hierarchy. Therefore,

$$S_{2^i} \leq m + \frac{m}{\sqrt{k}} + \frac{m}{(\sqrt{k})^2} + \frac{m}{(\sqrt{k})^3} + \ldots$$

$$= \frac{m}{1 - \frac{1}{\sqrt{k}}}$$

$$= \frac{m\sqrt{k}}{\sqrt{k} - 1}$$

$$= \frac{m}{\alpha^{2^i}} \left( \frac{2^{2^i-1}}{2^{2^i-1} - 1} \right)$$

$$\leq \frac{2n}{\alpha^{2^i}}.$$

Therefore, the total size contribution of the augmented static hierarchies forms a constant multiple of a subsequence of the ordinary hierarchy's geometric sequence. Therefore, the overall space requirement for the dynamic augmented hierarchy is $O(n)$. \(\S\)
Note that if $\alpha$ were greater than or equal to 2 then it would not be necessary to augment the ordinary hierarchy; the space reduction from level to level would be sufficient to allow $k$ processors to search the ordinary hierarchy element $G_{2^{i-1}}$ directly in constant time. Typically, $\alpha$ is much closer to 1, necessitating the use of the augmenting static hierarchies.

With regard to construction times for an $n$-vertex subdivision, the ordinary hierarchy can be constructed sequentially in $O(n)$ time [K83] and in parallel in $O(\log n)$ time using $O(n/\log n)$ processors with an expander graph construction [CZ87] or in $O(\log n \log^*n)$ time using $O(n/(\log n \log^*n))$ processors by identifying fractional independent sets (cf. Chapter 4 & [DaK89a]).

The static augmented hierarchy (for a given $k$) can be constructed sequentially in $O(n \log n)$ time [LT77] [LT80] [Ve87] or in parallel in $O(\log^2n)$ time using $n^3$ processors employing iterated application of Miller's separator algorithm [M86]. The dynamic augmented hierarchy incorporates an ordinary hierarchy and a geometrically decreasing sequence of static hierarchies resulting in overall $O(n \log n)$ sequential time construction or $O(\log^2n)$ parallel time construction using $n^3$ processors. The initial triangulation and subsequent retriangulations can be performed within these resource bounds [G85].

Building on the results described in this chapter (as reported in [DaK89c]), Tamassia and Vitter [TamV90] demonstrate how to augment their fractional cascaded data structure to support cooperative search. Again, the polyhedral applications discussed in the next section seem to be beyond the scope of their technique.
5.4 Polyhedral Separation Applications

As in [DK85] [DK90] [DaK87a] and [DaK89a], we exploit the fact that the surface of a convex polyhedron is isomorphic to some planar subdivision and define an hierarchical representation for convex polytopes similar to the hierarchical representation for planar subdivisions. With the augmented polyhedral hierarchy in place, cooperative polyhedron separation algorithms follow from their sequential equivalents in [DK85] and [DK90] and the cooperative search algorithm outlined in Section 5.1.

Theorem 5.4.1: Given an n-vertex convex polyhedron P which has been preprocessed into a dynamic augmented polyhedral hierarchy and a direction d, the extremum of P in direction d can be determined in $O(\log n/(1 + \log k))$ time using $k$ processors.

Theorem 5.4.2: Given a linear subspace L and an n-vertex convex polyhedron P which has been preprocessed into a dynamic augmented polyhedral hierarchy, the separation of P and L can be determined in $O(\log n/(1 + \log k))$ time using $k$ processors for $1 \leq k \leq n$.

Theorem 5.4.3: Given two n-vertex convex polyhedra P and Q which have been preprocessed into dynamic augmented polyhedral hierarchies, the separation of P and Q can be determined in $O((\log n/(1 + \log k))^2)$ time using $k$ processors for $1 \leq k \leq n$.

The polyhedron extremum and polyhedron/linear subspace separation results are straightforward generalizations of the point location algorithms. As an implementation note, it is necessary that the data structures used to represent each polyhedral hierarchy element and the pointers between elements support constant time processor allocation. As described in the proof of Theorem 5.2.3, an array of pointers to the component faces suffices for the subdivision application. For the polyhedral application, an array of pointers to the component
faces, edges and vertices are sufficient to implement a parallel generalization of the scheme described by Edelsbrunner and Maurer [EM85] for sequential extrema queries.

The polyhedron/polyhedron separation result uses the polyhedron/linear subspace algorithm of Theorem 5.4.2 as a subroutine. A single processor is used to initialize the parallel planes of support which delimit the separation of the smallest (constant size) element of each of P's and Q's k-static hierarchy. The algorithm proceeds by taking a step in one of the hierarchies (say P's) and using Q's hierarchy to update the separation by maintaining the parallel planes supporting the current element of each of P and Q's hierarchy. This continues until the top of both hierarchies has been reached and the separation of P and Q has been determined.
Chapter 6: Conclusion

6.1 Summary

This thesis has demonstrated that the geometric hierarchy paradigm, which has been useful in the design of many sequential geometry search algorithms, can be applied to the solution of geometric problems on a parallel model of computation. We have shown that given standard representations of convex polygons, planar subdivisions and convex polytopes, it is possible to construct and exploit geometric hierarchies to answer efficiently a variety of geometric search and separation queries.

It is interesting to contrast and compare the geometric hierarchy paradigm with competing approaches for subdivision search problems. In particular, the unique contribution of the geometric hierarchy is its applicability to polyhedral separation problems. The geometric hierarchy paradigm is formulated on topological properties of planar graphs rather than geometric properties of planar embeddings. This permits its application to problems formulated on topological constraints such as the 'establishing order in planar subdivisions' problem as posed by Kirkpatrick [K88].

Other schemes for planar subdivision search (cf. section 4.1) (notably that proposed by Edelsbrunner, Guibas and Stolfi [EGS86]) are based on the exploitation of geometric properties of planar graph embeddings such as discrimination of a query point against monotone chains or x-coordinates. This allows the Edelsbrunner et al scheme to deal with curved (but monotone) edges uniformly (depending only on an efficient method for curved segment discrimination) but seems to prohibit its application to polyhedral separation problems. The scheme recently proposed by Seidel [S90] (outlined in section 4.1), although based on a
geometric hierarchy approach, is also formulated on geometric/visibility criteria and does not seem to be applicable to polyhedra separation problems.

6.2 Directions for further work

There are many avenues for further research regarding geometric hierarchies, for both sequential and parallel models of computation. In the parallel setting, in particular, are the general questions of improving the preprocessing resource bounds, further examination of cooperative search algorithms and, in settings where a geometric hierarchy is used as a working representation (cf. the 3-dimensional convex hull construction algorithm of section 4.3) maintenance of this representation through iterations of the algorithm. We conclude with some open questions arising from the research presented in this thesis.

The lower time bounds for the polygon-vertex separation/mutual tangent problems proven in chapter 2 are \( \Omega(\log n/(1+ \log k)) \) for \( k \) CREW processors and \( \Omega(\log n - \log k) \) for \( k \) EREW processors. The CREW algorithms presented in chapter 2 demonstrate that polygon-polygon separation/mutual tangents can be computed within these bounds. In section 2.6, we sketch an algorithm (a natural extension of an EREW \( k \)-ary search in a sorted table algorithm) which computes polygon-vertex separation/mutual tangents in \( O(\log n - \log k) \) time with \( k \) EREW processors. Is there a cooperative \( k \)-processor EREW polygon-polygon separation algorithm which matches the \( \Omega(\log n - \log k) \) lower bound demonstrated by the reduction to sorted table lookup?

Can the tree contraction sequence be augmented efficiently for cooperative search within region trees? The Voronoi Diagram construction algorithm was originally developed using \( O(n) \) processors. Is a processor reduction possible? Is an improvement possible using cooperative search? When recursively combining the Voronoi Diagrams across the merge
contours, all of the region tree information is discarded; can the region tree be maintained efficiently through iterations to effect some asymptotic improvement?

Aggarwal, Guibas, Saxe and Shor [AGSS87] have described a linear sequential algorithm for computing the Voronoi diagram of a set of points presented as the vertices of a convex polygon. Can tree contraction be applied in developing a parallel version of this algorithm?

Kirkpatrick [K88] defines the problem of converting an end-point embedding of a planar subdivision (specified by a list of vertex locations together with an unordered list of edges given as vertex pairs) to one of the standard ordered representations (such as the DCEL) as the problem of *establishing order in a planar subdivision*. The algorithm presented by Kirkpatrick for solving this problem has the same batched vertex removal design as the sequential hierarchy construction algorithm [K83]. Can the fractional independent set technique presented in this section be applied to obtain a parallel version of that "establishing order" algorithm?

Can the polyhedral hierarchies be maintained through applications of the convex union algorithm to effect an asymptotic improvement in the 3-D convex hull construction algorithm? Can the static and/or dynamic hierarchies be constructed any more efficiently by utilizing bottom-up planar separator techniques, randomized algorithms or other tools?
References


[DaK89b] N. Dadoun and D.G. Kirkpatrick, "Optimal parallel algorithms for convex polygon separation", Computer Science Department TR 89-21, University of British Columbia, Sept. 1989, also submitted for publication.


