Parallel Voronoi Diagrams for VLSI design

by

Ryan Taylor

A thesis submitted to

the Faculty of Graduate Studies and Research

in partial fulfillment of

the requirements for the degree of

MASTER OF COMPUTER SCIENCE

School of Computer Science

at

Carleton University

Ottawa, Ontario

January, 2006

Your file Votre référence
ISBN: 0-494-13460-7
Our file Notre référence
ISBN: 0-494-13460-7

Abstract

The Voronoi diagram and its variants are widely used and studied in Computational Geometry. In particular, the Hausdorff Voronoi diagram has interesting applications in analysing the geometry of circuits in VLSI manufacturing. Due to the large size of VLSI circuits, it is natural to study these problems in a parallel setting.

The CGM parallel computing model has proven to be a useful model for the design of practical parallel algorithms. We present a novel parallel CGM algorithm to construct Hausdorff Voronoi diagrams. We also discuss an initial implementation of this algorithm. It turns out that our results present an improved sequential algorithm for *non-crossing* input sets.

Acknowledgements

I am grateful for the encouragement and support of my parents, Don and Sandra, my sister, Krista, and especially my fiancée, Laura. It has been because of them that this thesis was even possible.

Also, I remain indebted to Dr. Frank Dehne and Dr. Anil Maheshwari for their financial and academic support. Thanks are also due to the thesis committee members for their valuable feedback.

Contents

1 Introduction **1**

1.1 Organization of Thesis 5

2 Computational Geometry and Parallel Computing Review **7**

2.1 Voronoi Diagrams of Points 7

2.1.1 Overview 7

2.1.2 Definition and Properties 8

2.1.3 Sweepline Algorithm 11

2.1.4 Divide and Conquer Algorithm 13

2.2 Parallel Computing 16

2.2.1 Overview 16

2.2.2 PRAM Model 17

2.2.3 CGM Model 18

2.2.4 Analysing CGM Algorithms 20

2.2.5 Basic CGM Algorithms 21

2.3 A CGM Algorithm for Voronoi Diagrams of Points 27

2.3.1 Voronoi Diagram CGM Algorithm 27

2.3.2 Voronoi Diagram CGM Merge Algorithm 28

2.3.3 Analysis 31

2.4 CGM Segment Tree Algorithms 32
2.4.1 Parallel Next Element Search 32
2.4.2 Parallel Red-Blue Line Segment Intersection 36

3 Hausdorff Voronoi Diagram Review 47

3.1 Definition and Properties 47
3.2 Sweepline Algorithm 56
3.2.1 Overview 56
3.2.2 Algorithm 61
3.2.3 Analysis 63
3.3 Divide and Conquer Algorithm 65
3.3.1 Algorithm 65
3.3.2 Correctness 67
3.3.3 Analysis 73
3.4 VLSI Applications 74
3.4.1 Critical Area Problem 75
3.4.2 Shorts 76
3.4.3 Breaks 77
3.4.4 Via Blocks 78

4 CGM Algorithm for Hausdorff Voronoi Diagrams 79

4.1 Introduction 79
4.2 Algorithm Overview 80
4.3 Point Location 84
4.4 Red-Blue Line Intersection 87

4.5 Creating the Merge Chains 91
4.6 Analysis 92
4.7 Extensions and Evaluation 93
4.7.1 Improved Sequential Algorithm 93
4.7.2 Speedup 94

5 Implementation and Performance Results 97

5.1 Overview 97
5.2 Code Design 98
5.2.1 Existing Libraries 98
5.2.2 Voronoi Diagram Data Structure 99
5.2.3 Major Modules 100
5.2.4 Robustness Issues 101
5.2.5 Input Generation 102
5.3 Results 104
5.4 Interpretation 104

6 Conclusion and Future Work 113

List of Tables

5.1 Percent of total running time in each merge sub-algorithm, P=16 112

List of Figures

2.1 An example Voronoi diagram of Points 8

2.2 Sweepline Algorithm 12

2.3 Merging two subdiagrams 15

2.4 CGM Architecture 18

2.5 Example Distributed Segment Tree 34

2.6 Example Hereditary Segment Tree 38

2.7 Example Secondary Tree 39

2.8 Extension to the Hereditary Segment Tree Data Structure. 42

3.1 An example Hausdorff Voronoi diagram 48

3.2 Region is Essentially Star Shaped 52

3.3 Limiting Shape 53

3.4 Proof of Rear Limiting Shape 54

3.5 Handling a valid minimum priority event 59

3.6 Handling a valid furthest vertex event 60

3.7 Example Hausdorff hull 71

3.8 Summary of types of defects occurring a VLSI layer 76

4.1 An edge cannot cross the merge chain more than twice. 85

4.2 A Left edge with two close endpoints cannot contain crossings. . . 86

4.3 Merge vertices on edges in E_1^l may be found by Binary Search . . 89

4.4 Merge vertices on edges in E_2^l may be found by Binary Search . . 89

4.5 Sorting to find Merge Chain edges 91

4.6 Example merge of two Hausdorff sub-diagrams 96

5.1 Small example of possible Hausdorff input 103

5.2 Running Time of (Sequential) H-Vor Sweepline 105

5.3 Parallel Speedup fixed N 106

5.4 Parallel Speedup (fixed P) 107

5.5 Communication Overhead (16 procs) 108

5.6 Communication Overhead (1/3) 109

5.7 Communication Overhead (2/3) 110

5.8 Communication Overhead (3/3) 111

List of Algorithms

1 SWEEPLINE() 12

2 SITE_EVENT() 13

3 SPIKE_EVENT() 13

4 DIVIDE-AND-CONQUER(S) 14

5 MERGE($\mathcal{L}, \mathcal{R}$) 15

6 CGM_SORT() 22

7 GROUPED_SORT() 24

8 LOADBAL() 25

9 CGM_BINSEARCH() 26

10 CGM-VORONOI() 28

11 CGM-MERGE() 30

12 PNTLOC_SEGTREE_CONSTRUCT() 35

13 PNTLOC_SEGTREE_QUERY() 35

14 RB_SEGTREE_SEQUENTIAL() 41

15 RB_SEGTREE_BUILD() 44

16 RB_SEGTREE_SEARCH() 45

17 HAUSDORFF_SWEEPLINE() 61

18 INVALID_SITE() 62

19 HANDLE_MIN_PRIORITY() 62

20 HANDLE_MIXED_VERTEX() 63

21 HANDLE_FURTHEST_VERTEX() 63

22 HVOR-DIV-AND-CONQ() 65

23 HVOR-MERGE() 67
24 Merge two Hausdorff hulls 71
25 CGM-HAUSDORFF() 81
26 HAUSDORFF-CGM-MERGE() 82
27 HVOR-POINTLOC() 84
28 HVOR-RBINTERSECT() 87
29 HVOR-NOCROSS-MERGE() 93

Chapter 1

Introduction

We study the parallelisation of Hausdorff Voronoi diagram construction algorithms. The Hausdorff Voronoi diagram is a generalisation of the Euclidean Voronoi diagram of points, where sites are generalised to sets of points. The construction of Hausdorff Voronoi diagrams has gained recent attention due to its applicability to the computation of *Critical Area*, which is an important measure in VLSI circuit manufacturing.

Parallel Computing is the study of algorithms and architectures for computers with concurrent processing. Within this field, we focus on parallel algorithm design and analysis using the Coarse-Grained Multicomputer (CGM) model. This model of parallel computing is not restricted to a particular parallel architecture. Rather, the CGM model allows any architecture where the problem size is significantly larger than the number of processors. Algorithms designed using the CGM model can lead to implementations which perform well compared to sequential versions on existing parallel hardware.

Computational geometry is the study of the design and analysis of algorithms for geometric problems. The field contains a rich, diverse set of problems such

as line intersection, convex hull construction, point location, and triangulation. Computational geometry also has many practical application areas, such as geographic information systems (GIS), very-large scale integrated (VLSI) circuits, computer-aided design (CAD), computer graphics, and robotics.

One widely studied data structure in computational geometry is the Voronoi diagram. In its canonical form, the Voronoi diagram is constructed for a set of input points called sites. The plane is partitioned into a region for each site. Each region defines the area closest to its associated site. Thus, the Voronoi diagram, among many applications, presents a way to quickly determine an object's closest site by determining the region that the object is in. A sweepline algorithm and a divide-and-conquer algorithm are among the methods used to compute Voronoi diagrams.

The standard Voronoi diagram of points has been extensively studied, and the diagram has been generalized in a variety of ways. For example, Voronoi diagrams of higher dimensions provide interesting problems. Also, one may look at a variety of metrics other than the Euclidean metric, such as the more general class of convex metrics. Examples of other metrics are the L_∞ and L_1 metrics. Existing Voronoi literature also studies such varieties as additive and multiplicative metrics. Research into generalized Voronoi diagrams has attempted to abstract from specific metrics to algorithms that construct abstract Voronoi diagrams. A third way to generalize Voronoi diagrams is to discuss different types of input sites. For example, rather than just single points, a site may be generalized to comprise a set of points. Such sites may be, for example, line segments.

We study the Hausdorff Voronoi diagram, which is a variety of Voronoi diagram

that generalises the input sites to point sets in the plane. The distance metric is still essentially Euclidean. However, the distance measured to a site is the distance to the farthest point in the site. Hence, a Hausdorff Voronoi diagram may be considered as a Voronoi diagram of covering circles. The Voronoi diagram determines the distance which would create a circle which completely covers at least the closest input set. As the Hausdorff Voronoi diagram is a generalisation of the Voronoi diagram of points, the methods for computing Hausdorff Voronoi diagrams likewise tend to be generalisations of algorithms for Voronoi diagrams of points.

The Hausdorff Voronoi diagram has an interesting application in VLSI manufacturing. Engineering a new VLSI circuit layout is an expensive and time-consuming task. Part of the design process for new chips is to determine how resilient the chip's circuit geometry will be to defects in the manufacturing process. Rather than perform an experimental test production of a new VLSI design, it is faster and cheaper to use reliable analytical methods to predict chip yield. Yield is the fraction of chips produced without faults. A defect may occur on a chip without causing a faulty chip. One type of fault is when a specific component on the chip, a contact on the via layer, is disconnected. Since a block of redundant contacts are placed in the via layer, to disconnect the contact, the defect must entirely cover at least one via block. Preprocessing the Hausdorff Voronoi diagram is very effective for this problem, since in each Hausdorff region of the diagram we need only consider the farthest point of one set, rather than all points from all sets, when determining whether a fault occurs for a circular defect at some point.

We analyse the current state of the art in sequential Hausdorff Voronoi diagram

construction and then proceed to the presentation of a novel CGM algorithm for non-crossing Hausdorff Voronoi diagram construction. Although the literature provides examples of parallel algorithms for Voronoi diagrams of points, no parallel algorithm is known to exist for constructing Hausdorff Voronoi diagrams. Our algorithm provides theoretical speedup of p and may be easily extrapolated to a novel sequential algorithm which improves on previous known bounds for the non-crossing case. Due to the size of the input data in VLSI applications, it may be of direct practical relevance to provide a parallel algorithm which is faster than conventional sequential algorithms and which can provide access to the additional memory of a distributed memory multicomputer or cluster. To provide an initial evaluation of the algorithm in practice, we also discuss the results of our prototype implementation and its performance.

Our Hausdorff Voronoi diagram parallel algorithm for non-crossing input uses the divide and conquer paradigm to merge subdiagrams sequentially computed at each processor. This CGM algorithm performs the merge by locating the endpoints of each subdiagram's edges in the opposite subdiagram. These located endpoints may then determine the subset of edges crossing the *merge curve.* For the Hausdorff variant of the Voronoi diagram, the merge curve is interesting, since it may be comprised of multiple, disjoint components that are not necessasrily y-monotone. In fact, some of these *merge components* may even be cyclic. So to determine the merge vertices of the merge curve, we make use of a parallel algorithm that performs a search variant of the Red-Blue line intersection problem. Line intersection is used to search for merge vertices on the edges which potentially cross the merge curve.

Our primary contribution in this thesis is to develop the first parallel CGM algorithm that computes the Hausdorff Voronoi diagram for non-crossing input, using $O(\frac{n \log^4 n}{p})$ local computation for input of size n on a CGM machine with p processors. As a second contribution, we perform an initial experimental study of this parallel algorithm. Experimental work involves the implementation and performance evaluation of our parallel algorithm and the sequential sweepline algorithm for the Hausdorff Voronoi diagram problem. Our parallel algorithm also results in a direct adaptation to a sequential algorithm which requires $O(n \log^4 n)$ time. This new sequential algorithm improves on existing time bounds for the non-crossing case.

1.1 Organization of Thesis

In Chapter 2 we review the relevant canonical literature in both computational geometry and parallel algorithm design. This chapter presents an introduction to the Voronoi diagram of points and a variety of sequential algorithms to construct such diagrams. Chapter 2 then discusses parallel computing models, focusing attention on the application of the CGM model to algorithm design. The chapter concludes by discussing parallel geometry algorithms for the Voronoi diagram of points and some algorithms using the *segment tree* data structure. Chapter 3 reviews the state of the art in the Hausdorff Voronoi diagram literature including sweepline and divide-and-conquer algorithms, as well as the Hausdorff Voronoi diagram's application to VLSI manufacturing. Chapter 4 presents our novel parallel CGM algorithm and our improved sequential algorithm for non-crossing input.

Chapter 5 presents our implementation of the parallel and sequential sweepline algorithms and then discusses their performance. Finally, Chapter 6 concludes the discussion of Hausdorff Voronoi diagrams.

Chapter 2

Computational Geometry and Parallel Computing Review

Let us begin by reviewing the existing literature surrounding our fields of study. From computational geometry, we discuss Voronoi diagrams and some existing algorithms for their construction. We proceed to a discussion of parallel computing models for parallel algorithm design. We conclude this chapter by discussing the relevant literature surrounding parallel computational geometry using the CGM model. This discussion includes CGM algorithms to solve batch point location, red-blue line intersection, and Voronoi diagram construction.

2.1 Voronoi Diagrams of Points

2.1.1 Overview

Voronoi diagrams are a fundamental structure in Computational Geometry, and have been studied for decades (e.g. [42]). In this Chapter, we review some important properties and algorithms for Voronoi diagrams of points. We review the

Sweepline and Divide-and-Conquer algorithms for computing Voronoi diagrams, and briefly describe an approach mapping the Voronoi diagram to 3D envelope computation. There are other algorithms for Voronoi diagrams, including an incremental approach and the computation of the dual Delaunay triangulation structure. However, we do not discuss all of these algorithms as they are not directly relevant to our parallel Hausdorff Voronoi diagram algorithm and implementation. For the interested reader, a broader survey of Voronoi diagrams may be found in, e.g., Aurenhammer [4], Okabe et al [30], or Fortune [21].

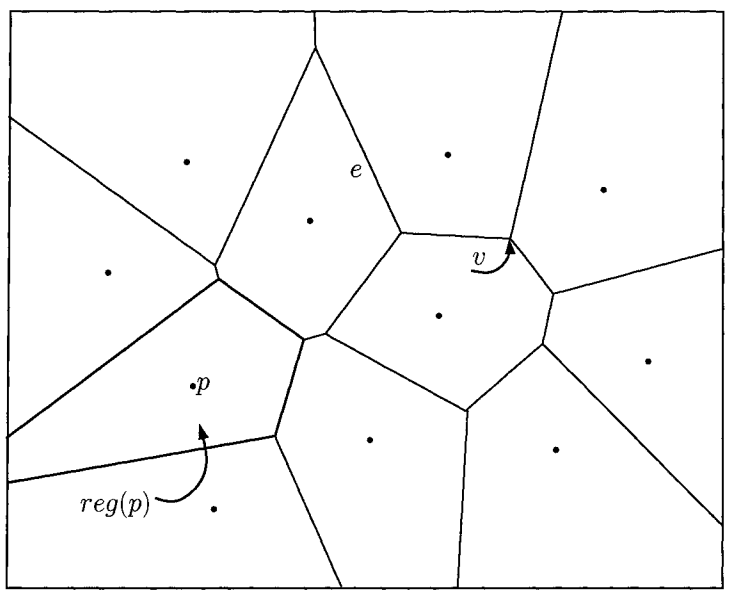


Figure 2.1: An example Voronoi diagram of Points

2.1.2 Definition and Properties

Informally, a Voronoi diagram partitions the plane so that, for each site, there is a region comprising the space *closest* to the site. The Voronoi diagram of points

defines closest as minimal Euclidean distance. The Euclidean distance between two points in the plane may be formally defined for two points, $p = (p_x, p_y)$ and $q = (q_x, q_y)$:

Definition 1 (Euclidean Distance) *The Euclidean distance between two points p and q is $d(p,q) = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}$.*

Given some distance function, such as the Euclidean metric we just described, we may define the concept of a bisector. Between two points, p and q, the bisector is the perpendicular line to $\overline{pq}$ passing through the midpoint of $\overline{pq}$. This is, more precisely, the locus of points equidistant from p and q, thus:

Definition 2 (Bisector of two points) *The bisector of two points p and q is $\mathcal{B}(p,q) = \{z | d(p,z) = d(q,z)\}$.*

Note that we may use term bisector in a more general fashion, although it will still mean the locus of equidistant points between two sets of geometric objects. For example, we will make reference to a bisector between a point and a line, as well as a merge bisector between two set of points, $\mathcal{L}$ and $\mathcal{R}$. The type of bisector should be obvious from the context. Let us now define halfplanes, closely related to bisectors:

Definition 3 (Halfplane of two points) *The halfplane $H(p,q)$, of two points p, q, is the the set of points $H(p,q) = z | d(z,p) < d(z,q)$.*

Given these preliminary definitions, we may construct a formal definition of a Voronoi diagram:

Definition 4 (Voronoi Region $reg(p), p \in \mathcal{S}$**)** *A Voronoi Region is the locus of points closer to p than all* $q \in \mathcal{S}, q \neq p$. *By definition,* $reg(p) = \bigcap_{q \in \mathcal{S}} H(p, q)$.

Definition 5 (Voronoi Edge, e**)** *A Voronoi Edge, e, is the locus of points with exactly two closest points* $p, q \in \mathcal{S}$. *By definition, e is a portion of* $\mathcal{B}(p, q)$.

Definition 6 (Voronoi Vertex, v**)** *A Voronoi Vertex, v, is a point with at least three closest points in the input set* $\mathcal{S}$.

Since computing the Voronoi Regions amounts to determining the boundaries of the regions, we define the whole Voronoi diagram accordingly:

Definition 7 (Voronoi diagram $Vor(\mathcal{S})$**)** *Given a set S of input points (sites), the Voronoi diagram of points,* $Vor(\mathcal{S})$, *is the union of* Voronoi Edges *and* Voronoi Vertices *of* $\mathcal{S}$.

Given the definition of the Voronoi diagram, it is now possible to describe how to construct the diagram for a given input set, $\mathcal{S}$. In Chapter 3, we discuss how these methods can be extended to the more general Hausdorff metric.

Before discussing the sweepline and divide and conquer algorithms, let us briefly describe how to compute Voronoi diagrams via 3D upper envelopes. We lift all input sites from the plane (e.g. $z = 0$) onto a unit paraboloid, and create the set P of planes tangent to the paraboloid at each lifted point. The key observation is that the vertical distance from a point in the plane to an input site is equals to the square root of the vertical distance between the lifted point and the input site's plane in P. It follows that the upper envelope of P encodes the pieces of each plane in P that vertically map to a Voronoi region.

2.1.3 Sweepline Algorithm

Sweepline techniques form an important algorithmic paradigm in computational geometry, and are useful for computing the Hausdorff Voronoi diagrams. Conceptually, the sweepline paradigm involves a vertical line, L, moving from left to right. Special points in the plane are denoted events. Events comprise the points at which the topological structure of the geometric object in question changes. In our case, the geometric structure is a Voronoi diagram, and the events come in two types, *spike* events and *site* events. A site event occurs where an input site is located, and it is where a new regions begins to be traced. A spike event is where a Voronoi vertex occurs, and it is where a regions ends.

We must keep three data structures as the sweepline progresses. One is the constructed portion of the Voronoi diagram. A second is the *Beachline* providing an ordered sequence of partially finished Voronoi regions. The third is the *Event Queue* that contains events which we have not yet processed, ordered from left to right.

Algorithm Data Structures

Definition 8 (Event Queue) *The Event Queue, Q, is a priority queue storing events in left-to-right order.*

Definition 9 (Beachline) *Let $S_L \subseteq S$ be the points to the left of the sweepline, L. Let V_L be the Voronoi diagram for the input sites $S_L \bigcup L$. Then the Beachline, T, stores the sequence of (parabolic) Voronoi Edges between L and points in V_L. Let these edges be called* waves.

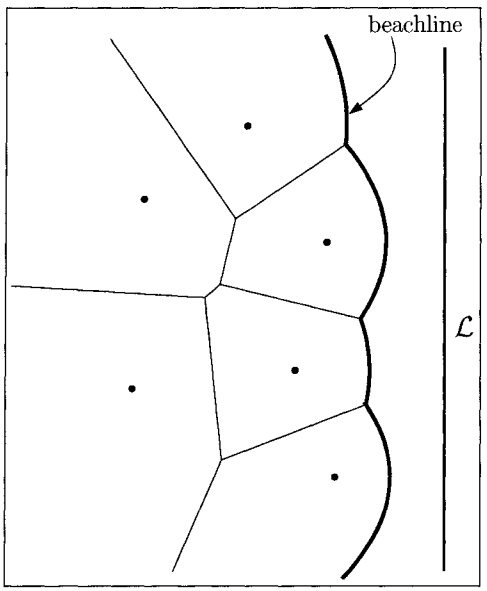


Figure 2.2: Sweepline Algorithm

Note that the waves in the beachfront form a y-monotone chain of parabolic segments.

Outline of the Algorithm

Algorithm 1: SWEEPLINE()

1 For each $p \in S$, add an event for p into the event queue, Q.

2 While Q is not empty, get the next event in the Queue. If it is a site event, run SITE_EVENT(). Otherwise, the event is a spike event, so run SPIKE_EVENT().

Algorithm 2: SITE_EVENT()

1 Locate the wave, $w \in T$ whose y-interval contains the new site, p.

2 Split w in two pieces separated by a new wave, w', associated with p.

3 If w was involved in a spike event with its two neighbours, then delete this event from Q.

4 Create a new edge in the Voronoi diagram associated with w'.

5 Check if any new spike events are induced by w' with its neighbouring waves, and if so, add them to the queue.

Algorithm 3: SPIKE_EVENT()

1 A Spike Event is associated with three adjacent waves. Determine the middle wave, w. This wave has now degenerated to a Voronoi Vertex.

2 Delete w from the Beachfront, T.

3 If there are any (other) events in Q that involve w then delete them.

4 There are three sites associated with the adjacent waves in the site event. Create a Voronoi Vertex at the centre of the circle through these sites.

The sweepline algorithm for computing Voronoi diagrams was originally studied by Fortune [20]. The algorithm's time complexity is bounded by $O(n \log n)$. For each of the $O(n)$ events, we perform $O(\log n)$ computation in the beachfront, assuming it is represented as a balanced binary tree.

2.1.4 Divide and Conquer Algorithm

Divide and Conquer is another technique frequently employed in solving computational geometry problems. The first worst-case optimal algorithm for constructing

Voronoi diagrams, presented by Hoey and Shamos in 1975 [42], uses this technique.

As usual, we are given a set of sites, $S = \{p_1, p_2, \ldots, p_n\}$. Each site, $p_i \in S$, is a point in $\mathbb{R}^2$ and is described by the coordinates $p_i = (x_i, y_i)$.

Algorithm 4: DIVIDE-AND-CONQUER(S)

1 Before computing the Voronoi diagram, pre-sort S lexicographically, by x_i.

2 Compute m, the median of the sites' x-coordinates. Since S is sorted, $m = x_{\frac{n}{2}}$.

3 Partition S into a left and a right set, $\mathcal{L}$ and $\mathcal{R}$, i.e. $\mathcal{L} = \{p_i : x_i \leq m\}$, $\mathcal{R} = S - \mathcal{L}$.

4 Recursively compute $Vor(\mathcal{L})$ and $Vor(\mathcal{R})$.

5 $Vor(\mathcal{S}) \leftarrow$ MERGE($Vor(\mathcal{L}), Vor(\mathcal{R})$).

The divide-and-conquer algorithm is straightforward. Dividing the problem into roughly equal subproblems is satisfied simply by recursing on left and right partitions of the input. Most of the computation is in the MERGE() sub-algorithm, below. Between the subdiagrams of the Left and Right partitions, we have a y-monotonic chain of Voronoi Edges. This *merge chain* forms $B(\mathcal{L}, \mathcal{R})$, the bisector between $\mathcal{L}$ and $\mathcal{R}$. Merging the subdiagrams requires only finding the Voronoi Edges of the merge chain, and removing the portion of $\mathcal{L}$ (resp. $\mathcal{R}$) to the right (resp. left) of the merge chain.

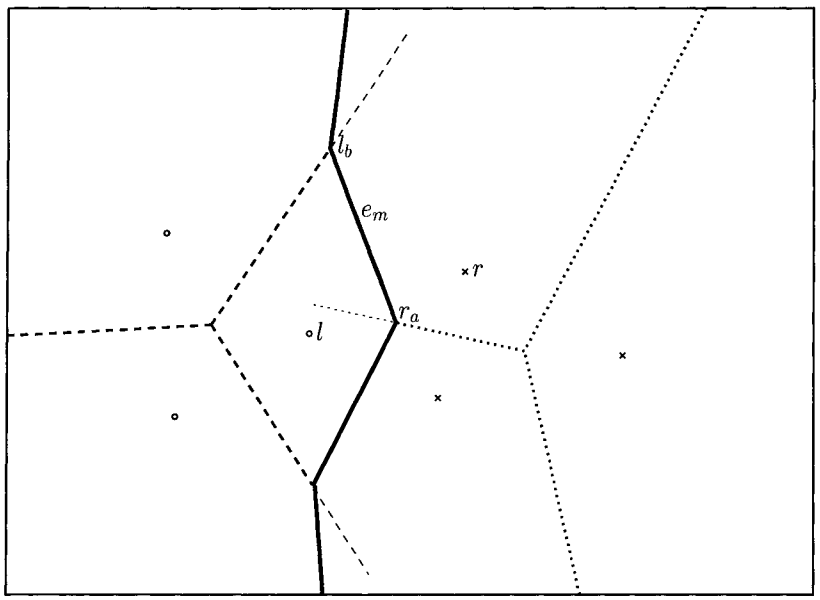


Figure 2.3: Merging two subdiagrams

Algorithm 5: MERGE($\mathcal{L}, \mathcal{R}$)

1 Using the convex hulls $CH(\mathcal{L})$ and $CH(\mathcal{R})$, compute the lower supporting segment.

2 Starting from the bisector of the lower supporting segment, repeat this process until reaching the bisector of the top supporting segment:

3 W.l.o.g., assume the current merge edge, e_m, starts at the boundary of some region $reg(r)$ of $Vor(\mathcal{R})$. Let this starting point be r_a. Let r_a be in the Left Voronoi region $reg(l)$. To find the endpoint of e_m, we scan the edges bounding $reg(r)$ counterclockwise, starting from r_a, until we find the intersection point r_b. Scanning edges of $reg(l)$ clockwise, we likewise find an intersection point l_b. See Figure 2.3 for an example.

4 From l_b and r_b we take point closest to r_a, and update $reg(r)$ or $reg(l)$ accordingly. In our next clockwise/counterclockwise edge scans, we start scanning from l_b and r_b.

The first step of the merge determines the supporting segments of the Left and Right convex hulls. The bisectors of these two supporting segments provide the start and end of the merge chain. The remainder of the algorithm performs a sequential walk along the edges of the merge chain. Since regions are convex, we never need to rescan edges already scanned during our walk.

The running time of the DIVIDE-AND-CONQUER algorithm is easily analysed. When walking through regions intersected by the merge chain, edges are visited at most once. Hence, the entire MERGE() algorithm runs in linear time. The entire recursive DIVIDE-AND-CONQUER therefore has a worst-case running time of $O(n \log n)$.

2.2 Parallel Computing

2.2.1 Overview

Parallel computing offers the possibility of significantly increased performance over conventional sequential computing. However, algorithm design and implementation is difficult in a parallel setting. Attempts have been made to create a general, useful model for designing parallel algorithms. A parallel computing model should simplify both algorithm design and code implementation by capturing the similarities between the wide variety of target parallel architectures while abstracting away unnecessary details of any particular architectural design. We briefly review two such models.

2.2.2 PRAM Model

One of the oldest and most popular models of parallel computation is the Parallel RAM (PRAM) model, introduced by Fortune and Wyllie [22]. A PRAM machine is a conceptual extension of the sequential RAM model of computation. A PRAM has a set of processors sharing a global shared memory to which all processors may simultaneously read or write. Many variations of the PRAM model exist. The major variation in PRAM models concerns whether and how multiple, concurrent memory reads and writes by processors to the same memory location may occur.

PRAM provides a platform for creating fine-grained parallel algorithms. Many processors may be used (for example, $O(n)$ processors for a problem of size n), and frequent communication can occur through shared memory.

The PRAM model does provide a computational model well-suited to designing algorithms. However, a large-scale parallel computer following the PRAM model has not been created, and it is frequently asserted in the literature that the model is unrealistic approximation of existing machines (e.g. [24]). One main problem is that the PRAM model assumes a shared-memory machine. However, to have a large number of processors, most existing machines make use of distributed memory, such as clusters. With a large-scale distributed architecture, communication costs between nodes can become quite significant, but are not accounted for in the PRAM model.

2.2.3 CGM Model

The PRAM model may be powerful and theoretically useful, but is has been acknowledged (e.g. [9, 14, 24]) to provide algorithms that are difficult to efficiently implement on existing parallel architectures. There have also been many algorithms designed for specific architectures (e.g. mesh, hypercube, fat tree, etc.). Yet ideal algorithm design should provide an algorithm not limited to one specific architecture. The ideal parallel model should be general. This is an issue addressed by Valiant in his classic paper [45].

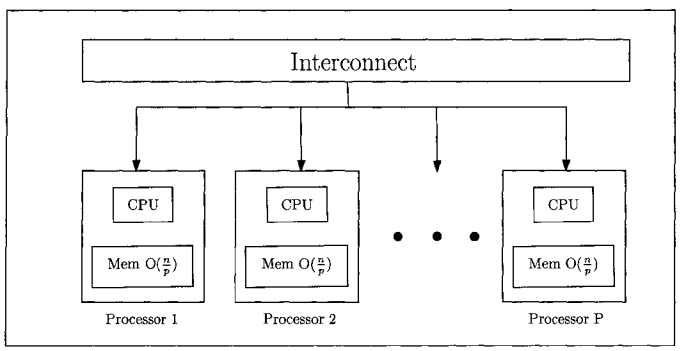


Figure 2.4: CGM Architecture

Valiant proposes the BSP model. This model is bulk-synchronous, which means that an algorithm is composed of a series of supersteps. In each superstep, processors run independently. The end of a superstep comprises a synchronisation between the processors, after which completion of any communication between processors from that superstep is guaranteed.

The CGM model, introduced by Dehne et al. [14] provides for a computational model similar to the BSP. The key features of the CGM model include

its enforced granularity due to lower bounds on the ratio $\frac{n}{p}$ and its explicit message packing between rounds to reduce the effect of communication latency costs. Unlike the PRAM model, which is modelled using shared-memory, CGM uses message-passing. Message-passing is realistic since communication is relatively expensive on current hardware (compared to computation), and should be accounted for. Also, CGM enforces a coarse grained computing model, so $p \ll n$, contrasting with the fine-grained PRAM model where, often, $p \approx n$.

A CGM has p processors with arbitrary interconnect, each typically with $O(\frac{n}{p})$ local memory. A CGM has the ability to realize an h-relation. An h-relation is an operation that routes h data (usually $h = O(\frac{n}{p})$ on a CGM) data to and h data from each processor. Algorithms implemented on a CGM consist of rounds comprised exclusively of local computation, separated by rounds of communication.

This type of coarse-grained model works particularly well on cluster machines, such as a Beowulf. Clusters have a moderate number of quite powerful distributed memory processors.

However, a CGM algorithm is quite general, and allows for various interconnects by using the abstract notion of an h-relation. Analysis of the CGM model on a particular parallel architecture only requires one to determine the time to realize an h-relation on that architecture.

The CGM model encompasses a small number of critical parameters. Hence it is a straightforward and easily used tool in designing parallel algorithms. Moreover, such algorithms should be transferable to efficient implementations on a variety of architectures.

2.2.4 Analysing CGM Algorithms

Algorithms designed for the CGM model may be analysed with three important measurements which are commonly used in the literature (e.g. [6, 14]) to profile the performance of algorithms. These measures are local computation, number of communication rounds and scalability.

Local computation is a standard tool for algorithm analysis and applies equally well to algorithms designed for a CGM. Each processor performs a certain amount of computation throughout all of an algorithm's computation rounds. The maximum of each processor's computation time complexity is taken to be the local computation of the parallel algorithm.

We can also derive another measurement from the local computation of a parallel algorithm. Assume that we have an optimal sequential algorithm for our particular problem, which has a running time of T_s. Then, if our algorithm has local computation of T_p, we can calculate speedup, $S = \frac{T_s}{T_p}$. Note that for a CGM algorithm, maximum possible speedup is p.

Our second measure is the number of communication rounds. Ideally we will have $O(1)$ rounds, which provides a constant upper bound on the total amount of data communicated during the life-cycle of the algorithm. However, other efficient algorithms have more rounds, for example $O(\log p)$ [6, 15]. While not ideal, we note that for a coarse-grained algorithm $p \ll n$, i.e. in a typical multicomputer, p is insignificant compared to n. For a particular machine, p tends to be fixed at a constant size, while the problem size may vary and be quite large. Hence, on such a coarse-grained machine, $\log p$ rounds may still be practical.

Our other measure is slackness. Slackness is indicated by the required minimum ratio of n to p. For example, $\frac{n}{p} \geq p$ is common.

To illustrate the use of our CGM analysis techniques, we describe a few well known algorithms. For example, let us discuss the computation of three-dimensional convex hulls. An algorithm exists [13] to compute the three-dimensional convex hull using the CGM model. To summarise the analysis of this algorithm, we say that is uses $\tilde{O}(\frac{n \log n}{p})$ local computation and $O(1)$ rounds, assuming $n \geq p^{3+\epsilon}$.

Sorting may be one of the most fundamental techniques in algorithm design. An algorithm appropriate for the CGM model was introduced by Goodrich [23] that can sort using $O(\frac{n \log n}{p})$ local computation and $O(1)$ rounds, for any scalability $p \leq n^{1-1/c}, c \geq 1$. Previous work on parallel sorting included the parallel sort by regular sample, a straightforward and practical algorithm that requires $n \geq p^3$, as describe by Shi and Schaeffer [43]. An algorithm for the special case of sorting bounded integers was given in [8] and requires $O(n)$ time, $O(1)$ rounds, if $n \geq p^2$.

2.2.5 Basic CGM Algorithms

Before proceeding to discuss some existing CGM algorithms which are relevant to our topic, we first present some basic algorithms which may serve as standard tools when designing CGM algorithms. These algorithms include a form of load balancing, global sorting, and global batch binary search.

Sorting and searching are fundamental techniques used in most algorithms, including the algorithms we discuss. Although neither sorting nor searching are difficult to understand, reviewing them provides a good starting point for our discussion of CGM algorithms.

Sorting

We discuss here a practical and easy to implement sort called the Parallel Sort by Regular Sample [43]. We start with n elements arbitrarily distributed into groups E_i of $O(\frac{n}{p})$ elements for each processor P_i, which we wish to sort globally. That is, the concatenation of elements across processors $P_0, P_1, \ldots, P_p$ provides a globally sorted sequence.

Algorithm 6: CGM_SORT()

1 Each processor locally sorts its portion, E_i of the input.

2 Each processor selects p regularly spaced samples from the sorted E_i. All samples are collected at the a root processor (e.g. P_0).

3 The root processor sorts the p sample sequences and then chooses p regularly spaced *global* samples from this sorted sequence.

4 The root processor broadcasts the global sample sequence, $(S_0, S_1, \ldots S_{p-1})$ to all processors.

5 Each processor divides its sorted E_i sequence into p subsequences, $E_i^k, 0 \leq k < p$ such that all elements in E_i^k are between S_k and S_{k+1}.

6 The kth processor, P_k receives $E^k = \bigcup_{i=0}^{p} E_i^k$ from each processor P_i. Processor P_k sorts this E^k.

To analyse the correctness this sorting algorithm, we verify a key point; E^k will always be balanced, so that $|E^k|$ is $O(\frac{n}{p})$. Recall that E^k is the set of elements between two global samples. By definition, there are p samples between two global samples. Each group of $\frac{n}{p^2}$ samples in a processor's S_i^k will contain a local sample. But since there can be no more than p such local samples present from

all processors, the size of S^k must be $O(p\frac{n}{p^2}) = O(\frac{n}{p})$.

We note that the bound on local computation derives from the local sorts of $O(\frac{n}{p})$ elements that requires $O(\frac{n \log n}{p})$ computation. There is a communication round in Steps 2, 4, 6, and hence the overall algorithm requires $O(1)$ rounds. For the root processor to receive all samples in Step 2 and sort them in Step 3, the algorithm requires $n \geq p^3$.

Grouping

Another simple parallel technique is to sort a sequence of sets across processors, where each set may be of size $O(\frac{n}{p})$. The sets are, in total, of size $O(n)$. The problem is to sort these sets while still maintaining the $O(\frac{n}{p})$ local memory bound and not splitting sets across processors.

Although this is an almost trivial problem, we make use of this simple algorithm in later, more interesting CGM algorithms, so we explain the technique once here to simplify the presentation of future algorithms.

Each processor starts with some unsorted sets of total size $O(\frac{n}{p})$. Each set is stored entirely at one processor.

Algorithm 7: GROUPED_SORT()

1 Use global sort to rank each set (without sending the entire set).

2 For each element, perform global sort with set rank as primary key, and element rank as secondary key.

3 Share each processor's first and last elements' set ranks with all other processors.

4 Rebalance by sending a set split across processors to the first processor that stores a portion of the set.

This algorithm clearly requires O(1) rounds, $O(\frac{n \log n}{p})$ local computation, and $O(\frac{n}{p})$ memory with the restriction that $n \geq p^2$.

Loadbalancing

To ensure that a CGM is being effectively used, it is often necessary to redistribute the data held at each processor, so that each processor has about the same amount of computation to perform. When designing an algorithm for the CGM model, we may explicitly perform this loadbalancing. For the algorithms we discuss, there is one particular form of loadbalancing which is very useful. To facilitate an abstract discussion of this technique, we refer to a set S of n small items that are each constant sized objects. We also discuss a set D of p large data structures, each of size $O(\frac{n}{p})$. We assume each s is uniquely associate with one d, i.e. there is some mapping $id : S \mapsto D$. The goal is to distribute the $s \in S$ and $d \in D$ such that every s is locally stored with the correct d, while maintaining the $O(\frac{n}{p})$ local memory bound of the CGM model. We assume that each data structure d is

initially stored at processor P_d, and each processor stores a $O(\frac{n}{p})$ subset of S.

Algorithm 8: LOADBAL()

1 For each $d \in D$, each processor, P_i, locally determines $C_i(d)$, the number of s that for which $id(s) = d$.

2 Each processor P_i sends $C_i(d)$ to all other processors (for every $d \in D$).

3 Each processor independently computes $C(d) = \sum_i C_i(d)$ for all d.

4 For every d, each processor determines the number $n_d = \lceil \frac{C(d)}{n} p \rceil$, which determines the number of copies to make of each data structure d. Note that each such copy will then have $O(\frac{n}{p})$ items of S associated with it. Processor P_d performs a segmented broadcast of its data structure, d, to n_d processors. There are at most $2p$ total copies made the d's, so each processor can receive at most 2 data structures.

5 Each processor determines where to send its local subset of S such that no processor will receive more than $O(\frac{n}{p})$ of S, but each $s \in S$ is located at a processor storing the d such that $id(s) = d$. The items $s \in S$ are distributed accordingly.

This algorithm requires a constant number of rounds. If $n \geq p^3$ then the algorithm will run on a CGM in local computation time $O(\frac{n}{p})$.

The correctness of the algorithm hinges on three points. First, every processor has complete knowledge of every processors' item counts, so that the computation of Steps 4 and 5 can be computed at each processor independently, but coordinated with all other processors. Second, enough copies of each d are made so that no copy need be associated with more than $O(\frac{n}{p})$ s objects. Third, the number of

total copies of all $d \in D$ is bounded by $\sum_{d=1}^{p} \left(\lceil \frac{C(d)}{n} p \rceil \right) < \sum_d \left(\frac{C(d)}{n} p + 1 \right) \leq 2p$.

Binary Search

A simple batch binary search of n queries in a sequence of n elements may now be trivially described, using the previous load balancing. Let us assume that we have a sequence $(s_0, s_1, \ldots s_n)$ that is globally sorted across processors, such that each processor stores a portion S_i. Each processor P_i also stores a set of queries, Q_i.

Algorithm 9: CGM_BINSEARCH()

1. Each processor, P_i sends the largest element in S_i to all other processors.
2. Each processor then determines in which processor's subsequence each query $q \in Q_i$ needs to be searched.
3. The queries and subsequences are loadbalanced (Algorithm 8).
4. Each processor locally performs sequential binary search for balanced queries and subsequences.

This search algorithm, originally described by Devillers et al [19] is clearly possible using $O(\frac{n \log n}{p})$ local computation, and a constant number of rounds, given $n \geq p^3$.

2.3 A CGM Algorithm for Voronoi Diagrams of Points

Considering the long history and broad application of the Voronoi diagram, it is not surprising that much attention has been given to finding efficient parallel algorithms for constructing them. Many PRAM algorithms have been devised (e.g. [12, 40, 46]). A CGM algorithm, introduced by Diallo et al. [15] also exists for Voronoi diagrams of points. This CGM algorithm uses techniques devised by Jeong and Lee [25, 26, 27] for a mesh Voronoi diagram algorithm.

As seen in Section 2.1.4, the divide-and-conquer approach divides Voronoi diagrams into subproblems, and then merges the recursively computed results. The dividing step may be trivially parallelised. The merge step requires the most attention, since this is where the merge chain is traced. In the original sequential algorithm, merge chain tracing is performed by sequentially walking up the merge chain. In a parallel setting, the major objective is to avoid this sequential walk and rather compute each Voronoi Edge in the merge chain independently of the rest of the merge chain. Diallo et al's algorithm does exactly that.

2.3.1 Voronoi Diagram CGM Algorithm

This section and the following section closely follow the results from [15], but also incorporates some changes suggested by Singler [44]. An overview of this CGM algorithm for computing Voronoi diagrams of points is given:

Algorithm 10: CGM-VORONOI()

1 Divide the $O(n)$ input points into p vertical slabs., with $O(\frac{n}{p})$ points in each slab. Each processor gets a slab.

2 Each processor independently computes the Voronoi diagram of its slab's $O(\frac{n}{p})$ input sites.

3 CGM-MERGE() (see below) combines $\frac{p}{2^i}$ diagrams into $\frac{p}{2^{i-1}}$ diagrams, such that two adjacent subdiagrams, $Vor(\mathcal{L})$ and $Vor(\mathcal{R})$ merge to form $Vor(S)$. Repeat this $O(\log p)$ times, resulting in the final Voronoi diagram.

2.3.2 Voronoi Diagram CGM Merge Algorithm

An efficient CGM-MERGE() algorithm is critical for Algorithm 10. The merge relies on a parallel Point Location algorithm, as well as standard Global Sort operations. Here is a review of the merge algorithm.

In many divide-and-conquer Voronoi diagram algorithms, including Algorithm 10, the important computation is that of the *merge chain*. Recall that we have already taken a point set S and partitioned it into a left half, $\mathcal{L}$, and a right half, $\mathcal{R}$. We have computed $Vor(\mathcal{L})$ and $Vor(\mathcal{R})$. The Merge procedure then computes $Vor(S)$ from these two sub-diagrams. The *merge chain* consists of the edges in $Vor(S)$ which border on both a site from $\mathcal{L}$ and a site from $\mathcal{R}$. These new edges form a monotonic unbounded chain. The merge step determines the merge chain, and then removes the portion of $Vor(\mathcal{R})$ to the left of the merge chain, and the portion of $Vor(\mathcal{L})$ to the right of the merge chain.

This CGM algorithm takes a different approach from the classic sequential

divide-and-conquer algorithm (Section 2.1.4). Instead of walking up the merge chain as it is calculated, the CGM algorithm instead independently determines the subset of edges (from both $Vor(\mathcal{L})$ and $Vor(\mathcal{R})$) intersected by the merge chain and orders these. It is then trivial to perform smaller local walks at each processor.

To determine the subset of edges intersected by the merge chain, we use parallel point-location. The endpoints of an existing edge in (W.L.O.G.) $Vor(\mathcal{L})$ may be found in Voronoi Regions of $Vor(\mathcal{R})$. After this point location, it is easy to compute to which set ($\mathcal{L}$ or $\mathcal{R}$) each endpoint is closer to. Note that this is equivalent to computing to which side of the merge chain each endpoint lies in. Edges with endpoints on different sides of the merge chain must intersect the merge chain at some point. The following algorithm uses this fact:

Algorithm 11: CGM-MERGE()

1 W.L.O.G., for the left Voronoi diagram, $Vor(\mathcal{L})$, partition edges of $Vor(\mathcal{L})$ into three sets (L_L, L_R, R_R), such that:

1. L_L contains edges whose two endpoints are both closer to $\mathcal{L}$ than $\mathcal{R}$. These edges do not cross sides, so they remain intact.

2. L_R contains edges where one endpoint lies closer to $\mathcal{L}$, and the other to $\mathcal{R}$. These edges cross the Merge Chain once.

3. R_R contains edges whose two endpoints are both closer to $\mathcal{R}$ than $\mathcal{L}$. These edges do one of two things. Either they do not cross sides, thus being completely deleted in $Vor(S)$, or else they cross the merge chain twice.

   If an edge crosses twice, then break it up into two pieces, such that each piece crosses the edge once (i.e. create two pieces for the set L_R).

2 Repeat above, but now locating edges of $Vor(\mathcal{R})$ in $Vor(\mathcal{L})$.

3 Perform a Global Sort (on y-coordinates) of the complete set of intersecting edges, ensuring that endpoints of edges with overlapping endpoints are ordered appropriately. This gives $O(\frac{n}{p})$ adjacent edges at each processor. Each processor can then do local "walking" to produce the complete merge chain.

4 Remove every edge that is completely on the opposite side. The previous step determined intersection points on edges that crossed the Merge Chain. Remove the portion of these edge on the opposite side.

The merged Voronoi diagram is now computed, distributed across processors.

2.3.3 Analysis

The CGM Voronoi diagram construction algorithm performs sequential Voronoi diagram construction followed by merge steps. The merge algorithm uses point location, a common geometric operation with complexity $O(\frac{n \log n}{p})$. Since the Voronoi diagram of points' regions are always convex (a property which fails for Hausdorff Voronoi diagrams), the algorithm's authors [15] use the chain method [28] and a series of load balancing steps to perform parallel point location.

Point location is used on the endpoints of edges in the two subdiagrams. These located endpoints provide the merge chain and determine which subdiagram edges to remove or to keep.

One critical observation for demonstrating the correctness of the merge is to show that the case analysis on the located endpoints is correct. Since a Voronoi region is connected, only one continuous piece of a subdiagram edge may be kept. This implies that no more than two merge chain intersections occur on an edge. As a result, an edge with endpoints closer to different subdiagrams (i.e. an edge in L_R) will cross exactly once. An edge with endpoints closer to the opposite subdiagram (i.e. an edge in R_R) may cross either zero or two times. Otherwise, the endpoints are closer to the original subdiagram, and the entire edge will remain in the final diagram. A more formal proof which has been extended to handle the more general Hausdorff Voronoi diagram, is provided in Section 4.3.

There is another critical observation for demonstrating correctness of the merge algorithm. Some subdiagram edges are determined to contain a merge vertex. We

order these edges vertically. Unfortunately, achieving this is not quite as simple as the method proposed in [15]. However, it is possible to use the original method proposed in [27]. This method ensures that even edges whose y-intervals overlap will be correctly ordered. Crucially, the merge chain of the Voronoi diagram of points is y-monotonic, unlike the Hausdorff Voronoi diagram. Using this y-monotonicity and the point located endpoints, we may determine the vertical ordering of even overlapping edges. Using an exhaustive case analysis, Jeong and Lee [27] demonstrate that for any given case, there is only one possible order that ensures that both the merge chain is y-monotonic and that the endpoints lie closer to the left and right subdiagram, as determined by point location.

Theorem 1 ([15]) *The CGM algorithm for Voronoi diagrams of points constructs the Voronoi diagram of an input set S of size n in $O(\frac{n \log n \log p}{p})$ local computation, with $O(\log p)$ rounds, given that $n \geq p^3$.*

2.4 CGM Segment Tree Algorithms

2.4.1 Parallel Next Element Search

Standard Segment Tree

Parallel point location in planar subdivisions is one subproblem which we make use of in our parallel Hausdorff construction algorithm. The problem is as follows: given an embedded planar graph of size n, determine the face in which each of n query points reside. This problem is a specific application of the more general next element search problem. This more general problem is to find, for each query

in a $O(n)$ set of query points, a segment directly above the query from a $O(n)$ set of non-intersecting segments.

A segment tree is a balanced binary tree. The segment tree is constructed for a set L of n non-intersecting input segments (i.e. the segments forming the embedded graph). The x-projections of the $2n$ segment endpoints creates a partitioning of the x-axis into elementary x-intervals. There is one leaf for each elementary x-interval. The x-interval for an internal node is the union of the node's two children x-intervals.

Each node is associated with a catalog. The catalogs are filled with references to the input segments. A segment is placed in a node's catalog exactly when the line segment completely spans the node's interval, but does not completely span the interval of the node's parent.

Constructing a standard segment tree is not difficult and is discussed in [39]. Querying the tree is also straightforward. A query point q can traverse a path from the root to a leaf such that all nodes along that path contain q in their x-intervals. At each of these $\log n$ nodes we may perform binary search in the node's catalog to determine $O(\log n)$ segments above q. A final scan of these elements determines the segment directly above q.

However, note that the query algorithm requires $O(n \log^2 n)$ time. We introduce segment trees in order to present a parallel version. However, for solving the next element search subproblems sequentially we may use a simple sweepline algorithm running in $O(n \log n)$ [39]. Recall that these segments are assumed to be non-intersecting. The sweepline algorithm is not difficult but briefly, we create sweepline events for each segment endpoint and query point. At endpoint events,

we add/remove segments from an ordered sequence. At a query event, we search the sequence for the segment directly above the query point.

Now, we turn to parallelising the standard segment tree using the CGM model to give a parallel next element search algorithm.

CGM Algorithm

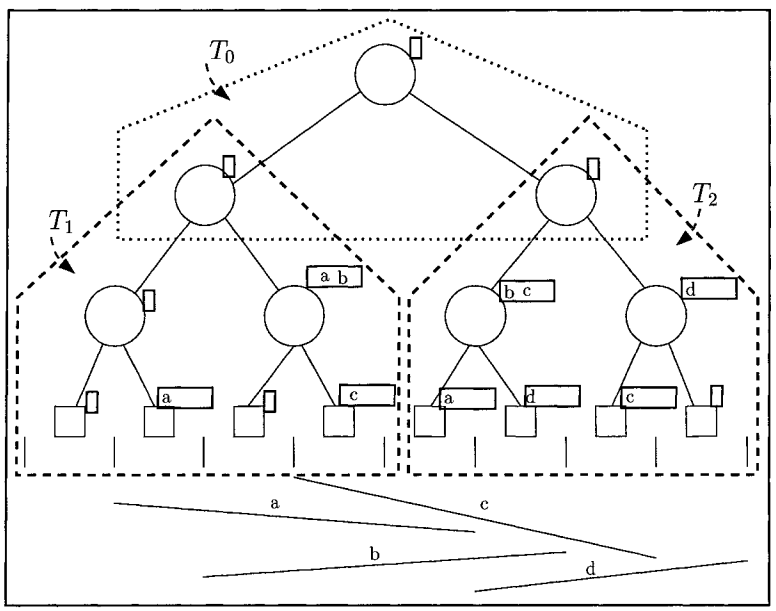


Figure 2.5: Example Distributed Segment Tree

Chan et al [9] present a CGM algorithm that solves the point location problem. For this algorithm, a parallel segment tree must be constructed. This distributed structure takes the first $\log p$ levels as subtree T_0. At the leaves of T_0 are rooted p subtrees, $T_1, \ldots T_p$. Each processor, P_i, stores the tree (without catalogs) for T_0 and the complete tree, with catalogs, for T_i.

The catalogs for T_0 are distributed. Since the point location for subtree T_i will

be a sequential operation, we do not need to distributed its catalogs.

The segment tree is constructed as follows:

Algorithm 12: PNTLOC_SEGTREE_CONSTRUCT()

1 Sort endpoints of input segments by x coordinates. Each processor now has its $S_i \subseteq S$ set of line segments and the x-interval of its T_0 segment tree leaf.

2 Share globally the x-intervals of all T_0 leaves, and let each processor independently compute T_0 (without catalogs).

3 Each processor computes the catalogs of T_0 for its set of line segments, S_i.

4 Globally sort the catalog segments using the catalog's node as primary key, and then as secondary key use the segment's y-coordinate at the node's left interval boundary.

Once the parallel segment tree is constructed, we may perform a point location query on a set Q of n points.

Algorithm 13: PNTLOC_SEGTREE_QUERY()

1 For each query q, create a copy destined for each of the $O(\log p)$ nodes of T_0 which must be searched.

2 Perform binary search (Algorithm 9) on the query copies in the sorted catalog list.

3 Determine how many queries are destined for each processor's T_i subtree and load balance (Algorithm 8) the queries in the subtrees.

4 Each processor performs sequential next element search for its queries received in the previous steps.

5 Gather queries from steps 3 and 5, using global sort by query and reduce to find the final answer for each query.

Let us briefly analyse this algorithm. Computing in $O(\frac{n \log n}{p})$ time is sufficient for the construction algorithm, noting that the sort in Step 4 may be completed by an integer sort. The query algorithm is bounded by $O(\frac{n \log n \log p}{p})$ where the binary search of Step 2 performs a search for each of the $O(n \log p)$ query copies created in Step 1.

Theorem 2 ([9]) *The Next Element Search problem can be solved on a CGM with* $O(\frac{n \log p}{p})$ *global memory in* $O(\frac{n \log p \log n}{p})$ *local computation time and* $O(1)$ *communication rounds, given* $n \geq p^3$.

2.4.2 Parallel Red-Blue Line Segment Intersection

Overview

Another subproblem useful for a CGM Hausdorff Voronoi algorithm is the red-blue line intersection problem – given a set of non-intersecting lines (the "red" set) and another set of non-intersecting lines (the "blue" set), the problem is to find all the intersections between the red and blue line segments.

There exist two common variations of the red-blue line intersection problem. The first is to only *count* the total number of intersections, while the second is to explicitly *report* all of them. We require the use of something in between. For each line, we must *search* its intersections to determine one intersection closest to a certain point along the line, and only report this intersection. We will describe our algorithm for this search variant of the problem. Although reporting all intersections and then searching these would be correct, it may be time-consuming, as its complexity is dependant on the number of intersections to report. There

exist $O(n^2)$ intersections in the worst case, so we develop a faster method which avoids the time to perform a complete reporting of all intersections between red and blue segments.

Hereditary Segment Tree

Chazelle et al [11] describe a solution to the red-blue intersection problem with an algorithm that uses a *hereditary segment tree.* The tree is explicitly constructed, and by pre-sorting the segments, using fractional cascading, and using streaming, good bounds are maintained [11]. Namely, the intersection counting algorithm runs in $O(n \log n)$ time in $O(n)$ space. Palazzi and Snoeyink provide a more practical algorithm in [31] which does not actually build the *hereditary segment tree* explicitly. However, we do not discuss this algorithm, because for our search variation of the problem we modify the algorithm of Chazelle et al. A hereditary segment tree is similar to the standard segment tree used in point location. The red and blue endpoints' x-coordinates are ordered, forming a partitioning of x-intervals. Each interval is associated, in order, with the leaves of a balanced tree, and inner nodes are given the union of their two children's x-intervals. Along with the interval, each node stores four catalogs, two red and two blue.

A red segment is in a node n_i's long red catalog iff the segment completely spans n_i's x-interval, but not the x-interval of n_i's parent. The same segment is stored in the short red catalog of every node n_j that is a proper ancestor of n_i. Long and short blue catalogs are populated similarly. Conceptually, each node's x-interval represents a vertical strip. A (e.g. red) long catalog may be sorted vertically in such a strip, since one colour's segments do not intersect. To

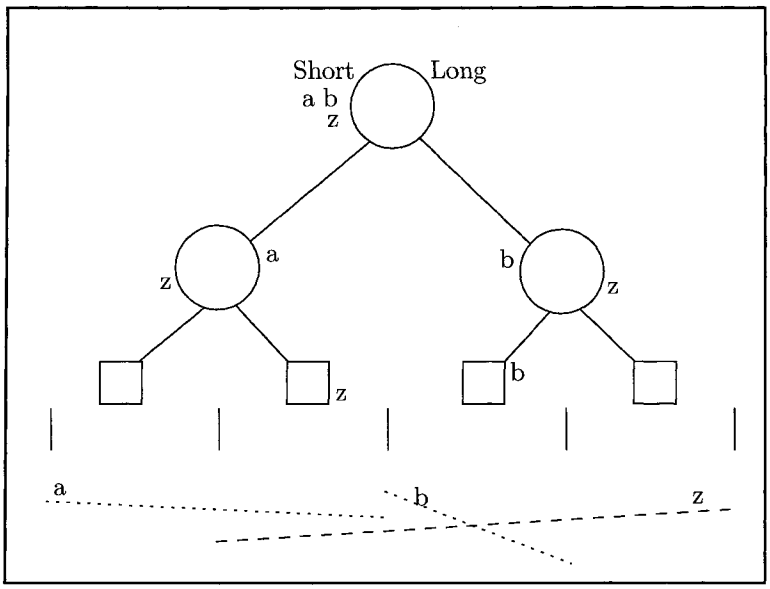


Figure 2.6: Example Hereditary Segment Tree

find intersections, we compare, at each node, pairs of red and blue catalogs. We compare a node's two long catalogs, as well as a short with a long. Note that we always compare with a long, which has a total ordering in the node's vertical strip.

We briefly describe how we may guarantee that each intersection uniquely exists in these comparisons. Let us assume that there is an intersection between some blue segment e_b and red segment e_r. Then this will occur in the vertical strip of exactly one leaf, and we look at the nodes from this leaf to the root. There are three possibilities. The intersection may occur in a node on this path where both e_b and e_r are long, and we will not find it elsewhere. Otherwise, the two cases are symmetrical; assume e_b is long above the node in which e_r is long. Then where e_r is long, e_b will not be stored, so the intersection will not be counted there. Where

e_b is long, however, e_r is short, and the intersection will be counted.

Secondary Tree Extension

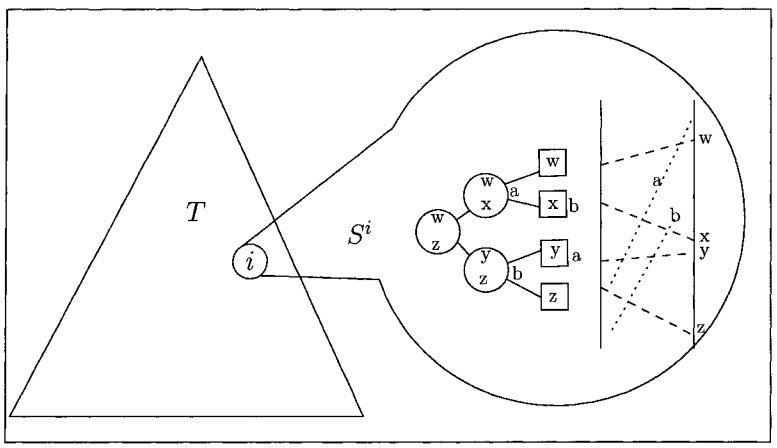


Figure 2.7: Example Secondary Tree

We must now describe the algorithm to solve our intersection search problem. Let the red segments be our queries. That is, for any red segment, e_r, we wish to search for some blue segment e_b which intersects e_r directly above some point. The exact search criteria for two slightly different queries are given in a later chapter. Here, we assume there is some total ordering on a red query segment's intersections that allow us to perform binary search.

We must search e_r against the long blue catalog at nodes where e_r is short. We must also search e_r against the long and short blue catalogs at nodes where e_r is long. Unfortunately, we cannot order a node's short blue catalog, making it difficult to efficiently search e_r against a short blue catalog. However, we may use a secondary structure to extend the hereditary segment tree, proposed by

Chazelle et. al [11]. The purpose of the secondary structure is to arrange a node's catalog lists into pairs of sublists such that all red and blue segments in a pair intersect. Specifically, we create this secondary structure for long red and short blue segments at each node.

Taking the (vertically ordered) long red segments in the node's vertical slab, we construct another balanced tree where the long red segments are assigned, in order, to the secondary tree's leaves. Each inner node receives the union of its subtree's long red segments. These intervals of red segments are analogous to the x-intervals of the main segment tree.

This secondary tree's catalogs are populated with short blue segments. For each blue segment, we locate its endpoints in the long red sequence, with which we may determine the interval of nodes that the short blue segment intersects. A short blue segment is placed in a secondary node, s_i, exactly when the blue segment intersects all of s_i's long red segments, but not all of s_i's parent. Note that each blue segment is stored in at most $O(\log k)$ of the k nodes in the secondary tree. Finally, once all short blue segments are stored, we may order the short blue catalog at each node by the order in which they cross the long red segments.

Intersection Search Variant

To query the tree for some red segment, e_r, we traverse the segment tree, looking at the $O(\log n)$ nodes where e_r is stored in some red catalog. Regardless of whether e_r is short or long, we may locate its endpoints in the long blue list. When e_r is long, we also traverse the secondary structure, searching the secondary nodes' blue catalogs along the path where e_r is stored. For every ordered catalog of blue

segments we find, we may perform a binary search.

Hence, the running time of the sequential red-blue intersection search problem is dominated by the time to search the secondary trees. The secondary trees are of total size $O(n \log^2 n)$, so both the sorting and querying of the secondary trees' short red catalog requires a total of $O(n \log^3 n)$.

Algorithm 14: RB_SEGTREE_SEQUENTIAL()

1. Sort the x-coordinates of all red and blue segment endpoints. Create a balanced binary tree on these elementary intervals
2. For each red and blue segments traverse the tree and insert the segment into the appropriate long and short catalogs. Order the long catalogs
3. Search each short red catalog's segments in the corresponding long blue catalog
4. Search each long red catalog segment in the corresponding short blue catalog
5. For each node, construct a secondary tree skeleton on the long red catalog, and insert the short blue catalog segments. Order each secondary catalog.
6. For each secondary node, search the node's long red segments in the short blue catalog list.
7. Reduce the results from steps 3, 4, and 6 into a unique result for each red segment.

Theorem 3 *The red-blue line intersection search problem may be solved in* $O(n \log^2 n)$ *space and* $O(n \log^3 n)$ *time.*

CGM Search Algorithm

CGM algorithms for both the counting and reporting variants of the red-blue line intersection problem are given by Devillers and Fabri [18]. Again, we require a search variation, rather than the usual counting or reporting, and hence give appropriate modifications. For the parallel algorithm, we make use of load balancing techniques similar to those used by Devillers and Fabri, although we have a larger, more complex data structure on which to operate.

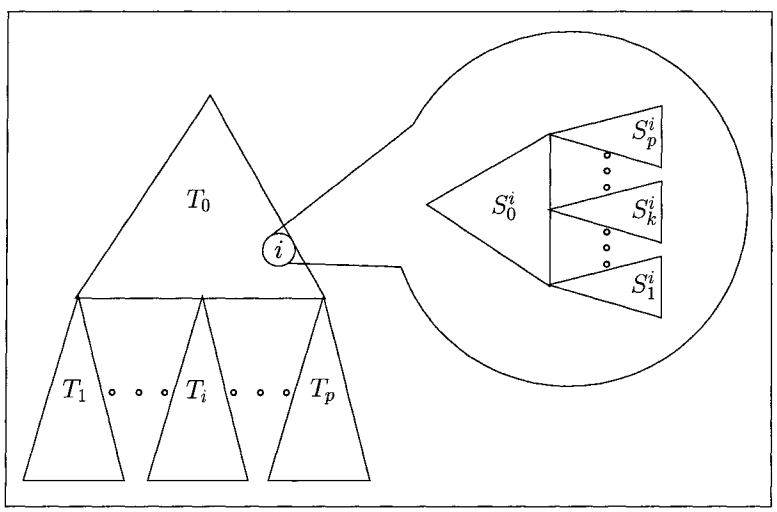


Figure 2.8: Extension to the Hereditary Segment Tree Data Structure.

To distribute this new data structure across processors, we divide the main segment tree into a shared piece, T_0 of the first $O(\log p)$ levels, and into P subtrees $(T_1, \ldots T_P)$. Figure 2.8 show an example data structure. For each node of T_0, we may share the linear long blue catalogs using a global sort, similarly to the scheme for the standard CGM segment tree of Section 2.4.1. We concentrate on describing how to process and balance the root piece's secondary trees. The secondary tree,

S^i, for each segment tree node i is also divided into a top portion, S_0^i, of size p, and a set of subtrees, $S_1^i, \ldots S_p^i$.

Let us first discuss the top S_0^i secondary trees. These trees' catalogs can be concatenated into a global sequence, sorted by key (n_j, s_k, e_i), where n_j is a segment tree node, s_k is a secondary node in n_j's secondary tree, and e_i is a short blue catalog entry at s_k.

Then we can determine for each long red segment e_i, the $O(\log^2 p)$ secondary catalogs (s_k, in a segment tree at node n_j) that need to be searched. A copy of e_i with the key (n_j, s_k, e_i) may be created, and then we may perform the binary search algorithm of Section 2.2.5 to perform the query for S_0^i tree tops.

Each processor stores a set of the S_k^i secondary subtrees. We may use the loadbalancing algorithm (Algorithm 8) to balance out the subtrees and the short blue segments destined for each subtree, and then solve each sequentially.

The main segment tree's subtrees do not require loadbalancing, since the nodes' intervals are based on red and blue segment endpoints. Hence, $O(\frac{n}{p})$ queries are distributed to each T_i subtree.

Algorithm 15: RB_SEGTREE_BUILD()

1 Global Sort endpoints of input segments (both R (the red queries) and B) by x coordinates. Each processor now has its $R_i \subseteq R$ and $B_i \subseteq B$ sets of line segments and the x-interval of its T_0 segment tree leaf.

2 Share globally the x-intervals of all T_0 leaves, and let each processor independently compute T_0 (without catalogs).

3 Each processor computes the long and short, red and blue catalogs of T_0 for its line segments.

4 Global Sort long red catalogs, and locally construct S_k^i subtrees for the catalog portions received at each processor

5 For each long red catalog, n_i, share globally the S_k^i intervals (i.e. S_0^i leaves), and construct S_0^i.

6 For each processor and at each node n_i, insert the local short blue catalog into S_0^i. Globally sort the catalogs of all S_0^i.

Algorithm 16: RB_SEGTREE_SEARCH()

1 At each processor, traverse T_0 and create a query (n_j, r_i) for each red segment r_i in the node n_j's short red catalog. Binary Search (Algorithm 9) these queries in the global long blue catalog list.

2 At each processor, traverse T_0 and create a query (n_j, r_i) for each red segment r_i in the node n_j's long red catalog. Binary Search (Algorithm 9) these queries in the global long blue catalog list.

3 Taking the same set of queries as above, binary search (Algorithm 9) these queries in the global short blue catalog list.

4 Determine in which secondary subtrees S_k^i each query $r \in R_i$ must be searched.

5 Perform loadbalancing (Algorithm 8) to group a constant number of S_k^i subtrees (i.e. the subtree's set of blue segments) with a set of queries.

6 Perform sequential algorithm on each processor's secondary subtrees and queries received in the previous step. (Algorithm 14)

7 Perform sequential algorithm on each processor's T_i segment subtree. (Algorithm 14)

8 Globally reduce the results from Steps 1,2,3,6,7 to a unique result for each red query segment.

Analysis

The most complex portion of T_0 is the secondary catalog querying. The upper S_0^i portions of the secondary trees reduce to sequential batch binary search sub-

problems of total size $O(\frac{n \log^2 p}{p})$, which require $O(\frac{n \log n \log^2 p}{p})$ local computation. The lower S_k^i portions of the secondary trees and the secondary trees in T_i reduce to sequential subproblems of total size $O(\frac{n \log^2 n}{p})$ per processor, which require $O(\frac{n \log^3 n}{p})$ local computation. Hence,

Theorem 4 *The red-blue line segment intersect search problem may be solved on a CGM in* $O(\frac{n \log^2 n}{p})$ *space and* $O(\frac{n \log^3 n}{p})$ *local computation time, with* $O(1)$ *rounds, and the restriction that* $n \geq p^3$.

S

Chapter 3

Hausdorff Voronoi Diagram Review

3.1 Definition and Properties

The Voronoi diagram, introduced in Section 2.1, is a fundamental data structure in computational geometry. The diagram is a subdivision into regions closest to each input site. The Voronoi diagram of points is constructed for input sites that are single points in the plane, and closeness to such an input site is defined by the Euclidean metric.

Many generalisations of the Voronoi diagram's space, site and metric properties have been studied (e.g. [5]). Our particular diagram, the Hausdorff Voronoi diagram, generalises sites to sets of points in the plane. The metric, while still based on Euclidean distance between individual points, must be modified to precisely define distance to a point set. Let us now define the Hausdorff metric and introduce the Hausdorff Voronoi diagram.

A Hausdorff Voronoi diagram is constructed for a set system with a universe I of *input* points. A subset of the power set of I, $S = \{P_1, P_2, \ldots, P_m\}$, is given as input, such that $\bigcup_i P_i = I$ and $P_i \bigcap P_j = \emptyset$. We let the input size, $n = |I|$.

For a $P_i \in S$, we use the terms *shape*, point set in S, and *site* interchangeably. We say *input point* to mean a point $p \in P_i$ for some shape $P_i \in S$. We can assume that all input points in a set P_i are vertices of the convex hull $CH(P_i)$. We will see that points internal to the hull do not affect the Hausdorff Voronoi diagram.

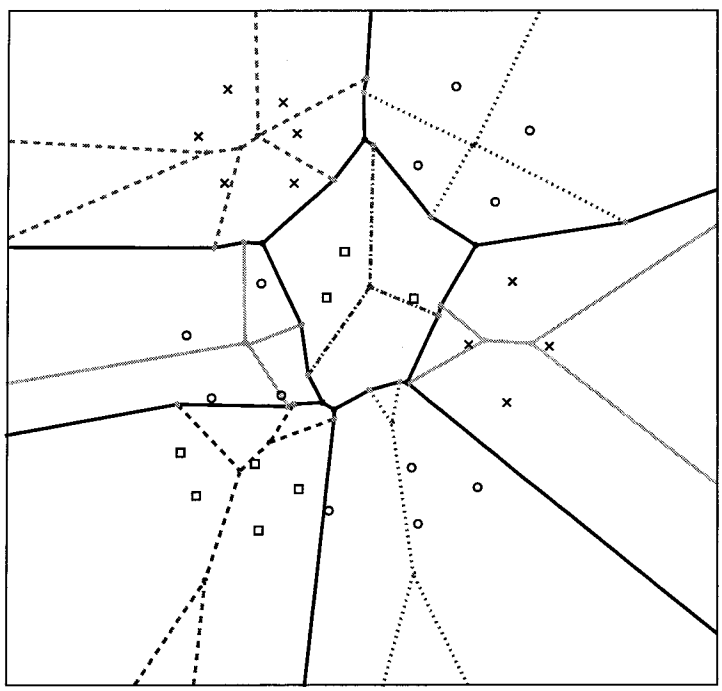


Figure 3.1: An example Hausdorff Voronoi diagram

Atallah [3] provided an early example of the Hausdorff metric in computer science by providing a linear time algorithm for computing Hausdorff distance between point sets. The Hausdorff metric has originally been defined between two point sets, say A and B. The directed Hausdorff distance from A to B is denoted $\overrightarrow{d_h}(A, B)$. The undirected Hausdorff distance between A and B is $d_h(A, B)$.

Definition 10 (Hausdorff Distance) *The undirected Hausdorff distance is* $d_h(A, B) =$

$\max\{\overrightarrow{d_h}(A,B), \overrightarrow{d_h}(B,A)\}$ *where the directed Hausdorff distance is*
$\overrightarrow{d_h}(A,B) = \max_{a\in A}\{\min_{b\in B}\{d(a,b)\}\}$

However, we note that for Hausdorff Voronoi diagram computation, we only apply the distance function between sites and arbitrary points, z, in the plane. It has be shown [37] that $d_h(A,\{z\}) = d_f(A,z)$ where d_f is farthest Euclidean distance. For clarity, we continue to use d_h to denote the distance between a site and a point in the plane. Since Hausdorff distance deals with farthest points in a point set, we assume for convenience that only the vertices of a point set's convex hull are included in a $P_i \in S$.

There is one other property of the input that merits discussion. In the literature, it is possible to allow two input shapes to be *crossing.* For our discussion, we limit our input set S to be *non-crossing*, i.e. no two shapes $P_i, P_j \in S$ are crossing. As mentioned in [37], a consequence of a non-crossing S is that a Hausdorff Voronoi region is connected (See Lemma 3 below). Note that although shapes cannot cross, there is no restriction against overlapping shapes. Intuitively, two shapes P_i, P_j are crossing iff the region of $CH(P_i) - CH(P_j)$ is disconnected. A more formal definition is as follows:

Definition 11 (Crossing) *Two shapes, $P_i, P_j \in S$ are said to be crossing iff there exist two vertices p_i, p_j on P_i's convex hull and q_i, q_j on P_j's convex hull such that (1) $\overline{q_iq_j}$ intersects $\overline{p_ip_j}$ and (2) all of p_i, p_j, q_i, q_j are on the convex hull of $P_i \bigcup P_j$.*

From now on, we assume that no two shapes are crossing. Now that we have characterised our generalisations of the input sites and distance metric, let us

provide the usual definition of a Hausdorff Voronoi diagram with respect to these more general properties:

Definition 12 (Hausdorff Voronoi Edge, e) *A Hausdorff Voronoi Edge, e, is the locus of points with exactly two closest (under Hausdorff metric) points in the input set I.*

Definition 13 (Hausdorff Voronoi Vertex, v) *A Voronoi Vertex, v, is a point with at least three closest (under Hausdorff Metric) points in the input set S.*

Definition 14 (Hausdorff Voronoi Set Region) *A Hausdorff Voronoi Region for a shape $P_i \in S$ is $HReg(P_i) = \{z \in \mathbb{R}^2 | d_h(z, P_i) < d_h(z, P_j), \forall P_j \neq P_i\}$. By definition, $hreg(P_i) = \bigcap_{P_j \in \mathcal{S}, P_j \neq P_i} H(P_i, P_j)$.*

Definition 15 (Hausdorff Voronoi Point Region) *A Hausdorff Voronoi Region for an input point $p \in P_i$ is $hreg(p) = \{z \in \mathbb{R}^2 | d(z, p) = d_h(z, P_i), d_h(z, P_i) < d_h(z, P_j), \forall P_j \neq P_i\}$.*

Definition 16 (Hausdorff Voronoi diagram) *Given a set S of point sets (sites), the Hausdorff Voronoi diagram, $HVor(\mathcal{S})$, is the union of* Hausdorff Voronoi Edges *and* Hausdorff Voronoi Vertices. *See Figure 3.1 for an example Hausdorff Voronoi diagram.*

For brevity, where it is unambiguous, we will frequently call Hausdorff Voronoi components as just Voronoi components (e.g. Voronoi Edge instead of Hausdorff Voronoi Edge). Note that a Voronoi Edge has exactly two closest points in S, which we call the *inducing* points. The two inducing points could be from the

same point set (i.e. a point set with two equidistant farthest points), or the two points are from different point sets. If the inducing points are from the same point set, the edge is said to be an *intra-edge*. Otherwise, it is an *inter-edge*.

Likewise, a Voronoi Vertex has (in general position) exactly three closest points, which we call the *inducing* points of the Vertex. If all inducing points are from the same point set, then the vertex is an *intra-vertex*. If all three points are from different point sets, then the vertex is an *inter-vertex*. If two points are from the same set, and the third from a different set, then the vertex is a *mixed-vertex*.

We also make use of another convenient notation, that of the *Furthest Voronoi Tree*. The Furthest Voronoi diagram, FVOR(P_i) for some point set P_i is well known. The regions of $FVOR(P_i)$ are divided by an acyclic, connected collection of edges and vertices. Given some arbitrary point on an edge in FVOR(P_i) as a root, we can view the diagram's edges as a tree. This is the Furthest Voronoi Tree:

Definition 17 (Furthest Voronoi Tree) *The Furthest Voronoi Tree $\mathcal{T}(P_i)$ is the tree formed by the edges of $FVOR(P_i)$. Unless otherwise noted, we choose the root of $\mathcal{T}(P_i)$ to be the centre of the minimum enclosing circle of P_i.*

We have now defined the Hausdorff Voronoi diagram and some related notation. To conclude this section, we provide lemmas which will be useful throughout our ensuing discussion of Hausdorff Voronoi diagram construction algorithms.

Lemma 1 ([38]) *The region $hreg(p)$ is essentially star-shaped. That is, for any point $x \in hreg(p)$, $\overline{px} \bigcap freg(p)$ is contained in $hreg(p)$.*

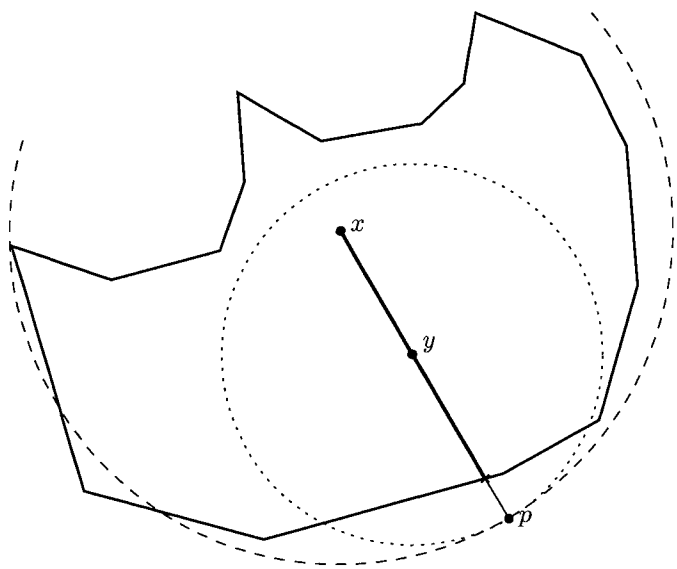


Figure 3.2: Region is Essentially Star Shaped

Proof. Let $x \in hreg(p)$ for some $p \in P_i$, let y be a point on $\overline{px} \bigcap freg(p)$, and let $y \in hreg(q)$, $q \in P_j \neq P_i$ (see Figure 3.2). Let $\mathcal{C}(y)$ be a circle centred at y of radius $d(y,p)$ and let $\mathcal{C}(x)$ be a circle centred at x of radius $d(x,p)$. We know P_j is completely contained in $\mathcal{C}(y)$. However, $\mathcal{C}(y) \subset \mathcal{C}(x)$, hence P_j must be completely contained in $\mathcal{C}(x)$, a contradiction. □

We now describe *limiting shapes*, a key geometric fact which is useful in characterising properties of Hausdorff Voronoi diagrams. Informally, if some shape P_j is *limiting* on a shape P_i's farthest tree, $\mathcal{T}(P_i)$, then one continuous portion of $\mathcal{T}(P_i)$ is closer to P_j. More formally, given some shape, $P_i \in S$, we define $C_v(P_i)$ to be the circle centred on a point $v \in \mathcal{T}(P_i)$, with radius $d_h(v, P_i)$. Note that this circle must have at least two input points, $p_i, p_j \in P_i$ on its border. Given these constructions, we may now clearly define a limiting shape:

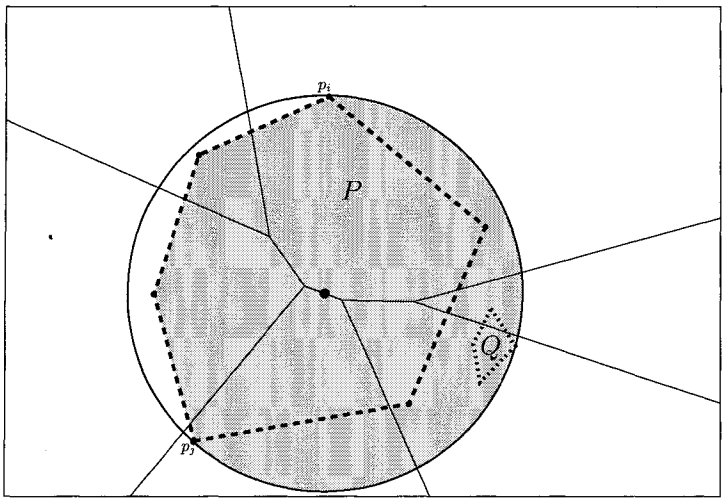


Figure 3.3: Limiting Shape

Definition 18 (Limiting Shape) *For some shape P_i, the shape P_j is a limiting shape at point $v \in \mathcal{T}(P_i)$ if P_j is completely contained in $C_v(P_i)$*

We are interested in limiting shapes because they determine portions of $FVOR(P_i)$ that are not contained in $HVor(S)$. We categorise a limiting shape P_j based on which side of P_i it lies on. A point v divides $\mathcal{T}(P_i)$ into two pieces, one which contains the root, $\mathcal{T}'_v(P_i)$, and one which does not, $\mathcal{T}_v(P_i)$. The chord $\overline{p_i p_j}$ divides $C_v(P_i)$ in two. The forward portion, denoted by $C^f_v(P_i)$, induces edges in $\mathcal{T}'_v(P_i)$, and the reverse portion, $C^r_v(P_i)$ induces edges in $\mathcal{T}_v(P_i)$. So a shape P_j is forward limiting if $P_j \subset C^f_v(P_i) \bigcup \mathrm{CH}(P_i)$. Conversely, P_j is rear limiting if $P_j \subset C^r_v(P_i) \bigcup \mathrm{CH}(P_i)$.

Lemma 2 ([38, 34]) *If P_j is rear limiting on P_i at point $v \in \mathcal{T}(P_i)$, then all of $\mathcal{T}'_v(P_i)$ is closer to P_j than P_i. Conversely, if P_j is forward limiting on P_i, then all of $\mathcal{T}_v(P_i)$ is closer to P_j than P_i.*

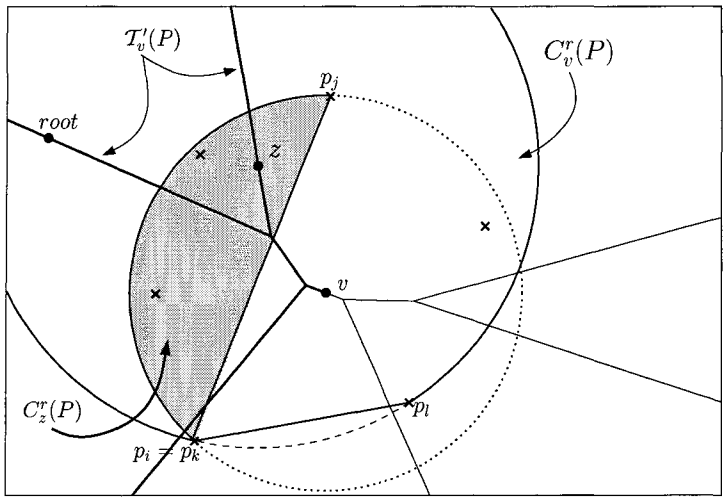


Figure 3.4: Proof of Rear Limiting Shape

Proof. We only prove for the first case. For some point $z \in \mathcal{T}_v'(P_i)$, let p_k, p_l be the sites inducing z's edge. It is clear that $C_z^r(P_i) \subset C_v^r(P_i)$, since $C_z(P_i)$ is a circle containing P_i, $p_i, p_j \in C_z^f(P_i)$, and $C_z(P_i)$ intersects $C_v(P_i)$ at most twice. Since $P_j \in C_v^r(P_i) \bigcup \mathrm{CH}(P_i)$, we conclude that $P_j \in C_z^r(P_i) \bigcup \mathrm{CH}(P_i)$. □

Let us end this section with one final, but important lemma:

Lemma 3 (based on [38]) *The region hreg(p) is connected, and the size of HVor(S) is* $\mathrm{O}(n)$.

Proof. Let v_m be some mixed vertex induced by $q \in P_j$ and $p_i, p_j \in P_i$. Then $v_m \in \mathcal{T}(P_i)$ and P_j is limiting on P_i at v_m. on the chain of edges bordering some $freg(p)$, there may be at most two mixed vertices, and the $freg(p)$ edges that remain in $hreg(p)$ must form a connected chain. Hence, $hreg(p)$ itself must also be connected. Similarly, $HReg(P_i)$ must also be connected. The entire diagram,

$HVor(S)$, is a planar graph with at most n faces. Therefore, $HVor(S)$ is $O(n)$.

□

We have introduced the Hausdorff Voronoi diagram and discussed some of its interesting properties. Let us continue on to discuss existing sequential algorithms to construct the Hausdorff Voronoi diagram. There exists a sweepline algorithm, described in Section 3.2 and a divide and conquer algorithm in Section 3.3. Also, there is an elegant and well-known mapping of Voronoi diagram input to planes in three dimensions [17]. Let us briefly review the computation of three-dimensional envelopes and their relation to Hausdorff Voronoi diagrams.

To compute the Hausdorff Voronoi diagram using a mapping to three-dimensional arrangements, we may first compute the farthest Voronoi diagram of each point set. This may be accomplished by mapping the points to planes tangent to the unit paraboloid directly above the points. The lower envelope of these planes will project down to the farthest Voronoi diagram. To obtain the Hausdorff Voronoi diagram, we may then take the lower envelope for each point set and compute their upper envelope, thus minimising distance between point sets. An algorithm for computing envelopes of piecewise linear functions was given in [16].

The piecewise linear envelope algorithm performs divide-and-conquer on the set of triangles obtained by triangulating the lower envelopes. At each merge step, the upper envelopes of two subsets of triangles have been computed. The original triangles in each subset are projected to the plane, and then the arrangement of the two projections is taken. Clipped to the boundary of each region in this arrangement are pieces of each sub envelope that contain no vertices. Thus, within each arrangement region, the subenvelopes may be merged by computing

the merged envelope as a convex envelope, similar to an envelope of planes. The algorithm requires $O(n^2\alpha)$ time. More details are given by the authors in [16].

3.2 Sweepline Algorithm

3.2.1 Overview

The sweepline algorithm from Section 2.1.3 has been extended to handle Hausdorff Voronoi diagrams. This Hausdorff Voronoi sweepline algorithm, introduced by Papadopoulou [34, 33] provides a more practical solution than the Hausdorff Divide-and-Conquer algorithm of Section 3.3. The sweepline algorithm is practical because it poses fewer special and degenerate cases, and is easier to implement correctly and efficiently. The sweepline algorithm computes the Hausdorff Voronoi diagram for non-crossing input in $O((n+K_a)\log n)$, where K_a measures the number of shapes which are enclosed other shapes' *anchor circles* (See definition in Section 3.2.3)

The Hausdorff Voronoi Sweepline algorithm follows the sweep paradigm which we introduced while discussing Fortune's Voronoi diagram sweepline algorithm. Conceptually, we sweep a vertical line, L, from left to right. Throughout the algorithm we maintain three structures. We maintain the partial Hausdorff Voronoi diagram to the left of the sweepline. Second, we store the beachline of partial Voronoi regions. Third, we keep a queue of events to the right of L.

Definition 19 (Beachline) *For a vertical sweepline L and a subset of shapes $S_L \subset S$ that are completely left of L, the beachline is the bisector $\mathcal{B}(S_L, L)$, in the*

Euclidean metric, using the definition of bisector from Section 2.1.2. The beachline is a chain of parabolic segments, similar to the beachline for the standard Voronoi sweep.

Events are points where the combinatorial structure of B changes. Events, as in Fortune's algorithm, do one of two things. At an event, either waves are added or deleted from B. Waves are deleted at spike events, and are handled essentially the same as the standard Voronoi case. Spike events still occur when three adjacent waves converge into two waves, forming a Voronoi vertex. In Fortune's algorithm new waves come into existence at *site events* that occur exactly on sites. However, for Hausdorff diagrams site events become more involved. First, let us define the priority of an event, and then we may clarify how new waves are added to the beachline.

Definition 20 (Priority) *Let $z = (z_x, z_y)$ be a point in the plane which is closest (i.e. in the region of) an input point $p \in P_i$. Then the priority of z is defined as $z_x + d(z, P_i)$.*

We may conceptually regard the mapping of events to priorities as a way to provide enough delay to the construction of the Voronoi diagram to the left of the sweepline. Once the sweepline has reached a point's (e.g. an event's) priority, we may be certain that any other closer site has already been handled by the sweepline. For each site's region, we require an event which starts the region. This should be the point in the region with the smallest priority. For Voronoi diagrams of points, this minimum priority event occurs at the site itself. In the

standard Voronoi diagram, all sites must have a non-empty region which at least contains the site itself.

The Hausdorff case is more complex. As described below, we again find a minimum priority event where the region for $\text{HVor}((P_i)), P_i \in S$ begins. The minimum priority event might not be included in $\text{HVor}(P_i)$. In fact, it is possible for a site to have an empty region. Once a region of $\text{HVor}(P_i)$ is started in the beachline, we must change active input points (and their waves) every time an edge of $\text{FVOR}(P_i)$ being traced in the beachline ends at an intra-vertex of $\text{FVOR}(P_i)$. Hence, we must additionally include "site" events for the intra-vertices of $\text{FVOR}(P_i)$).

To perform the Hausdorff Sweepline algorithm, we first precompute the Furthest Voronoi diagram, $\text{FVOR}(P_i)$, for all $P_i \in \mathcal{S}$, and retain $\mathcal{T}(P_i)$, the tree of $\text{FVOR}(P_i)$'s edges. Note that unlike the standard definition of $\mathcal{T}(P_i)$, in this section we let the tree be rooted at the *anchor point* of P_i (defined in Section 3.2.3). As the algorithm progresses, we handle three types of site events: *ordinary* site events, *mixed-vertex* site events and *minimum-priority* site events.

There is one minimum-priority site event for each point set in $\mathcal{S}$. This is a point, v_e, on $\mathcal{T}(P_i)$ which is first encountered by the sweepline (see Figure 3.5). This point, v_e, must have the minimum priority over all points on $\mathcal{T}(P_i)$. The first point encountered will contain P_i, and will be directly left of the rightmost point $p_r \in P_i$, lying on an edge induced by p_r and some other point p_i. When this minimum priority event is encountered, we determine whether v_e actually falls behind the sweepline. If it does, then by definition this point is in some other point set's Voronoi Region, and this minimum priority event is invalid. If it does

not, then the minimum priority event is valid, and we insert new waves at the event point to start tracing out the edges bordering P_i's region (including edges between input points inside P_i).

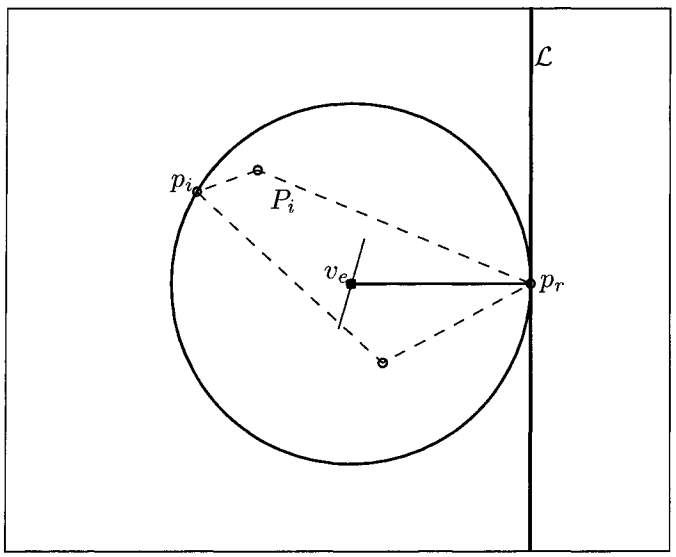


Figure 3.5: Handling a valid minimum priority event

In the case where P_i's minimum priority event was invalid, it is still possible that a later portion of FVOR(P_i) is valid and hence in the final Hausdorff diagram. So we must trace down the tree structure of FVOR(P_i), maintaining mixed-vertex events where P_i's regions could start. Note that at most one mixed-vertex or minimum-priority event will be valid for P_i.

Once a portion of FVOR(P_i) has entered the diagram and is active in the beachline, we must handle ordinary vertex events. These occur at the vertices of FVOR(P_i), and allow the individual regions of the points of P_i to be appropriately inserted into the Hausdorff diagram. These extra insertions ensure that we trace out the Voronoi Edges between points in P_i and adjacent shapes' regions, as well

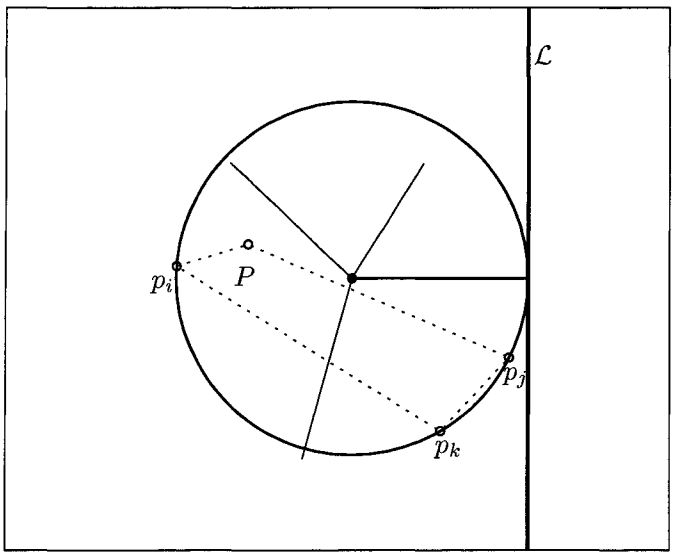


Figure 3.6: Handling a valid furthest vertex event

as Voronoi Edges between two points in P_i.

The portion of FVOR(P_i) included in the Hausdorff diagram ends when spike events close off all of P_i's waves that have been introduced by ordinary vertex events. These spike events will be between input points in multiple sets. Each spike event closes off a $hreg(p)$ for some input point p whose region is currently being traced.

We have now outlined, conceptually, the Hausdorff Voronoi diagram algorithm. We provide below a more precise, formal description of the algorithm, followed by a brief analysis of its running time.

3.2.2 Algorithm

Algorithm 17: HAUSDORFF_SWEEPLINE()

```
/* Initialisation                                                    */
foreach P_i ∈ S do
    Compute FVOR(P_i), the Furthest Voronoi diagram of point set P_i.
    For each vertex in FVOR(P_i), add a site event into the Event Queue.
    Determine the point of Minimum Priority of P_i, and add a site event for
    this point into the Event Queue.
end
/* Main Event Loop                                                   */
while Not Q.empty() do
    Pop next event, e. If e is a spike event, run SPIKE_EVENT().
    Otherwise, e is a site event. If e is invalid, then run INVALID_SITE().
    Otherwise, e is a valid site event. Handle the appropriate type of site
    event (HANDLE_MIN_PRIORITY(),
    HANDLE_FURTHEST_VERTEX(), HANDLE_MIXED_VERTEX()).
end
```

Algorithm 18: INVALID_SITE()

1 Site event e is associated with a point $v_i \in \mathcal{T}(P_i)$, and lies on some edge whose descendant vertex is v_j. If e is a furthest vertex event, then v_i happens to be a vertex, and there are multiple descendant edges ending at different v_j's.

2 **foreach** *direct descendant edge*(v_i, v_j), *in* $\mathcal{T}(P_i)$ **do**

3 **if** v_j *is right of the Beachline (i.e. valid)* **then**

4 Find the point v_t where an edge associated with v_i, v_j that intersects the beachline, and find the input point site, $q \in P_j$ owning the intersected beachline wave.

5 Intersect edges of $FVOR(P_j)$ with $\overline{v_t, v_j}$ to find the point in P_j which forms a mixed-vertex event at point $m_i \in v_i, v_j$.

end

end

Algorithm 19: HANDLE_MIN_PRIORITY()

1 The point v_e of a minimum priority event for shape P_i occurs on an edge induced by two points $p_i, p_r \in P_i$, with a priority such that the sweepline is tangent to P_i (i.e. $p_r \in L$). See Figure 3.5.

2 Search point v_e in the beachline, finding the wave's owner, q.

3 Split q's wave into two pieces, seperated by three new waves that trace out the edges between q and p_i, p_j, and both directions of the edge between p_i, p_j.

Algorithm 20: HANDLE_MIXED_VERTEX()

1 Find the wave of input point q which induces the mixed-vertex event. The mixed-vertex lies on the wave. Split the wave with two new waves for the other inducing input points p_i, p_j. This traces out the three edges emanating from the mixed vertex.

Algorithm 21: HANDLE_FURTHEST_VERTEX()

1 An ordinary event forks an existing edge into two new edges (See Figure 3.6). Search the beachline for the point where the waves for the two existing input points, p_i, p_j meet.

2 Create a third wave for the third input point, p_k that separates the two existing waves.

3.2.3 Analysis

This algorithm traces out the regions of the final HVor(S). Once input points from a shape have been initially introduced into the beachline, the ensuing tracing is straightforward. However, we have also ensured that the each shape, P_i, will be correctly added to the beachline by scanning through all possible starting points on the edges of $\mathcal{T}(P_i)$, until one is found. Let us now discuss the complexity of this algorithm.

Definition 21 *The* anchor circle, $C_a(P_i)$ *of an input shape* $P_i \in S$ *is the smallest circle centered at the* anchor point *of* P_i *that encloses* P_i. *Given a horizontal ray extending left from the rightmost point* p_r *of* P_i, *let there be a circle centered on this ray tangent to the sweepline which encloses exactly one shape. Let this*

enclosed shape be called the anchor shape. If P_i is the anchor shape, then the anchor point is P_i's point of minimum priority, otherwise the anchor point is the common ancestor on FVOR(P_i) (rooted at the point of minimum priority) of vertices of the bisector between P_i and the anchor shape.

We count the number of input shapes contained in other shapes' anchor circles. Let K_a^P be the number of input shapes contained in P's anchor circle. Then, let K_a be the sum of the number of shapes entirely enclosed in the anchor circle of each shape, that is $K_a = \sum_{P \in S} K_a^P$. Since the anchor circle of P encloses P, it can also enclose shapes contained in P. Clearly, it is possible to have $O(n)$ shapes contained inside $O(n)$ other shapes. Hence, we conclude that $K_a = O(n^2)$.

The parameter K_a characterizes the input's geometric configuration. Informally, this is a bound on the number of invalid minimum-priority events which will be generated during the sweepline algorithm. For a shape P_i, we generate a mixed-vertex event, using INVALID_SITE(), for only those shapes, P_j, contained in the anchor circle, K_a as these are the shapes which can be rear limiting on $\mathcal{T}(P_i)$, and hence are the shapes which may create mixed-vertex events on $\mathcal{T}(P_i)$. To find a mixed vertex, we merely scan through the intersections of $\mathcal{T}(P_i)$ with one edge of P_i. Hence, the total extra time to compute additional mixed-vertices is $O(K_a \log n)$, due to the $O(\log n)$ search of the beachline.

Theorem 5 ([20]) *The sweepline Hausdorff Voronoi diagram algorithm constructs a Hausdorff Voronoi diagram in* $O((n + K_a) \log n)$.

3.3 Divide and Conquer Algorithm

3.3.1 Algorithm

In [38, 37], Papadopoulou and Lee propose a Divide and Conquer algorithm for constructing Hausdorff Voronoi diagrams. It uses an algorithm similar to the Divide and Conquer for Voronoi diagrams of points from Section 2.1.4, except the Hausdorff version is complicated due to a more complex metric and input sites.

The merge curve, $\mathcal{B}(\mathcal{L}, \mathcal{R})$ is the bisector of the left and right point sets. For Hausdorff Voronoi diagrams, the merge curve may be comprised of multiple, disjoint merge components. As well, these components might not just be unbounded chains. Input shapes which cross a vertical left/right dividing line may cause merge cycles. See Figure 4.6 for an example. So the algorithm must determine a starting point on all merge components (chains and cycles). For each component, the algorithm performs a sequential walk from the starting point through the two Voronoi subdiagrams to find the complete chain.

Algorithm 22: HVOR-DIV-AND-CONQ()

1 Compute m, the median of the sites' leftmost points, for the input set S.
2 Partition S into a left and a right set, $\mathcal{L}$ and $\mathcal{R}$, i.e. $\mathcal{R}$ comprises sites completely right of $x = m$, and $\mathcal{L} = \mathcal{S} - \mathcal{R}$.
3 Recursively compute subdiagrams for $\mathcal{L}$ and $\mathcal{R}$.
4 HVOR-MERGE() subdiagrams into a merged Hausdorff Voronoi diagram for all of S.

The merge algorithm must find multiple unbounded merge chains. To find

these, we use the Hausdorff hull, which is analogous to the convex hull in the standard Euclidean metric. Merging Hausdorff hulls of the two subdiagrams into the hull for the merged diagram presents new Hausdorff hull edges, or *supporting segments.* Such supporting segments provide the two points which create an unbounded edge between $\mathcal{L}$ and $\mathcal{R}$, and hence a starting point for a merge chain.

Finding starting points for merge cycles is more difficult. After determining all unbounded chains, we have the Voronoi diagram for S, except some cycles may be missing. We need to determine these cycles. A cycle will enclose a portion of the diagram from one of the two subdiagrams. When a merge cycle is found, we need to swap which subdiagram's portion is contained in the enclosing cycle. To find these cycles, we need only consider the set of shapes, S_c, that are the shapes crossing the dividing line. To determine starting points for merge cycles, we scan through Voronoi edges whose inducing input points are in S_c. We check for cycles on these Voronoi edges by using point location on the edges' endpoints.

For unbounded and cyclic merge components, tracing occurs in a similar fashion to the standard Voronoi divide-and-conquer algorithm. To ensure good time bounds an input point's Voronoi region is decomposed such that each of the decomposition's regions may be visited no more than once by $\mathcal{B}(\mathcal{L}, \mathcal{R})$.

This decomposition is called the Visibility-Based Decomposition. An $hreg(p_i)$ is subdivided by segments $\overline{p_i v} \bigcap \mathrm{hreg}(p_i)$ for all vertices v bordering $hreg(p_i)$. By Lemma 5, each subdivision boundary may be crossed at most once. Hence, the overall merge walking may be performed in $\mathrm{O}(n)$.

Algorithm 23: HVOR-MERGE()

1 Compute the Hausdorff hull for S and determine supporting segments. (Algorithm 24).

2 Trace unbounded merge chains using starting points determined in the previous step.

3 Currently, the two subdiagrams may be partitioned into parts which are kept and parts which are removed. For all edges to be removed, mark all those not in S_c, leaving S_c unmarked.

4 **while** *unmarked edges exist* **do**

5 Take some unmarked edge, $e \in HVor(\mathcal{L})$. Locate e's endpoints in $HVor(\mathcal{R})$.

6 Using point location, determine whether e's endpoints should be kept or discarded.

7 If (at least part) of e is to be kept, scan through regions of $HVor(\mathcal{R})$ intersected by e, determining a merge cycle starting point, if any.

8 If a merge cycle exists, perform the chain trace.

9 Mark e.

end

3.3.2 Correctness

We review the correctness of the algorithm, paying particular attention to the details relevant to our parallel Hausdorff Voronoi divide-and-conquer algorithm. First, we must examine in more detail the exact shape and properties of the merge

components and their traversal through the two subdiagrams' Voronoi Regions.

Lemma 4 (based on [38]) *$\mathcal{B}(\mathcal{L},\mathcal{R})$ is a of set Voronoi Edges forming disjoint, unbounded chains and cycles, of total size* $O(n)$

Proof. The bisector $\mathcal{B}(\mathcal{L},\mathcal{R})$ is by definition the locus of points equidistant from a $l \in \mathcal{L}$ and a point $r \in \mathcal{R}$. Therefore, $\mathcal{B}(\mathcal{L},\mathcal{R})$ is composed of Voronoi edges of the l, r pairs.

Since $\mathcal{B}(\mathcal{L},\mathcal{R})$ is a subset of $HVor(S)$'s $O(n)$ edges (Lemma 3), it also must be $O(n)$.

Since the Hausdorff Voronoi diagram is a planar graph, edges cannot cross. Vertices are also disjoint, since there cannot be three inducing input points such that each pair are from opposite sides.

Components are either unbounded or cyclic. Otherwise, for a point a closer to $\mathcal{L}$ and a point b closer to $\mathcal{R}$, there exists a path between a, b that does not intersect $\mathcal{B}(\mathcal{L},\mathcal{R})$. However, some point on this path must be equidistant between $\mathcal{L}$ and $\mathcal{R}$, and hence be on $\mathcal{B}(\mathcal{L},\mathcal{R})$. □

Lemma 5 *For any point x in $hreg(p)$, the segment $\overline{px} \bigcap hreg(p)$ (region in subdiagram) is intersected at most once by $\mathcal{B}(\mathcal{L},\mathcal{R})$.*

Proof. By Lemma 1, a segment $\overline{px} \bigcap hreg(p)$ must be one continuous piece, one end being on an intra-edge. For this segment to be intersected more than once by merge chains would violate this property. □

Lemma 6 (based on [38]) *The only regions which may be enclosed by a cyclic component of $\mathcal{B}(\mathcal{L},\mathcal{R})$ are $hreg(P_i)$ for a shape P_i crossing the dividing line.*

Proof. Note that a cycle may enclose more than one shape.

Assume for contradiction that there is a cycle which encloses a Hausdorff region $hreg(P_j)$ such that P_j does not intersect the dividing line. W.L.O.G., let P_j be a shape on the left. Then P_j is completely left of any shapes in $\mathcal{R}$, and hence there exists a region left of P_j which cannot be closer to any shape in $\mathcal{R}$, so there can be no merge component left of P_j. Thus, the merge cycle cannot exist, a contradiction. □

Lemma 7 (based on [38]) *Cycles must enclose at least one intra-edge of* $freg(P_i)$.

Proof. Let $p \in P_i$ have a region $hreg(p)$ which is enclosed in a cycle. Due to Lemma 6, P_i must have at least two input points, therefore $freg(P_i)$ has edges. Also, if $hreg(P_i)$ is non-empty then there exists some point $z \in hreg(P_i)$. By Lemma 1, there is an edge of $freg(p)$ that is crossed $\overline{zp}$, and this must be an edge on the border of $hreg(p)$. □

For the entire algorithm to work, we must demonstrate that a starting point is found on each component of the $\mathcal{B}(\mathcal{L}, \mathcal{R})$, and that the walk will correctly compute a merge component.

For the walk, we create a *Visibility-Based Decomposition.* This is $hreg(p)$ decomposed by $\overline{vp} \bigcap hreg(p)$, for all vertices v of $hreg(p)$'s boundary. We know by Lemma 5 that each subdivision is entered at most twice (each decomposing segment is crossed at most once). Hence, we can perform the walk through $hreg(p)$ in linear time by using the V.B.D.

It remains to be discussed how we may find starting points for unbounded and cyclic merge components.

Merging Hausdorff Hulls

As in the well-established standard Voronoi divide-and-conquer, we can determine the unbounded starting edges by merging hulls. This will provide the supporting segments between the left and right hulls. For Hausdorff input shapes, we need a generalised type of hull, such as in Figure 3.7:

Definition 22 (Hausdorff hull) *For a set S of shapes, the* Hausdorff hull *(HH(S)) is a generalisation of the convex hull. An input point $p \in P_i \in S$ is on the Hausdorff hull iff there exists a line through p such that P_i lies entirely on one side of the line, and all shapes $S - \{P_i\}$ lies at least partially on the other side of the line. Two Hausdorff hull vertices, $p \in P_i, q \in P_j$ form a hull edge iff there is a line through p, q such that P_i, P_j lie entirely on one side of the line and $S - \{P_i, P_j\}$ lies on the other side of the line.*

Let us assume that we have a set S of shapes that has been divided into a left half, $\mathcal{L}$, and a right half, $\mathcal{R}$, according to left-most point. Also assume that we are given HH($\mathcal{L}$) and HH($\mathcal{R}$). Let us describe how to merge the two sub-hulls to determine HH(S). First, an overview of the Hausdorff hull merging algorithm, in which we determine the edges and vertices which remain valid (i.e. are in the merged hull):

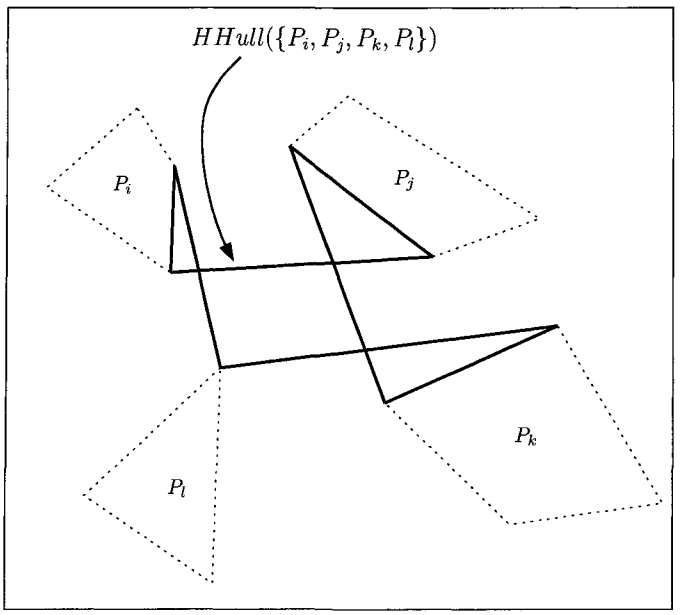


Figure 3.7: Example Hausdorff hull

Algorithm 24: Merge two Hausdorff hulls

1 Order the edges of HH($\mathcal{L}$) and HH($\mathcal{R}$) cyclically according to their slopes.

2 For each hull edge (e.g. $e \in$HH($\mathcal{L}$)), determine the opposite hull's adjacent edges in the slope-ordering (e.g. $e_r, e'_r \in$ HH($\mathcal{R}$)). In the slope-order, e is between e_r, e'_r.

3 e_r and e'_r border an unbounded region of $HVor(\mathcal{R})$, associated with some input point $r \in P_j$.

4 If e remains valid, then r must be inside HH(S). Note that we only need to check r with respect to e to determine this.

5 It is possible for two adjacent hull edges to be invalid, but their common hull vertex to be valid. For example, if r made some e invalid (as above), then r must be valid. Otherwise, r is invalid.

6 By keeping only valid hull edges and vertices, we may complete the merge, and find supporting segments between adjacent components in our cyclic order that come from opposite subdiagrams.

Finding Merge Cycles

A shape $P_i \in S_c$ will, by definition, cross the dividing line. Checking these edges for cycles is sufficient due to Lemma 6 and Lemma 7. For an edge between two input points, p_i, p_j we may perform point location, in $HVor(\mathcal{R})$. Note that both p_i and p_j are from sets in S_c, and their induced edge has been tentatively removed in the unbounded merge chain phase. Let such an edge be $e = \overline{y_i y_j}$, where $d_f(y_i, P_i) < d_f(y_j, P_i)$. After determining whether each endpoint is closer to the left or right, we may then analyse the possible cases:

1. y_i is closer to P_i. y_j closer to $Q_j \in \mathcal{R}$.

2. y_i, y_j closer to P_i. $\Rightarrow$ e is completely contained in a cycle.

3. y_i closer to $Q_i \in \mathcal{R}$ and y_j closer to $Q_j \in \mathcal{R}$. e remains entirely removed when

   (a) $Q_i = Q_j$

   (b) Q_i forward limiting

   (c) Q_j rear limiting

   (d) Q_j closer to y_i

   (e) Q_i closer to y_j

4. Same as previous, except none of five conditions hold. Then e is either intersected twice, or remains entirely removed.

We may briefly summarise the algorithm based on the above four cases:

For an edge of type 1, we may simply walk through the opposite diagram until we find a merge vertex on e.

For an edge of type 2, we may find a point on the merge chain as long as we always query an edge e adjacent to a component that we know is not in the cycle. Then we scan between the two components to find the merge chain.

For an edge of type 3, we know it is completely removed. Otherwise, the edge is of type 4, and we start traversing the edge from y_i and find the point where Q_j is equidistant from $\mathcal{L}$. Using another point location, we may determine if this equidistant point is in $Q_j's$ region, in which case we have found a point on a merge chain. Otherwise, we have removed part of the edge e, and found a new point which has been located in a different Q_j'. In this case, we repeat the step for an edge of type 3/4.

3.3.3 Analysis

Papadopoulou [37] provides worst-case bounds for the algorithm's running time based on parameters representing certain aspects of the input's geometric configuration:

- The measure K is the sum of the number of shapes entirely enclosed in the minimum enclosing circle of each shape.
- The measure M is the sum of the number of convex hull vertices of the shapes entirely enclosed in each shape's minimum enclosing circle..
- The measure N_d is the total number of input points in shapes in S_c throughout the algorithm. Recall that these are shapes intersected by a vertical

dividing line during some merge step.

The hull merge and merge component tracing both take time $O(n)$. It is finding merge cycles which dominates the running time complexity. Over all merge steps, there may be $N_d = O(n \log n)$ Voronoi edges to check, that is $O(n)$ per merge step. When looking for a cycle, we may generate additional candidate merge vertices for all shapes enclosed in the minimum enclosing circle of a shape, P_i. These shapes are exactly those that are rear limiting on any part of $\mathcal{T}(P_i)$, by Lemma 2. We perform a $O(\log n)$ point location search for each of the $K + n \log n$ candidate merge vertices. To determine a merge vertex for some shape P_j in P_i's minimum enclosing circle, we must scan the $|P_j|$ edges of $\mathcal{T}(P_j)$. Hence, we spend $O(M)$ to determine these vertices. In total:

Theorem 6 *The divide-and-conquer Hausdorff Voronoi algorithm constructs a Hausdorff Voronoi diagram in* $O(M + (n + N_d + K) \log n)$.

3.4 VLSI Applications

Voronoi diagrams are a fundamental structure in computational geometry whose study provides an intrinsic theoretical interest. They are, however, also applicable to a large array of application problems [5, 4, 30]. One such application problem is the computation of Critical Area in VLSI circuit design.

3.4.1 Critical Area Problem

Critical Area is one parameter required for calculating the yield of a VLSI design [34, 32]. The yield of a specific design is a prediction of the fraction of circuits which will be manufactured error-free. Critical area is the most complex parameter in yield calculation, so it is of particular algorithmic interest.

There is much active research on the broad area of yield prediction [34, 32]. We restrict our discussion to giving a brief overview highlighting the computational aspects of critical area calculations that are relevant to the design of our algorithm.

Critical area [32] is defined to be the area on a surface (of one layer) of a VLSI chip in which a defect would cause the chip to be defective, for some defect size. More precisely, critical area is

$$A_c = \int_0^\infty D(r)A(r)dr$$

where $D(r)$ is the density of defect of radius r, and $A(r)$ is the area comprising the locus of points where a circle centred of radius r would cause a defect. Known estimates exist for the density function $D(r)$ ([32, 36]), a typical example being

$$D(r) = \begin{cases} cr/r_0^2, & 0 \leq r \leq r_0 \\ cr_0^2/r^3, & r_0 \leq r \leq \infty \end{cases}$$

Voronoi diagram techniques have been studied which simplify the computation of $A(r)$ for three important types of defects. These spot defects can be caused by impurities, such as dust, on equipment or the chip's surface. Such defects may be

divided into two categories: missing material defects (*opens*), and extra material defects (*shorts*). See Figure 3.8 for a hierarchal diagram of defect types.

Applying Voronoi diagrams to the computation of Critical Area is efficient, deterministic and accurate. Other methods do exists, such as computing Critical Area using a regular or random (i.e. Monte Carlo) sampling of points, or other Geometric methods that compute A(r) independently for a sampling of radii [36]. Early Voronoi work centred on the L_∞ metric [35, 32], which allowed Voronoi Edges to be straight segments. However, this metric has been criticised for via blocks [37], and hence recent work has focused on the Hausdorff Voronoi diagram for this subproblem. The Hausdorff metric is based on the Euclidean metric, where a unit shape is a circle. Under L_∞, the unit shape is a square. Fortunately, the Hausdorff metric still allows for straight Voronoi Edges, and are therefore reasonably efficient to compute in practice.

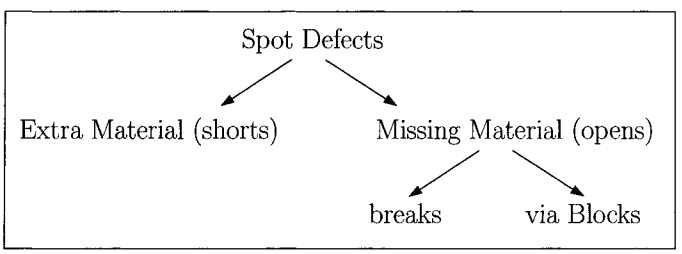


Figure 3.8: Summary of types of defects occurring a VLSI layer

3.4.2 Shorts

A short occurs when a defect causes two otherwise disconnected *shapes* (circuit components) in a circuit to be connected through additional material on a chip.

For a defect centred at a point c, one wishes to determine the point where the radius is at least as great as that of the distance to the two closest shapes.

In [36], Papadopoulou and Lee introduce their new technique of critical area calculation using Voronoi diagrams. Their algorithm for computing the critical area for shorts is a sweepline algorithm for computing the order-2 Voronoi diagram under the L_∞ metric. Input is restricted to a set of rectilinear polygons. The presented algorithm has a time complexity of $O(n \log n)$.

3.4.3 Breaks

In [32], Papadopoulou extends the application of Voronoi diagrams from shorts to missing material defects. Such defects include both break and via block defects. A break occurs when a component is severed into (at least) two disconnected pieces.

A sweepline algorithm is presented to compute the critical area for a set of rectilinear polygons under the L_∞ metric. The L_∞ medial axis of the polygons is required. The Voronoi diagram used is additively weighted; the distance from some point z to a component s of the medial axis is $d_w(z, s) = d(t, s) + w(s)$, where $d(t, s)$ is (closest) distance between s and z, while $w(s)$ is the distance from s to the associated polygon. Thus, a defect centred at z of radius $d_w(z, s)$ will disconnect the polygon associated with s. This algorithm has a time complexity of $O(n \log n)$.

3.4.4 Via Blocks

The original work [32] on Voronoi diagrams for via block defects used the L_∞ metric, which results in square defects as opposed to circles produced in the Euclidean metric. This original work limited input to rectilinear, disjoint polygons. Use of L_∞ for critical area computation has been criticised [37], so recent work [34, 37] has focused on providing improved solutions to the via block problem using the Hausdorff Voronoi diagram with Euclidean metric. A via is a connective component on a VLSI layer. Redundant vias are placed on a layer to improve reliability. To destroy the connection created by a via block, all its (redundant) vias must be destroyed. So for a defect centred at point c, one wishes to determine the smallest circular defect that will completely cover one via block.

We take the hulls of the redundant vias to be a via block. The original algorithm [32] used the L_∞ metric and allowed only disjoint hulls. This algorithm ran in $O((n+k)\log n)$ time. The algorithms we have discussed in this section use the Euclidean metric and allow for crossing shapes. Note that although via blocks cannot overlap, it is not impossible for two hulls to still have overlapping regions.

The construction of the Hausdorff Voronoi diagram that we are discussing is directly applicable to critical area computation of via blocks. Due to a VLSI design's large size, our parallelisation of the divide-and-conquer algorithm may be a useful step in improving the speed at which this critical area may be computed.

Chapter 4

CGM Algorithm for Hausdorff Voronoi Diagrams

4.1 Introduction

As a central contribution of this thesis we present a novel approach to constructing the Hausdorff Voronoi diagram in the CGM model of parallel computation. No known parallel algorithms exist for Hausdorff Voronoi diagram computation. Our algorithm is based on work by Diallo, Ferreira, and Rau-Chaplin [15]. They present a CGM algorithm for Voronoi diagrams of points, as discussed in Section 2.3. The CGM algorithm is based on results for a mesh algorithm developed by Jeong and Lee [25, 26, 27]. Although our divide-and-conquer algorithm uses a similar divide-and-conquer strategy to some previous parallel Voronoi algorithms, the more complex nature of the Hausdorff metric requires significant changes to the design.

Note that our algorithm assumes input shapes which are relatively small, as is appropriate for our VLSI application of critical area calculation for via blocks.

Formally, we limit the number of points of the convex hull of a shape to $O(\frac{n}{p})$. Thus we are able to assume that an entire shape may be completely stored at a processor. In addition, we restrict input to non-crossing shapes (see Definition 11). Although of theoretical interest, in practice, such crossings tend to be rare [34, 37]. As a consequence of non-crossing input, Voronoi Regions will always be connected (Lemma 3), and by Euler's theorem for planar graphs, the size of the diagram will therefore be bounded by $O(n)$.

Also note, however, that even with only non-crossing input shapes, there may still be multiple merge components. Figure 4.6 gives an example of two left/right subdiagrams which merge to form multiple merge components, both cycles and acyclic components. In addition, unlike in the case of standard Voronoi divide-and-conquer, a merge component is not necessarily y-monotonic.

Thus, we formally define our parallel Hausdorff Voronoi diagram problem to be the construction of HVor(S) for a set system (see Section 3) with $|I| = n$, $|P_i| = O(\frac{n}{p})$, $P_i \in S$, on a CGM with p processors. We present a novel solution to this parallel Hausdorff Voronoi diagram problem.

4.2 Algorithm Overview

Let us first discuss a broad overview of our CGM Algorithm. The basic top-level divide-and-conquer structure of this algorithm is similar to that of the CGM algorithm for Voronoi diagrams of points. (Algorithm 10).

Algorithm 25: CGM-HAUSDORFF()

1 Balance S across processors in sorted order. This may be accomplished using grouped sorting (Algorithm 7).

2 Divide the set S of input shapes into p equal vertical slabs, as determined above. Each processor receives a slab.

3 Each processor independently computes the Hausdorff Voronoi diagram of its slab's $O(\frac{n}{p})$ input.

4 HAUSDORFF-CGM-MERGE() (see below) combines $\frac{p}{2^i}$ diagrams into $\frac{p}{2^{i-1}}$ diagrams, such that two adjacent subdiagrams, $HVor(\mathcal{L})$ and $HVor(\mathcal{R})$ merge to form $HVor(S)$. Repeat this $\lceil \log p \rceil$ times, resulting in the final Voronoi diagram.

The dominating computation occurs in the merge stage. The standard Voronoi divide-and-conquer algorithm merges by first finding an end of the merge chain, and then constructing the entire merge chain in sequence by walking through the two subdiagrams. The sequential Hausdorff divide-and-conquer algorithm has a similar strategy, although a start point must be found on each of the merge components. Our Hausdorff CGM algorithm breaks apart the sequential walk by determining each vertex on the merge components independently. Point location on the endpoints of Voronoi edges of one subdiagram in the other subdiagram can be used to determine whether the subdiagram's edge is a part of a merge chain. The complete Hausdorff Voronoi CGM merge algorithm is presented in Algorithm 26.

Algorithm 26: HAUSDORFF-CGM-MERGE()

1 Use Point Location to find the subsets of Voronoi Edges crossing the merge chain. Let these subsets be, $E_m^l \subset E^l$ of left edges and $E_m^r \subset E^r$ of right edges. (Algorithm 27).

2 Use Red-Blue Line Intersection to find the merge chain vertices on edges in E_m^l and E_m^r. (Algorithm 28).

3 Remove edges (or portions of edges) which are not present in the merged Voronoi diagram.

4 Create a set of edge endpoints, two for each merge chain vertex. Global sort endpoints. Connect adjacent endpoints to form edges. (Section 4.5).

The task of the merge is to determine the new edges and vertices added to the merged diagram, and then determine which edges are removed partially or completely from the merged diagram. The new edges and vertices, which we term merge edges and merge vertices, form both unbounded acyclic merge components and cyclic merge components. These components partition the plane into two portions, those which retain edges from one (e.g. the left) diagram, and those which retain edges from the other (e.g. the right) diagram.

We concentrate our discussion on merge chain computation, as this is the critical portion of the merge algorithm. The classic sequential merge step for Voronoi diagrams of points uses an inherently sequential procedure. Using the convex hull of the two subdiagrams' input, supporting segments between the two subdiagrams' convex hulls may be found. These provide the extreme edges in the merge chain. From one such end, a walk may be performed through the

two subdiagrams, extending the merge chain and computing the merge vertices on the subdiagrams' edges. Sequential Hausdorff divide-and-conquer performs a similar step, except merge cycles may exist which cannot be discovered via a hull operation. Our algorithm is a conceptual simplification, as we do not require a special case to handle such merge cycles.

The sequential merge algorithm, unfortunately, presents dependencies between adjacent merge edges that makes parallelisation difficult. We need a way to decouple the computation of each portion of the merge chain. By using point location, we may treat each edge independently. Performing point location of edges' endpoints in the opposite subdiagram can be used to determine, for each edge endpoint, which subdiagram is closer. Determining the closer subdiagram is equivalent to determining on which the side of the merge chain an endpoint lies. Thus, we may determine, independently for each edge, those edges which cross the merge chain (edges to be cropped), those which lie on the far side of the merge chain (edge to be removed), and those which lie on the close side (edges to be kept).

After we have a subset of edges involved in the merge chain, we must determine where the merge vertices occur on these edges. Again, this may be done independently for each edge. Determining the merge vertex is equivalent to determining the input point from the opposite subdiagram which induces the merge vertex. However, we perform line intersection to determine the opposite subdiagram's edges which cross an edge. Conceptually, a parallel binary search through these edge intersections can then be used to determine the region of an input point inducing the merge vertex.

After determining the merge vertices, we sort, using a partial order, the vertices in such a way that vertices sharing an edge are adjacent. Constructing edges then becomes easy.

We proceed by discussing the details and correctness of each of these three stages, and then concluding this chapter with an analysis of our CGM Hausdorff algorithm's efficiency.

4.3 Point Location

Algorithm 27: HVOR-POINTLOC()

1 Construct a parallel standard segment tree for the right subdiagram's edges (Algorithm 12).
2 Query the left subdiagram edges' endpoints in the segment tree (Algorithm 13).
3 For each left edge, determine which side (left or right) each endpoint is closer to.
4 Create $E_1^l \subseteq E^l$, the left edges which have one endpoint closer to $\mathcal{L}$. Create $E_2^l \subseteq E^l$, the left edges with no endpoints closer to $\mathcal{L}$. All edges in $E^l - E_1^l - E_2^l$ are kept.
5 Repeat for the right edges in the left subdiagram.

The first step of Algorithm 26 is to perform point location on each set of subdiagram edges (left and then right), locating them in the opposite diagram. This subalgorithm is described in Algorithm 27. By doing so, we determine the closest input point in the opposite diagram to each end of a subdiagram edge. We may then easily compare and determine whether each end is closer to its own subdiagram or the opposite subdiagram. Since the procedure is symmetric for each subdiagram, we only discuss the point location of left edges in the right

subdiagram. We perform a case analysis on this information in a similar spirit to the parallel CGM algorithm for Voronoi diagrams of points:

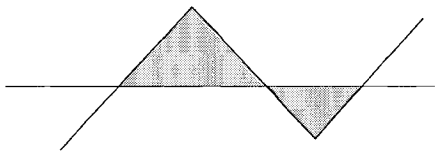


Figure 4.1: An edge cannot cross the merge chain more than twice.

Lemma 8 (based on [27]) *No edge is crossed more than twice by a merge chain.*

Proof. Recall that our Hausdorff input shapes are *non-crossing*, which implies that no disconnected regions exist (Lemma 3). Note how, as in Figure 4.1, having at least three intersections will necessarily divide a region into multiple disconnected components, which is contradictory. Therefore, an edge may be crossed no more than two times by a merge chain. □

Case 1:

Observation 1 ([27]) *An edge in $HVor(\mathcal{L})$ with one endpoint closer to $\mathcal{L}$ and the other closer to $\mathcal{R}$ will cross the merge bisector only once.*

Clearly, an edge with endpoints on different sides of the merge chain must cross the merge chain an odd number of times. By Lemma 8, the edge must cross exactly once.

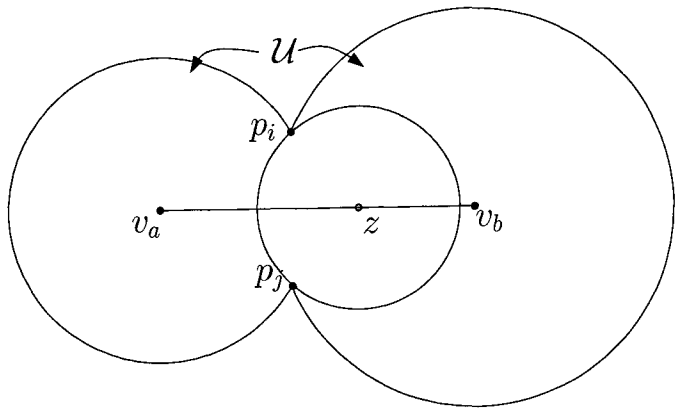


Figure 4.2: A Left edge with two close endpoints cannot contain crossings.

Case 2:

Lemma 9 (based on [27]) *An edge in $HVor(\mathcal{L})$ with both endpoints closer to $\mathcal{L}$ will not cross any merge chains.*

Proof. Let us demonstrate by a contradiction. Assume, for an edge with endpoints closer to $\mathcal{L}$, that some point, z, on the edge is closer to $\mathcal{R}$. Let there be two circles, tangent to the edge's two inducing points, p_i, p_j and centred on the edge's endpoints, v_a, v_b (see Figure 4.2). Let $\mathcal{U}$ be the union of the two circles. Note that no other input shape may lie completely in $\mathcal{U}$, by definition. Note also that for any point z along the edge, a circle centred at that point and tangent to the two inducing points is contained within $\mathcal{U}$. For z to be closer to $\mathcal{R}$, some set in $\mathcal{R}$ must completely lie in such a circle. But this is impossible. Hence, a merge chain cannot intersect an edge having close endpoints. □

Case 3:

Observation 2 ([27]) *An edge in $HVor(\mathcal{L})$ with both endpoints closer to $\mathcal{R}$ will cross the merge chain either 0 or 2 times.*

An edge with both endpoints on the same side of the merge chain must cross the merge chain an even number of times. Therefore, by Lemma 8, the merge chain must be crossed either 0 or 2 times.

Corollary 1 *Point location on edge endpoints can determine a subset of subdiagram edges, E_m^l from the left and E_m^r from the right, which intersect a merge chain. Each set may be subdivided, E_m^l (resp. E_m^r) into E_1^l (E_1^r), the edges intersected once, and E_2^l (E_2^r) the edges intersected two or zero times.*

4.4 Red-Blue Line Intersection

Algorithm 28: HVOR-RBINTERSECT()

1. Construct the extended hierarchal segment tree for the right subdiagram's edges (Algorithm 15).
2. Query the E_1^l and E_2^l in the segment tree to find merge vertices on these edges (Algorithm 16).
3. For edges in E_2^l, remove those for which no merge vertices were found. For all other queried edges, remove the appropriate portion of the edges.
4. Repeat for the right edges in the left subdiagram.

We have already determined the subdiagrams' Voronoi Edges to search for merge vertices. We now describe how this search may be viewed as a Red-Blue Line Intersection problem.

First, for simplicity, we restrict our attention to finding merge vertices on edges from $HVor(\mathcal{L})$ since the operation is symmetric for $HVor(\mathcal{R})$. Recall that we have determined a set, E_m^l, of edges to search for such merge vertices. Also, E_m^l is partitioned into two classes, the set E_1^l of edges intersecting the merge chain once, and E_2^l, the set of edges possibly intersecting the merge twice (or not at all).

Each merge chain crossing point is a Voronoi Vertex in the merged diagram. Hence, it has exactly three input points associated with it. Two of these points are from the same side (these induce the Voronoi Edge, $e \in E_m^l$, which was crossed), say p_i, p_j from shapes in $\mathcal{L}$. The third input point is from the other side, say r from $\mathcal{R}$. We need to determine r.

To determine r, we only need to find the Voronoi Region of r. Let us suppose that we could determine the intersections of edges in $HVor(\mathcal{R})$ with e. If these intersecting edges are ordered along e, then adjacent edges will define the boundary of Voronoi Regions in $HVor(\mathcal{R})$. We may easily compute the Hausdorff distance from each intersection to $\mathcal{L}$ and to $\mathcal{R}$ (we only need the distance from the intersection to the input points associated with each edge). The boundary of $hreg(r)$ will have one edge intersecting e closer to $\mathcal{R}$ and the another edge intersecting closer to $\mathcal{L}$.

For a singly-intersected edge $e \in E_1^l$, we have determined an endpoint closer to $\mathcal{L}$, v_l and an endpoint closer to $\mathcal{R}$, v_r. We note that the search variation of the Red-Blue Line Intersection algorithm, presented in Section 2.4.2, will search the intersection of edges between Red lines (edges in E_1^l) and Blue lines (edges in E^r). It is then sufficient to define the predicate, $\mathcal{P}(q, e), q \in E_1^l, e \in E^r$ to determine whether the intersection between q and e is above or below the required one, the

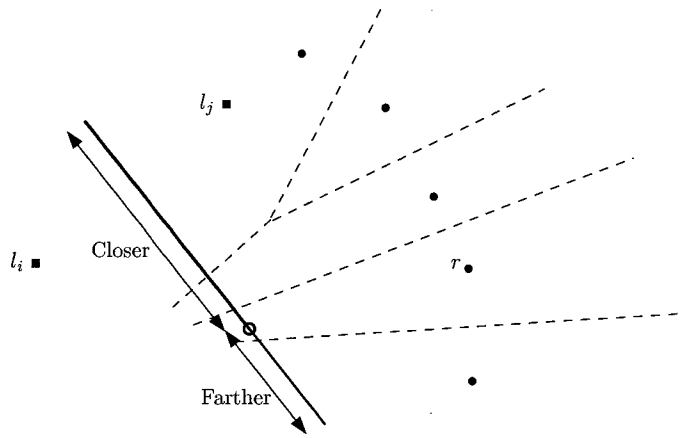


Figure 4.3: Merge vertices on edges in E_1^l may be found by Binary Search

required one being either of the edges adjacent to $hreg(r)$. Recall v_l is e's endpoint closer to $\mathcal{L}$, v_r is closer to $\mathcal{R}$. We may determine whether a particular intersection is closer to $\mathcal{L}$ (i.e. closer to the input points inducing q), or closer to $\mathcal{R}$ (i.e. closer to input points inducing e). If the intersection is closest to $\mathcal{L}$, we must search towards v_r. If the intersection is closest to $\mathcal{R}$, then we must search towards v_l.

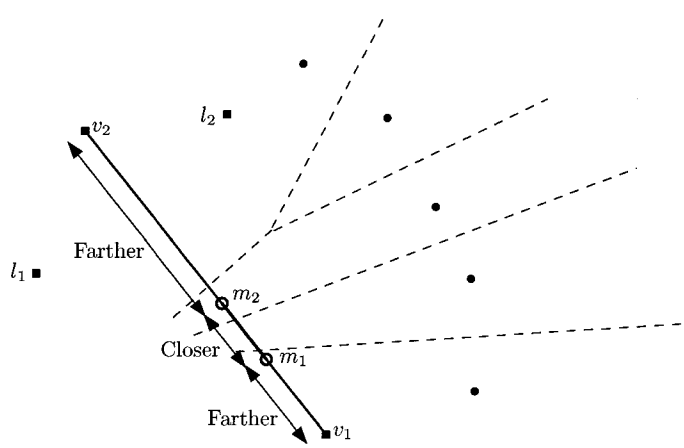


Figure 4.4: Merge vertices on edges in E_2^l may be found by Binary Search

The second case, when $e \in E_2^l$, is more difficult. There may or may not be two merge vertices on e. We know that e's two endpoints, say v_1 and v_2, are both farther from $\mathcal{L}$ than $\mathcal{R}$. Hence, if there exist two merge vertices, they partition e into three parts, the middle part closest to $\mathcal{L}$, and the two end parts closest to $\mathcal{R}$ (See Figure 4.3).

We must perform two searches on e, one for each potential merge vertex. Let us describe the search for a merge vertex, m_1 closest to v_1. The search for a merge vertex m_2 closest to v_2 is symmetric. We define a different predicate, $\mathcal{P}(q, e)$, for q induced by input points $l_1, l_2 \in E_2^l$ and e induced by input points $r_1, r_2 \in E^r$. If the intersection, i, of q and e is closer to $\mathcal{L}$ than $\mathcal{R}$, then we are in the middle close region and need to search towards v_1. Otherwise, we are closest to $\mathcal{R}$ and we need to determine whether we are above or below m_1. Let r_1, r_2 from $\mathcal{R}$ be the input points inducing e. By the Limiting Lemma (Lemma 2), we know that the shape $E_1^r \ni r_1$ is either limiting on v_1 or on v_2, since E_1^r is contained in the circle centred on i of radius $d(l_1, i)$. If E_1^r is limiting on v_1, then we search towards v_2, otherwise we search towards v_1.

Given this predicate for searching intersections on edges in E_2^l, we may again use the search variant of the Red-Blue Line Intersection algorithm to find the merge vertices, if they exist.

Trivially, both predicates may be computed in constant time given some q and e, since both predicates involve a constant number of intersections and distance computations between known points.

4.5 Creating the Merge Chains

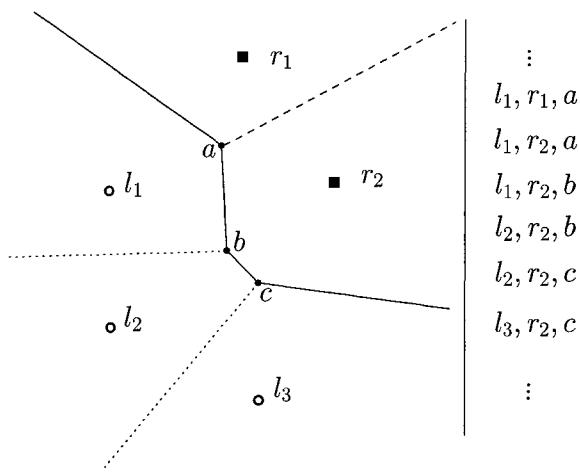


Figure 4.5: Sorting to find Merge Chain edges

Each merge chain crossing point is a Voronoi Vertex in the merged diagram. Hence, it has exactly three input points associated with it. Two of these points are from the same side (these induce the Voronoi Edge which was crossed), say l_i, l_j from $\mathcal{L}$. The third input point is from the other side, say r from $\mathcal{R}$. We have already determined this third point, which allows us to easily compute the merge chain vertices. All that remains is to compute the merge chain edges.

A merge vertex is adjacent to three edges. We note that the edge associated with l_i, l_j has already been computed in $HVor(\mathcal{L})$. The two new edges (i.e. the merge chain edges) will be those associated with l_i, r and l_j, r. Assuming that each merge chain vertex has a unique identifier, we may create triplets, e.g. $l_i, r, vertex$. These triplets may be globally sorted by the first two elements. Each processor may then scan its portion of the triplet list, and for each adjacent pair of entries,

create an edge between them iff their key (the first two elements) are identical. After creating these edges, we will have the merged Voronoi diagram.

4.6 Analysis

We now analyse the efficiency of the CGM Hausdorff Voronoi construction algorithm. The merge step itself includes a constant number of $O(\frac{n \log n}{p})$ global sorts [15]. Merging also includes CGM Point Location, which requires $O(\frac{n \log n}{p})$ time (Section 2.4.1), and our modified CGM Red-Blue Line Intersection search variation, which requires $O(\frac{n \log^3 n}{p})$ time (Section 2.4.2).

The entire algorithm performs a dividing step, which comprises global sort taking $O(\frac{n \log n}{p})$. The independent computation depends on the sequential algorithm, so we let the time $T(n)$ be the time to sequentially compute a Hausdorff diagram of size n. The merge step is performed on $\log p$ recursively divided parts, and so the merging in total takes $O(\frac{n \log^3 n \log p}{p} + T(\frac{n}{p}))$ local computation.

The global space required for the entire algorithm is $O(n)$ except in the merge step, where the sequential line intersection subproblem requires $O(\frac{n \log^2 n}{p})$ space in total. The subalgorithms restrict $n \geq p^3$.

All Global Sorts require a constant number of rounds, and each merge step also requires constant rounds (see Sections 2.4.2 and 2.4.1). Therefore, together all merge steps require $O(\log p)$ rounds. Hence, we conclude that:

Theorem 7 *The CGM Hausdorff Voronoi diagram construction algorithm solves the parallel Hausdorff Voronoi diagram problem in* $O(\frac{n \log^3 n \log p}{p} + T(\frac{n}{p}))$ *local computation, in* $O(\log p)$ *rounds with* $O(\frac{n \log^2 n}{p})$ *space per processor where* $T(\frac{n}{p})$ *is the*

bound on a sequential Hausdorff Voronoi algorithm, and $n \geq p^3$.

4.7 Extensions and Evaluation

4.7.1 Improved Sequential Algorithm

The existing divide and conquer (Papadopoulou [38]) and sweepline (Papadopoulou [34]) algorithms may be practical to implement in practice. They are also easily generalised to crossing input. However, they are not optimal. In fact, Papadopoulou leaves as an open problem the improvement of the asymptotic worse-case running time, even when limited to non-crossing input. For our parallel algorithm, we have restricted input to non-crossing shapes, and our parallel algorithm immediately presents an improved sequential algorithm compared to these recent sweepline and divide-and-conquer algorithms.

We repeat the same divide and conquer technique as in the existing Hausdorff Voronoi divide-and conquer algorithm (Algorithm 22), except we improve the merge step by doing a sequential version of Algorithm 26:

Algorithm 29: HVOR-NOCROSS-MERGE()

1. Use Point Location to find the subsets of Voronoi Edges crossing the merge chain. Let these subsets be , $E_m^l \subset E^l$ and $E_m^r \subset E^r$.
2. Use the Red-Blue Line Intersection search variant to find the merge chain vertices on edges in E_m^l and E_m^r.
3. Remove edges (or portions of edges) which are not present in the merged Voronoi diagram.
4. Create a set of edge endpoints, two for each merge chain vertex. Sort endpoints. Connect adjacent endpoints to form edges.

Note that step 1 may be performed using a sequential point location algo-

rithm, such as the $O(n \log n)$ time sweep algorithm discussed in Section 2.4.1. The bichromatic line intersection in step 2 may likewise be completed using the sequential $O(n \log^3 n)$ time algorithm of Section 2.4.2. So, we present an improved sequential algorithm for computing the Hausdorff Voronoi diagram of non-crossing input:

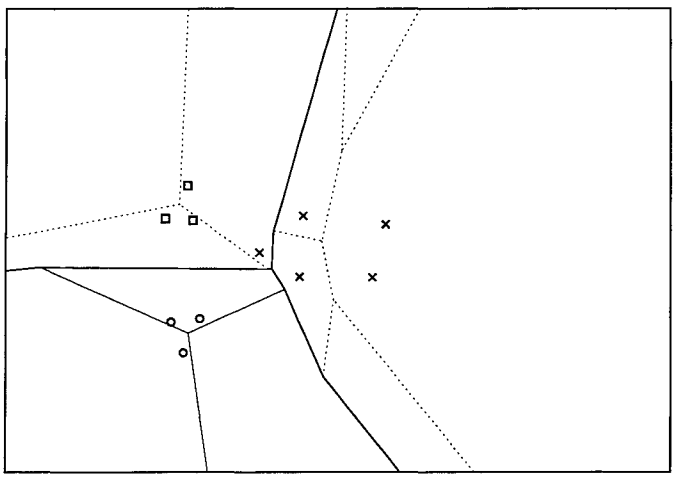
Theorem 8 *A Hausdorff Voronoi diagram of non-crossing input may be constructed in* $O(n \log^4 n)$ *time using the improved merge step of Algorithm 29.*

4.7.2 Speedup

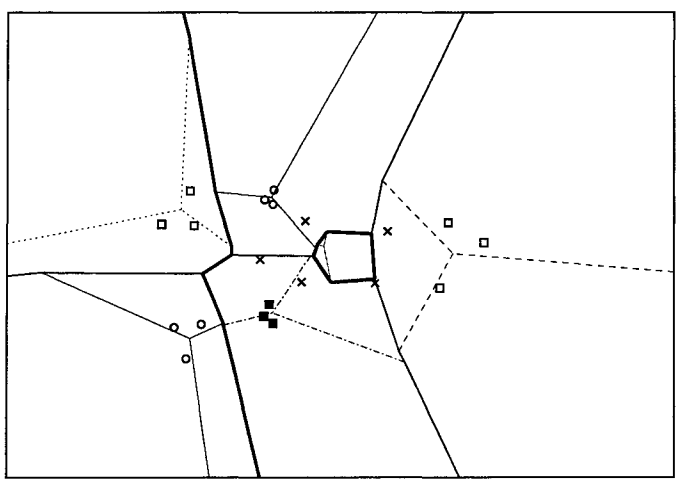
We may express the worst-case upper bounds of the sequential sweepline (Section 3.2) and the sequential divide-and-conquer (Section 3.3) only in terms of input size, n. Both algorithms compute the Hausdorff Voronoi diagram in $T(n) = O(n^2 \log n)$. The three-dimensional envelope mapping algorithm theoretically runs more efficiently, in $O(n^2 \alpha)$. When restricted to only non-crossing input, we present, in the previous section, an algorithm with a complexity of $O(n \log^4 n)$. Even using this sequentially, the parallel algorithm is dominated by the sequential computation and is therefore bounded by $O(\frac{n \log^4 n}{p})$. Hence, we obtain speedup of p.

However, in practice, the sequential divide and conquer and sweepline algorithms may also be expressed using parameters which encapsulate the input's geometric configuration. Indeed, for the VLSI manufacturing application, empirical evidence has been claimed to indicate that the geometric parameters for the sequential algorithms tend to remain small [34]. For cases where these extra

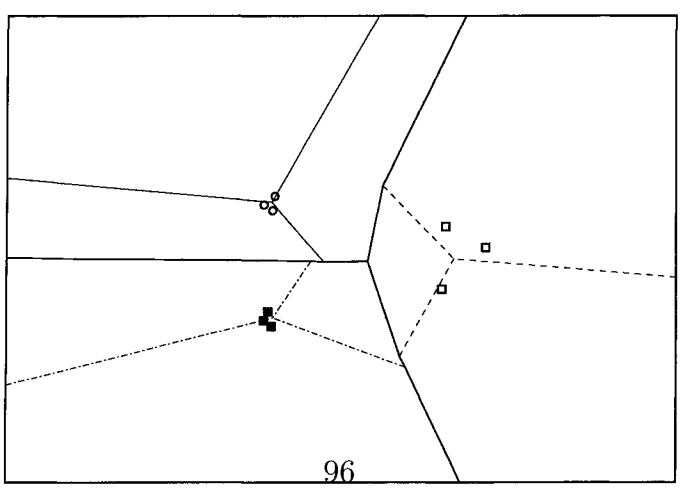
parameters are negligible, the algorithm would run in $O(n \log n)$ time. In the following chapters we discuss and evaluate the performance of an implementation of the CGM Hausdorff Voronoi algorithm and experimentally determine the running time and speedup of the CGM algorithm that we presented in this chapter.



Left Subdiagram



Merged Diagram



Right Subdiagram

Figure 4.6: Example merge of two Hausdorff sub-diagrams

Chapter 5

Implementation and Performance Results

5.1 Overview

Our goal has been to develop an efficient parallel algorithm using the CGM model. The CGM model provides a framework that should predict good parallel algorithms. In addition to our analysis of the algorithm via the CGM model, it is beneficial to obtain some insight into both the algorithm's feasibility and its experimental speedup on an existing parallel computer.

We therefore have completed a prototype implementation of the CGM algorithm, including the sequential sweepline algorithm. In this chapter, we describe this code and we also provide data showing the running time of the algorithm on the parallel cluster computer in the High Performance Virtual Computing Laboratory (HPCVL) [1]. The HPCVL cluster comprises 64 dual-processor Xeon machines, half with a 1.7GHz clock speed and 1GB RAM, the other half with a 2.0GHz clock speed and 1.5GB RAM. The cluster runs a version of the Linux 2.4

kernel and uses LAM MPI 7.1.1 for parallel programming. For our tests we use only one processor per machine.

5.2 Code Design

As a general overview of how the code has been implemented, we describe its major modules, the libraries used, important data structures, and simplifications that we made in the algorithm.

5.2.1 Existing Libraries

We used two standard libraries. The first library used was CGM*Lib*[10]. This library provides a C++ wrapper around the MPI parallel library. MPI is the current standardised programming interface for message passing programming. In the case of the HPCVL cluster, the LAM MPI implementation of this interface translates message passing commands into TCP/IP socket communication. The CGM*Lib* library builds on MPI by providing efficient algorithms and memory management that are particularly suited for writing CGM style parallel algorithms. All communication between processors occurs using CGM*Lib*, and most notably, we make use of CGMLib's parallel sorting algorithms.

The second library used was the Library of Efficient Data Structures and Algorithms (LEDA) [29]. The LEDA library implements many fundamental algorithms, including some from computational geometry. Our code relies on LEDA geometric primitives (orientation, incircle/distance tests) and geometric datatypes (points, segments, rays, circles, etc.). We use LEDA's farthest Voronoi dia-

gram construction algorithm. The LEDA library has efficient and versatile graph datatypes, which we use to implement a distributed Hausdorff Voronoi diagram data structure. Other LEDA datatypes and algorithms we use include priority queues and hash tables.

5.2.2 Voronoi Diagram Data Structure

One parallel data structure not explicitly described in the preceding chapters' algorithms is a distributed representation of a Voronoi diagram. The Hausdorff Voronoi diagram may be represented as an embedded planar graph, easily facilitated on one processor by a LEDA graph. We describe how components of this graph structure are distributed efficiently across processors.

Between iterations of the merge (Algorithm 26) we ensure that no edges are duplicated. Before merging, we start with independent Hausdorff Voronoi diagrams that are sequentially constructed at each processor. Then, in any given merge step, we combine pairs of diagrams. In doing so, we only update existing edges and create new edges of the merge components. Thus, edges within the distributed Voronoi data structure are not duplicated.

Unlike edges, vertices and regions may be duplicated. However, each edge is associated with exactly two vertices and with two regions. The total number of both vertex objects and region objects stored across processors is thus twice the number of edges. So each edge is of constant size, with references to two vertices and two regions, and since the graph is planar, there are a linear number of edges. Clearly, then, even with some duplicated components, the distributed Voronoi graph data structure is linear in the size of the graph.

5.2.3 Major Modules

The code can be divided into four submodules. The first, the main parallel code, stores and updates the main distributed Hausdorff Voronoi diagram data structure.

The second module, the sequential algorithm, computes a Hausdorff Voronoi diagram locally at a processor. This algorithm is the Hausdorff sweepline (see Section 2.1.3), and is the algorithm that we will compare to for determining speedup. We implemented the sweepline algorithm [1] , making use of LEDA's priority queue and linked list structures. Rather than using an existing library, the balanced tree for maintaining the wavefront was implemented for this specific application.

The third and fourth modules are the parallel point location and parallel red-blue line intersection algorithms. For these two algorithms, we implemented the sequential versions ourselves, using segment trees. For sequential point location, there are other, asymptotically better algorithms, but extending our existing parallel segment tree code to handle the sequential case was trivial. In the analysis contained in Section 4.6, the red-blue line intersection algorithm's complexity dominates the parallel point location. The red-blue line intersection implementation closely follows the presentation of the algorithm. The two modules are both variations on segment trees, and so there is much related code between the two segment tree algorithms.

[1] Although Hausdorff Voronoi literature [34] describes an existing sweepline implementation which is tailored to the VLSI manufacturing application (Section 3.4), neither the existing sweepline code nor experimental data have been published.

5.2.4 Robustness Issues

It is well known (e.g. [41]) that computational geometry algorithms can have many degenerate special cases and imprecision errors due to fixed-precision numbers. Algorithms are frequently presented in a simplified manner, assuming general position of the input so that degenerate cases do not clutter the discussion of the main algorithm. Also, algorithms use a Real RAM model, which is similar to the standard von Neumann model, but with the addition that real number arithmetic primitives require constant space and time. However, to ensure fast computation and to avoid communicating arbitrarily large precision numbers between processors, we used standard fixed-precision types. Let us enumerate the issues encountered that affect the robustness of the CGM Hausdorff Voronoi algorithm as implemented in the code.

A problem is when input points are cocircular, which makes it possible for more than three Voronoi edges to meet at a vertex. In, for example, the Hausdorff sweepline algorithm (Algorithm 17), each point set's farthest Voronoi diagram's edges are traced out during the sweep. If more than three of farthest Voronoi diagram edges meet at a vertex, then it is difficult to correctly order the occurrence of the site events that trace out this diagram. It is difficult to order site events that start multiple regions at the same vertex. The correct order is necessary so that regions will be created in the beachfront in the same order that they appear in the Voronoi diagram.

Another problem is when input points are collinear. For example, when performing the CGM merge algorithm, we perform point location among edges, in-

cluding unbounded ones. Both the point location and red-blue line intersection subproblems require sorting edges vertically, and unbounded edges may be sorted by slope. But if collinear points induce some unbounded edges which are parallel, then small roundoff in their slopes may incorrectly order these parallel edges. As well, vertical edges require special cases in segment tree code, and horizontal edges can require special cases in the Hausdorff sweepline code.

Problems are caused if an edge's endpoint lies on an edge from the opposing subdiagram. The orientation the the edge endpoint is then subject to rounding errors, which would affect, for example, the correctness of bichromatic line intersection.

Such problems can often be worked around by introducing special cases. However we can conclude from our experience implementing the code that careful consideration to enumerate these degenerate and special cases is essential to creating robust implementations. We have ensured a careful selection of input sets to avoid excessive problems.

5.2.5 Input Generation

Large input datasets are needed for testing the implementation. We generate large datasets which avoid the outstanding degenerate cases discussed in the previous section, and that have sets with disjoint convex hulls.

To do so, we create a grid of disjoint rectangles in the plane, and place one hull in each rectangle. Since the rectangles are disjoint, so are the resulting point sets. In generating points, we follow LEDA's convention and model point coordinates as large integers. We avoid generating random hulls constrained to each rectangle,

since for large input sets the likelihood of introducing degeneracy errors increases. Instead, we repeat a pattern of triangles, one in each rectangle, such that no two adjacent triangles' edges are collinear. Also, we tile a slightly skewed circular shape of boxes to ensure that an individual processor receives a vertical slab with hulls that produce no collinear unbounded Voronoi edges and no vertical edges.

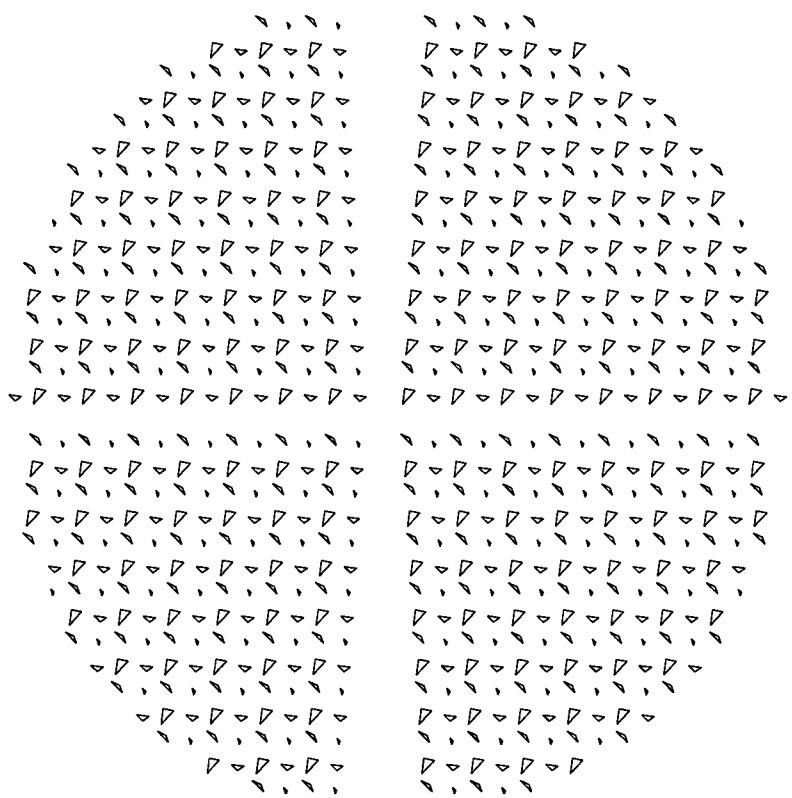


Figure 5.1: Small example of possible Hausdorff input

We have also developed tools to automatically verify and to visualize the input and output. Slow, but simple bruteforce prototypes were used to produce output which was compared to the CGM and sweepline implementations' output using

these verification tools.

5.3 Results

We now present the results of running the Hausdorff Voronoi diagram code on the HPCVL cluster. Results have been gathered for input generated in a variety of sizes. The previous section briefly described the generated input patterns.

We show the sequential running time of the Hausdorff Voronoi sequential sweepline algorithm in Figure 5.2. The algorithm ran for input sizes as large as possible. Larger input than shown here exhausts the available memory on an HPCVL node, which has at most of 1.5GB of RAM, although some of the memory is used by system overhead.

In Figure 5.3, we show the speedup for non-trivially sized problems that comfortably fit in memory. In Figures 5.6, 5.7, 5.8, we see the parallel running time for a fixed number of processors over a range of problem sizes. These results illustrate the local computation performed versus the communication overhead.

In the following section we discuss how these preliminary results compare with the theoretical measures of the efficiency of our CGM Hausdorff Voronoi parallel algorithm.

5.4 Interpretation

In Chapter 4, we presented a novel algorithm to compute the Hausdorff Voronoi diagram. In the previous sections, we briefly introduced implementations of both the

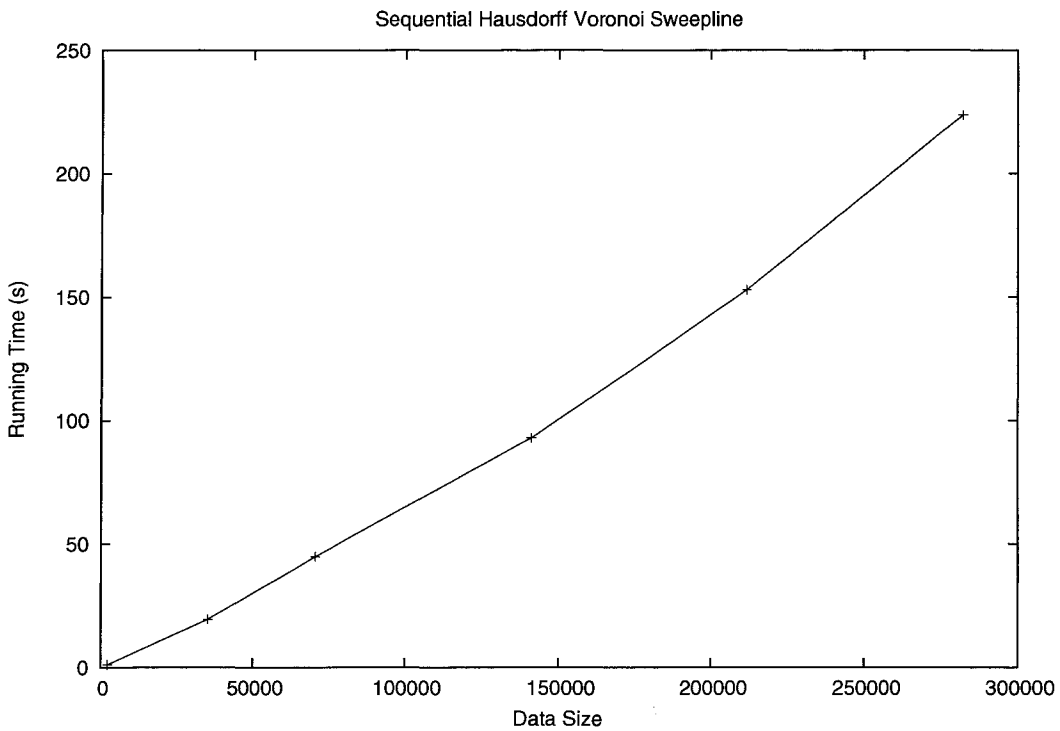


Figure 5.2: Running Time of (Sequential) H-Vor Sweepline

CGM Hausdorff Voronoi parallel algorithm and the Hausdorff Voronoi sweepline algorithm. We also provided results from running these programs and presented experimental results. Let us discuss our findings based on these experiments.

First, we note that the parallel program performs poorly. The sequential algorithm runs in less time than the parallel program in all tests. We show the parallel speedup of the algorithm in Figure 5.3, where the three curves show the ratio of the parallel running time to the sequential running time. In these experiments, the sequential version always performs better. Neither varying the number of processors nor the varying problem size has provided a speedup above 1.

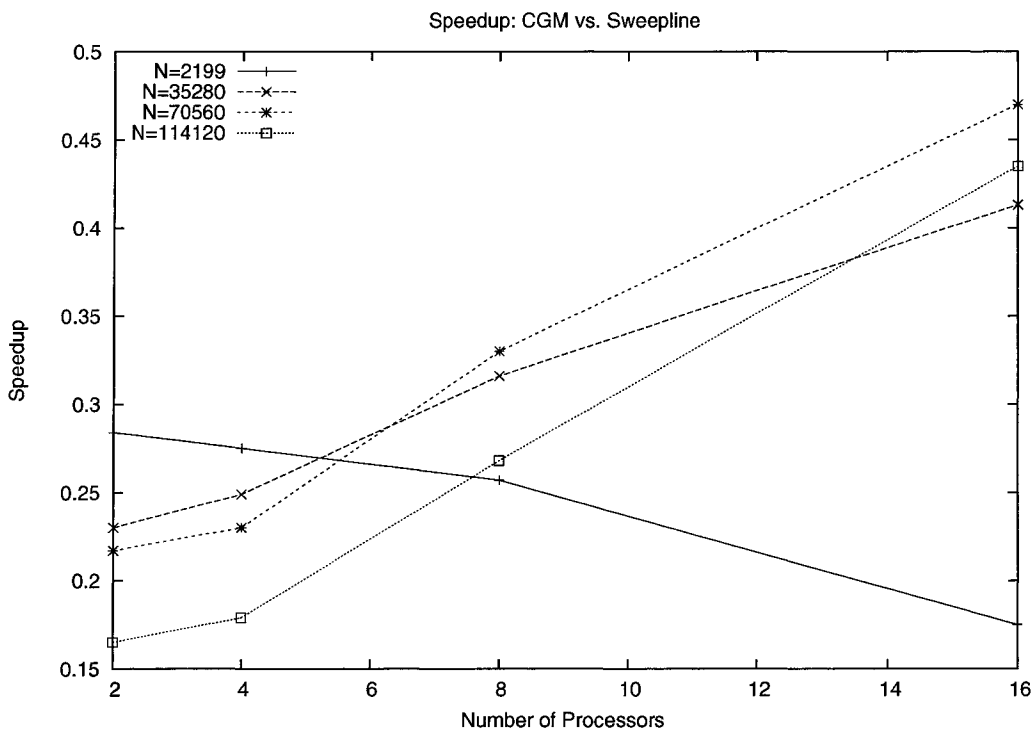


Figure 5.3: Parallel Speedup fixed N

Second, our results for the sequential algorithm in Figure 5.2 show the sequential algorithm's running time, which shows a somewhat flat upwards slope, indicating that the algorithm is quite efficient for problem sizes at least into the hundreds of thousands of points. We present parallel results for problem sizes in the range of tens of thousands of input sizes, because larger problems would not fit in the memory of HPCVL nodes. The parallel code requires more memory than the sequential code due to the construction of the nodes and catalogs of large segments trees. The sweepline algorithm does not require any data structure as large as these segment trees. However, in Figures 5.4, 5.5 where 16 processors are used, the greater memory requirements of the parallel code are spread out across

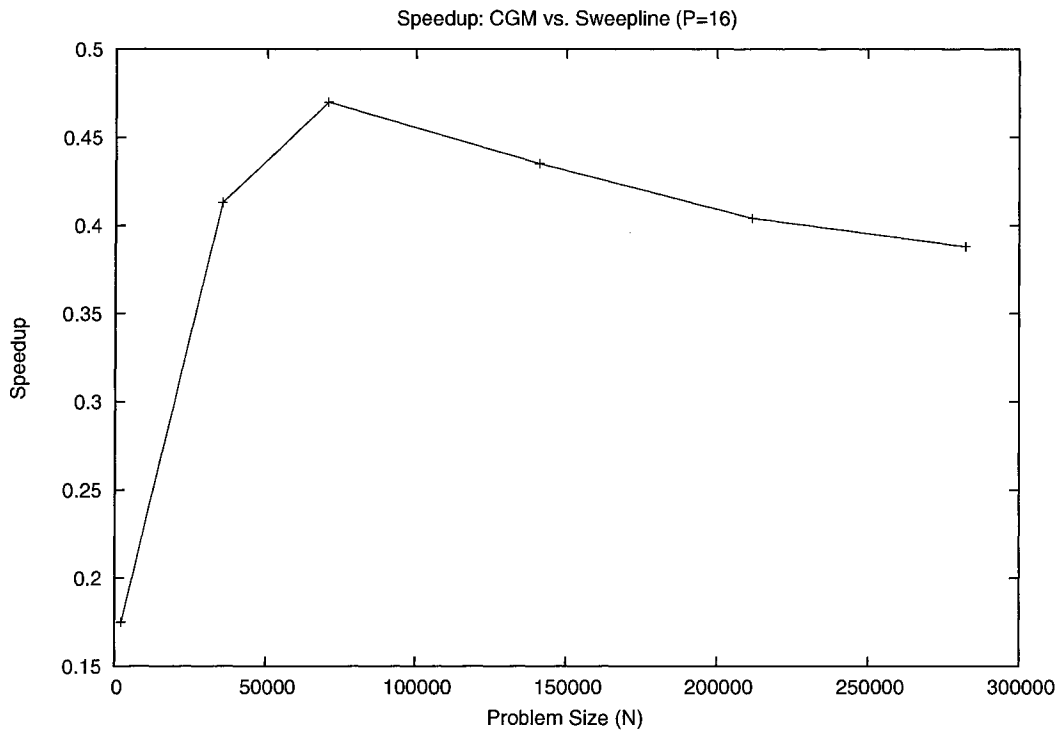


Figure 5.4: Parallel Speedup (fixed P)

more processors and thus larger problem sizes may be computed.

Now, let us look at why the parallel code does not run quickly compared to the sequential algorithm. Communication overhead for small problems dominates, as in Figure 5.6. As the problem size increases, (as in Figures 5.7, 5.8), we see that the computation time decreases as more processors are used, but a large amount of communication overhead still exists. In fact, as shown in Figure 5.5, the time for interprocessor communication is close to the sequential running time for a variety of input sizes on 16 processors. So communication overhead alone plays a critical role in the poor parallel performance.

The choice of input affects running time for both the sequential and parallel

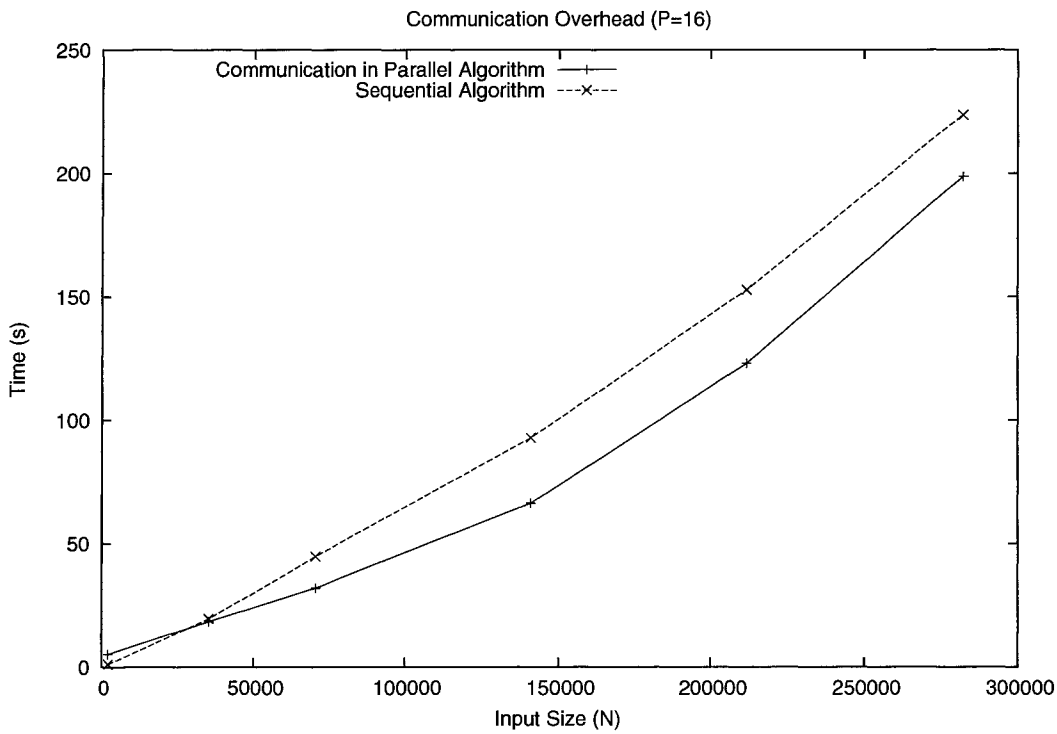


Figure 5.5: Communication Overhead (16 procs)

codes. Our generated input, described in Section 5.2.5, provides regularly spaced, disjoint sets, which can be handled in practice and which should approximate the large number of objects probably seen in VLSI applications. In fact, the worst-case bounds on the sequential algorithm are not known to be tight, and creating an instance illustrating worst-case properties is non-trivial. So our input, similar to what is reported for actual VLSI data [34], has an insignificant K_a value, and hence one would expect that the sequential algorithm runs in $O(n \log n)$ time.

It is interesting to look at the bottlenecks in the parallel code to see where the most time is spent. The merge algorithm dominates the running time, and we summarise the portion of the running time for each module in the merge algorithm

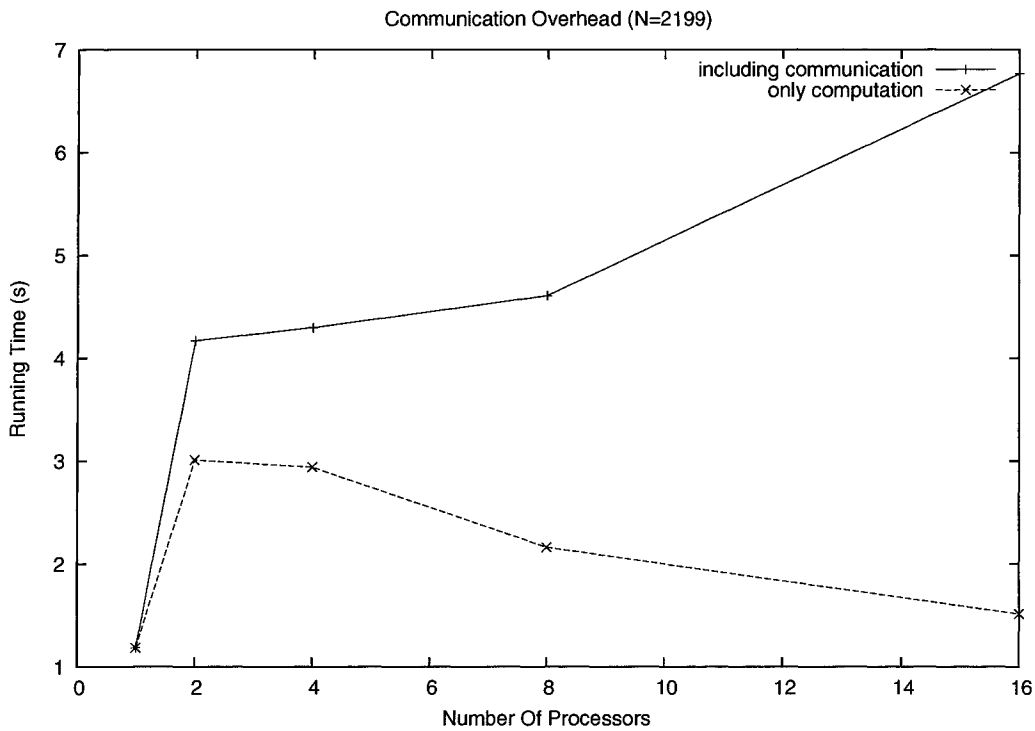


Figure 5.6: Communication Overhead (1/3)

in Table 5.4. Although the theoretical bottleneck lies in the parallel bichromatic line intersection algorithm, we note that slightly more time is actually spent in the parallel point location module. Even the extra effort to "stitch" the two subdiagrams together (Steps 3,4 in Algorithm 26) has some effect on the total running time.

This indicates what is probably the critical issue with the parallel code versus the sequential code. The "constant factors" in the parallel algorithm appear to outweigh the asymptotic advantage that the parallel algorithm holds over the sequential algorithm for reasonable input sizes.

The merge steps of the parallel algorithm manipulate sets of Voronoi edges.

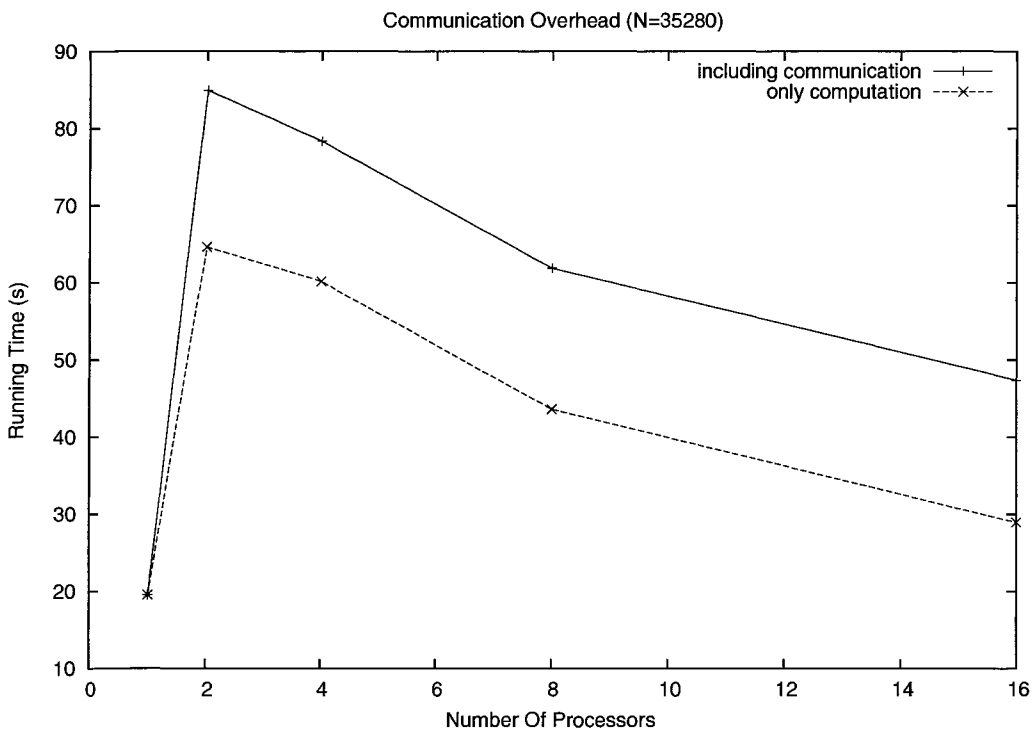


Figure 5.7: Communication Overhead (2/3)

These edges must be packed into CGM*Lib*'s CommObjectList vector collections, where packed edge objects must include physical copies of all vertices and sites associated with each edge. After the merge is complete, this data must be unpacked and returned into a format suitable for LEDA's graph data structure. This structure is necessary to maintain the overall topological structure between vertices, edges and regions.

Additionally, at different points during the program parallel modules require different additional information to be associated with each packed edge. Redundant copying occurs between these types of CGM*Lib* objects. Ideally inefficiencies such as this could be optimised, although it is not a critical factor in overall per-

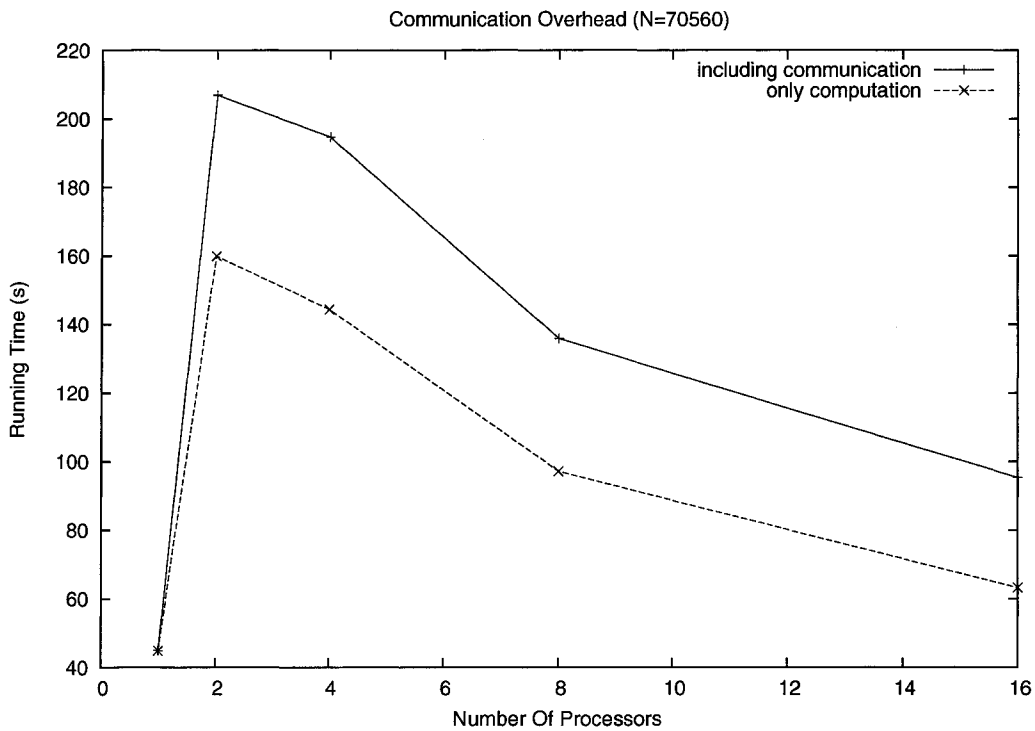


Figure 5.8: Communication Overhead (3/3)

formance.

A significant difficulty in achieving a fast parallel algorithm is the number of times algorithms are called. Even when the subalgorithms used in the code are efficiently implemented, frequent use indicates large constants of computational overhead. For example, a very clear indication of the parallel algorithm's constant factors is shown by a direct comparison of the number of sorts. Sorting is a well-studied algorithm, both theoretically and experimentally. Well tested sorting libraries have been used in both the sequential and parallel settings. Also, sorting occurs at a conceptual granularity that makes it easy to count and evaluate the overall sorting that takes place in each algorithm.

Table 5.1: Percent of total running time in each merge sub-algorithm, P=16

# Points	Point Loc.	RB-Line Int.	Voronoi Stitching
2199	39%	50%	2%
35280	50%	38%	0%
70560	49%	39%	1%
141120	48%	40%	3%
211680	44%	39%	5%
282240	44%	38%	8%

The sequential algorithm uses a priority queue to the sort events. When each events is handled, it is ordered in the existing sweepline using a balanced binary tree. Both balanced binary trees and heap based priority queues are simple, efficient data structures.

The parallel code, by comparison, has approximately 20 local sorts in each merge step. This would, for example, require 4 merge steps using 16 processors, for a total of 80 local sorts.

Recall that a global sort requires two local sorts. Each step performs two parallel point location and two parallel line intersections. The x-intervals to create the tree skeleton must be globally sorted for each algorithm instance. Then, the catalogs in the shared tree top (T_0) must be locally sorted. These catalogs are then globally sorted to group catalog lists together. Some optimisations are possible, such as an integer sort for this final catalog sort (see [9]), and perhaps a small number of sorting calls could be eliminated by optimisation. Also, in parallel we have $\frac{1}{p}$ of the data at each processor, so the sorts are smaller. However, the number of sorts required does provide some indication of the greater computational constants required throughout the parallel code.

Chapter 6

Conclusion and Future Work

We presented an algorithm to compute the Hausdorff Voronoi diagram of non-crossing input using a p processor CGM. Hausdorff Voronoi diagram input is non-crossing when the input convex hulls are pairwise non-crossing, although possibly overlapping. This algorithm requires $O(\log p)$ communication rounds and $O(\frac{n \log^4 n}{p})$ local computation. We demonstrated that this algorithm attains a theoretical speedup of p. We also improved on existing bounds for the sequential computation of Hausdorff Voronoi diagrams of non-crossing input. Our parallel algorithm implies a sequential algorithm requiring $O(n \log^4 n)$.

We presented an implementation of the CGM algorithm and the sequential sweepline algorithm, and provided experimental results evaluating these algorithms in practice. The parallel algorithm did not perform well due primarily to hidden constants which became evident in the actual implementation and to the behaviour on the chosen input sets.

Open problems related to this thesis remain. Refining the CGM algorithm may occur in two veins. First, a more general parallel algorithm which runs correctly and efficiently for crossing shapes would be theoretically interesting, and it

has been reported [34, 37] that crossing shapes may occasionally occur in practice. Second, one may want to refine the algorithm in ways which would improve the experimental performance of the algorithm. Further empirical support by a detailed experimental evaluation on actual VLSI input data would provide further validation to the practical usefulness of this algorithm. Although the Hausdorff Voronoi diagram's application to VLSI via breaks is the most interesting computational problem, other Voronoi diagram variations for VLSI manufacturing could be parallelised.

Another interesting approach to computing Hausdorff Voronoi diagrams exists in the mapping to 3D envelopes. Further research is required to determine whether this technique could lead to an efficient parallel algorithm, which may even allow for crossing input in parallel.

Bibliography

[1] `http://www.hpcvl.org/`. Online. HPCVL.

[2] Alves, C., Caceres, E., Dehne, F., and Song, S. A CGM/BSP parallel similarity algorithm. In *Proc. Brazilian Workshop on Bioinformatics* (2002), pp. 1–8.

[3] Atallah, M. A linear time algorithm for the Hausdorff distance between convex polygons. *Information Processing Letters 17* (1983), 207–209.

[4] Aurenhammer, F. Voronoi diagrams - a survey of a fundamental geometric data structure. *ACM Computing Surveys 23*, 3 (1991), 345–405.

[5] Aurenhammer, F., and Klein, R. *Handbook of Computational Geometry.* North-Holland, 2000, ch. Voronoi Diagrams, pp. 201–290.

[6] Caceres, E., Dehne, F., Ferreira, A., Flocchini, P., Rieping, I., Roncato, A., Santoro, N., and Song, S. Efficient parallel graph algorithms for coarse grained multicomputers and BSP. In *Proceedings of the 24th International Colloquium on Automata, Languages and Programming* (1997), Springer-Verlag, pp. 390–400.

[7] CACERES, E., DEHNE, F., MONGELLI, H., SONG, S., AND SZWARCFITER, J. A coarse-grained parallel algorithm for spanning tree and connected components. In *Proc. Euro-Par* (2004), vol. 3149, LNCS, pp. 828–831.

[8] CHAN, A., AND DEHNE, F. A note on coarse grained parallel integer sorting. *Parallel Processing Letters 9*, 4 (1999), 533–538.

[9] CHAN, A., DEHNE, F., AND RAU-CHAPLIN, A. Coarse-grained parallel geometric search. *Journal of Parallel and Distributed Computing 57*, 2 (1999), 224–235.

[10] CHAN, A., DEHNE, F., AND TAYLOR, R. CGMGRAPH/CGMLIB: Implementing and testing CGM graph algorithms on pc clusters and shared memory machines. *International Journal of High Performance Computing Applications 19*, 1 (2005), 81–97.

[11] CHAZELLE, B., EDELSBRUNNER, H., GUIBAS, L., AND SHARIR, M. Algorithms for bichromatic line-segment problems and polyhedral terrains. *Algorithmica 11*, 2 (1994), 116–132.

[12] COLE, R., ODUNLAING, C., AND GOODRICH, M. T. A nearly optimal deterministic parallel Voronoi diagram algorithm. *Algorithmica 16* (1996), 569–617.

[13] DEHNE, F., DENG, X., DYMOND, P., FABRI, A., AND KOKHAR, A. A randomized parallel 3D convex hull algorithm for coarse grained multicomputers. *Theory of Computing Systems 30* (1997), 547–558.

[14] DEHNE, F., FABRI, A., AND RAU-CHAPLIN, A. Scalable parallel computational geometry for coarse grained multicomputers. *International Journal in Computational Geometry 6*, 3 (1996), 379–400.

[15] DIALLO, M., FERREIRA, A., AND RAU-CHAPLIN, A. A note on communication-efficient deterministic parallel algorithms for planar point location and 2D Voronoi diagram. *Parallel Processing Letters 11*, 2/3 (2001), 327–340.

[16] EDELSBRUNNER, H., GUIBAS, L., AND SHARIR, M. The upper envelope of piecewise linear functions: Algorithms and applications. *Discrete and Computational Geometry 4* (1989), 311–336.

[17] EDELSBRUNNER, H., AND SEIDEL, R. Voronoi diagrams and arrangements. *Discrete Computational Geometry 1* (986), 25–44.

[18] FABRI, A., AND DEVILLERS, O. Scalable algorithms for bichromatic line segment intersection problems on coarse grained multicomputers. In *WADS* (1993), vol. 709 of *LNCS*, pp. 277–288.

[19] FABRI, A., AND DEVILLERS, O. Scalable algorithms for bichromatic line segment intersection problems on coarse grained multicomputers. *International Journal of Computation Geometry and Applications 6*, 3 (1996), 379–400.

[20] FORTUNE, S. A sweepline algorithm for Voronoi diagrams. *Algorithmica 2* (1987), 153–174.

[21] FORTUNE, S. *Handbook of Discrete and Computational Geometry.* CRC Press LLC, 1997, ch. Voronoi Diagrams and Delaunay Triangulations, pp. 377–388.

[22] FORTUNE, S., AND WYLLIE, J. Parallelism in random access machines. In *Proc. of the 10th annual ACM STOC* (1978), ACM Press, pp. 114–118.

[23] GOODRICH, M. Communication efficient parallel sorting. In *Proc. 28th Annual ACM STOC* (1996), pp. 247–256.

[24] JAJA, J. *An Introduction to Parallel Algorithms.* Addison-Wesley, 1992.

[25] JEONG, C. An improved parallel algorithm for constructing Voronoi diagram on a mesh-connected computer. *Parallel Computing*, 17 (1991), 505–514.

[26] JEONG, C., AND LEE, D. Parallel geometric algorithms on mesh-connected computers. In *Proceedings of the 1987 Fall Joint Computer Conference on Exploring technology: today and tomorrow* (Dallas, Texas, 1987), IEEE, pp. 311–318.

[27] JEONG, C., AND LEE, D. Parallel geometric algorithms on a mesh-connected computer. *Algorithmica*, 5 (1990), 155–177.

[28] LEE, D., AND PREPARATA, F. Location of a point in a planar subdivision and its applications. *SIAM Journal on Computing 6*, 3 (1977), 594–606.

[29] MEHLHORN, K., AND NAHER, S. LEDA: a platform for combinatorial and geometric computing. *Communications of the ACM 38*, 1 (1995), 96–102.

[30] OKABE, A., BOOTS, B., AND SUGIHARA, K. *Spatial tessellations: Concepts and applications of Voronoi diagrams.* J. Wiley, 1992.

[31] PALAZZI, L., AND SNOEYINK, J. Counting and reporting red/blue segment intersections. In *WADS* (1993), vol. 703 of *LNCS.*

[32] PAPADOPOULOU, E. Critical area computation for missing material defects in VLSI circuits. In *Transactions on CICS* (May 2001), vol. 20, IEEE, pp. 569–570.

[33] PAPADOPOULOU, E. On the Hausdorff Voronoi diagram of point clusters in the plane. In *WADS* (2003), vol. 2748 of *LCNS.*

[34] PAPADOPOULOU, E. The Hausdorff Voronoi diagram of point clusters in the plane. *Algorithmica 40*, 2 (2004), 63–82.

[35] PAPADOPOULOU, E., AND LEE, D. L-inf Voronoi diagrams and applications to VLSI layout and manufacturing. In *ISAAC* (1998), vol. 1533 of *LNCS*, pp. 9–18.

[36] PAPADOPOULOU, E., AND LEE, D. Critical area computation via Voronoi diagrams. *IEEE Transactions on Computer-Aided Design 18*, 4 (1999), 463–474.

[37] PAPADOPOULOU, E., AND LEE, D. The min-max Voronoi diagram of polygons and applications in VLSI manufacturing. In *ISAAC* (January 2002), P. Bose and P. Morin, Eds., vol. 2518 of *LNCS*, Springer-Verlag Heidelberg, pp. 511–522.

[38] PAPADOPOULOU, E., AND LEE, D. The Hausdorff Voronoi diagram of polygonal objects: A divide and conquer approach. *International Journal of Computational Geometry and Applications 14*, 6 (2004), 421–452.

[39] PREPARATA, F., AND SHAMOS, M. *Computational Geometry: An Introduction.* Springer-Verlag, 1985.

[40] RAJASEKARAN, S., AND RAMASWAMI, S. Optimal parallel randomized algorithms for the Voronoi diagram of line segments in the plane and related problems. In *SCG '94: Proceedings of the 10th annual SOCG* (New York, NY, USA, 1994), ACM Press, pp. 57–66.

[41] SCHIRRA, S. *Handbook of Computational Geometry.* Elsevier, 2000, ch. Robustness and Precision Issues in Geometric Computation, pp. 597–632.

[42] SHAMOS, M., AND HOEY, D. Closest-point problems. In *Proc. 16th Annuual IEEE SFCS* (1975), pp. 151–162.

[43] SHI, H., AND SCHAEFFER, J. Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing 14* (1992), 361–372.

[44] SINGLER, J. Computing the Voronoi diagram on a coarse-grained multicomputer. Directed Study, Universität Karlsruhe, 2004.

[45] VALIANT, L. A bridging model for parallel computation. *Communications of the ACM 33*, 8 (1990).

[46] VEMURI, B. C., VARADARAJAN, R., AND MAYYA, N. An efficient expected time parallel algorithm for Voronoi construction. In *Proceedings of the 4th annual ACM SPAA* (1992), pp. 392–401.