A Coarse Grained Parallel Algorithm for Hausdorff Voronoi Diagrams

Frank Dehne, Anil Maheshwari and Ryan Taylor
School of Computer Science
Carleton University, Ottawa, Canada
{dehne, maheshwa, rtaylor}@scs.carleton.ca

Abstract

We present the first parallel algorithm for building a Hausdorff Voronoi diagram (HVD). Our algorithm is targeted towards cluster computing architectures and computes the Hausdorff Voronoi diagram for non-crossing objects in time $O\left(\frac{n \log^2 n}{p}\right)$ for input size $n$ and $p$ processors.

In addition, our parallel algorithm also implies a new sequential HVD algorithm that constructs HVDs for non-crossing objects in time $O(n \log^2 n)$. This improves on previous sequential results and solves an open problem posed by Papadopoulou and Lee [18].

1 Introduction

In this paper, we present the first parallel algorithm for building a Hausdorff Voronoi diagram (HVD). Our parallel algorithm also implies a new sequential HVD algorithm that improves on previous sequential results and solves an open problem stated in [18].

1.1 Background and Motivation

One of the most widely studied structures in Computational Geometry is the Voronoi diagram (see e.g. [1]). In its canonical form, a Voronoi diagram is constructed for a planar set of points (sites). The plane is partitioned into regions, one for each site, where each region is the set of points closest to the associated site. In this paper we study the Hausdorff Voronoi diagram (HVD), a generalization of standard Voronoi diagrams. Each site is replaced by an arbitrary object (point set in the plane) and the distance of a point to an object (point set) is defined as the distance to the farthest point in the object. See Section 2 for a formal HVD definition. Like the standard Voronoi diagram, a HVD divides the plane into regions. For any point in the plane, the covering circle centered at that point is the smallest circle that completely encloses at least one object. Observe that for any point within a Hausdorff Voronoi region the covering circle encloses the same object. Hence, a HVD may be considered as a Voronoi diagram of covering circles.

Due to this covering circle property, the HVD has recently gained considerable attention within the context of VLSI manufacturing. The use of HVDs for VLSI yield prediction has been pioneered at IBM and is discussed e.g. in [9, 10, 14, 15, 16, 17, 18, 12, 13]. Part of the design process for new VLSI chips is to determine how resilient the chip’s circuit geometry will be to defects caused during the manufacturing process. The HVD allows for the efficient computation of the critical area of a chip which is an important measure for a VLSI chip’s yield prediction. A chip defect is typically created by impurities or particles that settle on the chip during the manufacturing process. The question is whether or not such an impurity results in a faulty chip. One type of fault considered is when a component on the chip, e.g. a contact on the via layer, is disconnected. For each contact, redundant contact points are placed on the via layer to improve reliability. To destroy the connection created by a via block, all its (redundant) contact points must be destroyed. Hence, a defect (circle) that covers an entire via block causes a faulty chip. The minimum size circle that completely covers a via block is efficiently computed through a Hausdorff Voronoi diagram. It represents the smallest defect that would destroy the chip.

1.2 Previous Work

Voronoi diagrams have been extensively studied and generalized in a variety of ways (see e.g. [1] for an extensive survey). For the Hausdorff Voronoi diagram, sequential algorithms have been presented in [14, 13, 18, 17, 6]. A sequential sweepline HVD algorithm is presented in [14] and a sequential divide-and-conquer method is presented in [18]. A sequential method based on coordinate transformation and lower envelope calculation is presented in [6]. The worst case time complexities are listed in Table ??.

The parallel construction of standard Voronoi diagrams has been studied e.g. in [5, 8, 19]. However, there exists to our knowledge no parallel algorithm for the Hausdorff Voronoi diagram.

The VLSI application of HVDs discussed above requires the computation of very large HVDs. In [18] it was posed as an open problem to speed up HVD construction in the general case and in particular for the case of non-crossing objects. Such objects may overlap but not cross completely, and the geometric objects in VLSI design (e.g. via blocks) are typically non-crossing [18]. The algorithms in [6, 14, 18] are not faster for the case of non-crossing objects. This paper contributes towards solving the problem posed in [18] by providing a much improved sequential algorithm for non-crossing objects.

1.3 New Results

The primary contribution of this paper is to present the first parallel algorithm for Hausdorff Voronoi diagram construction. Our algorithm is coarse grained parallel [4] and targeted towards cluster computing architectures. Our coarse grained parallel algorithm computes the Hausdorff Voronoi diagram for non-crossing objects in time $O\left(\frac{n \log^4 n}{p}\right)$ for input of size $n$ on a coarse grained multiprocessor (CGM) with $p$ processors. We also present a first experimental evaluation of our parallel algorithm.

Computing Hausdorff Voronoi diagrams in parallel is a hard problem, and considerably harder than the parallel construction of standard Voronoi diagrams (e.g. [5, 8, 19]). Such methods are typically based on a parallel divide-and-conquer strategy. For canonical Voronoi diagrams, the merge curve used for “stitching together” two Voronoi diagrams is a single monotone chain. Therefore, the task of merging two canonical diagrams becomes relatively easy. For Hausdorff Voronoi diagrams this is not the case. The merge curve may be comprised of multiple, disjoint components that are not necessarily monotone. In fact, some of these merge components may even be cyclic. An example is shown in Figure 1. The main contribution of this paper is an efficient coarse grained parallel method that is able to deal with multiple merge components that are non-monotone and possibly cyclic.

In addition, the direct sequential adaptation of our parallel algorithm results in a sequential algorithm that constructs HVDs for non-crossing objects in time $O(n \log^4 n)$. This new sequential algorithm contributes towards an open problem posed in [18].

2 Preliminaries

A Hausdorff Voronoi Diagram (HVD) is constructed for a set system with a universe $I$ of $n$ input points in the plane. A subset of the power set of $I$, $S = \{P_1, P_2, \ldots, P_m\}$, is given as input, such that $\bigcup_j P_j = I$ and $P_i \cap P_j = \emptyset$, for all $i, j$ and $i \neq j$. Each set $P_i \in S$ is said to be an object. For HVD computation, the Hausdorff distance function from a point $z \in \mathbb{R}^2$ to an object $P_i \in S$ is defined to be $d_h(P_i, \{z\}) = d_f(P_i, z)$, where $d_f$ denotes the farthest (maximum) Euclidean distance between $z$ and points in $P_i$ [17]. Observe that since we are dealing with the farthest distance, vertices in the interior of the convex hull of any object in $S$ do not participate in the computation of HVDs. Hence, we can assume that each object $P_i \in S$ consists of points that are on its convex hull. It is known that the size of the HVD is linear in the number of points defining the objects.

Definition 1 (Crossing) Two objects, $P_i, P_j \in S$ are said to be crossing if there exist two points $p_i, p_j$ on $P_i$’s convex hull and $q_i, q_j$ on $P_j$’s convex hull such that (1) $\overline{p_ip_j}$ intersects $\overline{p_jp_i}$ and (2) all of $p_i, p_j, q_i, q_j$ are on the convex hull of $P_i \cup P_j$.

In this paper we only deal with objects that are non-crossing (but may overlap) and hence for the rest of the paper we assume that no two input objects are crossing. Next we define the vertices, edges and faces of HVDs.

Definition 2 A Hausdorff Voronoi edge, $e$, is the locus of points with exactly two closest (under Hausdorff metric) points in the input objects in $S$. A Hausdorff Voronoi vertex, $v$, is a point with at least three closest (under Hausdorff metric) points in the objects in $S$. A Hausdorff Voronoi region for an object $P_i \in S$ is $HReg(P_i) = \{z \in \mathbb{R}^2 | d_h(z, P_i) < d_h(z, P_j), \forall P_j \neq P_i\}$. We can further subdivide a Hausdorff region for an object $P_i$ with respect to points on its convex hull as follows. A Hausdorff Voronoi region for a point $p \in P_i$ is $hreg(p) = \{z \in \mathbb{R}^2 | d(z, p) = d_h(z, P_i)$, and $d_h(z, P_i) < d_h(z, P_j), \forall P_j \neq P_i\}$. Given a set $S$ of objects, the Hausdorff Voronoi Diagram, $HVD(S)$, is the union of Hausdorff Voronoi edges and vertices. It forms a planar subdivision of $\mathbb{R}^2$.

3 CGM Algorithm

In this section we present a novel parallel algorithm for computing HVD for non-crossing input objects. The input consists of the set $I$ of $n$ points in the plane and the set $S$ consisting of objects. Our algorithm is designed for a Coarse-Grained Multicomputer (CGM) consisting of $p$-processors. The processors are connected by an arbitrary interconnection network. Each processor has sufficient mem-
Our algorithm follows the divide-and-conquer paradigm. The set of objects are divided into an ordered sequence of \( p \) vertical slabs. We compute HVDs for objects in each slab and then merge them to obtain the HVD of \( S \). The algorithm is sketched in the following.

Algorithm: HVD(S)
Input: A set \( S \) consisting of objects. Each object is a subset of points taken from a set \( I \) consisting of \( n \) points.
Output: Hausdorff Voronoi diagram of \( S \).

1. Order the objects in \( S \) according to their leftmost points. Divide these objects, using the order and the number of points within an object, in \( p \) vertical slabs resulting in sets \( S_i \), for \( i = 1, \cdots, p \). Each set \( S_i \) consists of \( O(\frac{w}{p}) \) input points and is assigned to the \( i \)th processor.
2. The \( i \)th processor computes the Hausdorff Voronoi diagram of objects within \( S_i \) using a sequential algorithm.
3. Perform \( \lceil \log p \rceil \) merge phases, where the \( j \)th phase combines \( \frac{p}{2^j} \) subdiagrams into \( \frac{p}{2^{j+1}} \) diagrams, such that pairs of adjacent subdiagrams are merged.

The overall top-level divide-and-conquer structure of this algorithm is similar to the existing CGM algorithm for computing canonical Voronoi diagrams of points [5]. But it is a completely nontrivial task to extend the algorithm in [5] to compute HVDs and the main reason is outlined in the following. Consider the divide-and-conquer algorithm for canonical Voronoi diagrams and assume that the set of points are partitioned into two groups according to a vertical line; all points to the left of vertical line are in the group \( L \) and the rest of them are in the group \( R \). Furthermore, assume that recursively we have computed Voronoi diagrams of the points in \( L \) and \( R \). The merge step needs to stitch the two diagrams. This is done by first finding the **merge curve**, i.e., the set of all points in the plane that are equidistant from a closest point in \( L \) and a closest point in \( R \). It turns out that the merge curve is \( y \)-monotone and a simple connected chain. Stitching is achieved by throwing away the portion of the Voronoi diagram of \( L \) (respectively, \( R \)) to the right (respectively, left) of the merge curve. Unfortunately, in the case of HVDs the merge curve need not be a simple chain or \( y \)-monotone. In general it is comprised of multiple, disjoint components that are not necessarily \( y \)-monotone and may in fact contain cycles (see Figure 1).

![Figure 1. Multiple Components In The Merge Curve For A Hausdorff Voronoi Diagram.](image)

### 3.2 Merging HVDs

In this section we outline our solution to the merging problem of two HVDs. Assume that \( S \) is split into two subsets \( S_l \) and \( S_r \), where all objects in \( S_l \) have their leftmost points to the left of all the points in objects in \( S_r \). Assume that we have already computed HVDs of \( S_l \) and \( S_r \) and our objective is to merge them to obtain the HVD of \( S \). 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 merge edges and merge vertices form both unbounded acyclic merge components and cyclic merge components. Together, all these edge-disjoint components form the merge curve. The merge curve partitions the plane into two portions, that which retains edges from the HVD of \( S_l \) and that which retains edges from the HVD of \( S_r \). The main idea is to use point location to locate the endpoints of Voronoi edges of one subdiagram in the other subdiagram and determine whether the subdiagram’s edge is a part of a merge chain or not. The main steps are as follows:

1. Use point location to find the subset of Voronoi Edges crossing the merge chain. Let these subsets be, \( E_{lm}^l \subset \)
of edges from HVD of $S_l$ and $E_m^r \subset E^r$ of edges from HVD of $S_r$. 

2. Find vertices of the merge chain on edges in $E_m^l$ and $E_m^r$.

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

4. Create a set of edge endpoints, two for each merge chain vertex. Globally sort endpoints. Connect adjacent endpoints to form edges and regions of the HVD. 

For point location each edge is treated independently. By performing point location of edges’ endpoints in the opposite subdiagram, we can determine, for each edge endpoint, which subdiagram is closer. Determining the closer subdiagram is equivalent to determining on which side of the merge curve an endpoint lies. Thus, this enables us to determine, independently for each edge, those edges which cross the merge curve (edges to be cropped), those which lie on the far side of the merge chain (edges to be removed), and those which lie on the close side (edges to be kept).

Once we have identified the set of edges involved in the merge chain, we must determine where the merge vertices occur on these edges. Again, this can be performed independently for each edge. Determining the merge vertex is equivalent to determining the input point from the opposite subdiagram inducing the merge vertex. However, we devise a variant of red blue line intersection algorithm 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 (See Figure 2).

After we have determined the merge vertices, it is sufficient to determine also the merge edges (on the merge curve) connecting these vertices. A merge vertex, $v$, is associated with two input points from the same side, say $l_1, l_2 \in S_l$, and the third input point is associated with the opposite site, say $r \in S_r$. We create two copies of $v$, one associate with the key $(l_1, r)$, and the other with the key $(l_2, r)$. By globally sorting the vertices using these keys, merge vertices sharing a merge edge will be adjacent, and a simple walk will complete the construction of the merge edges to form the merge curve.

### 3.3 Algorithm for finding merge vertices

Critical to the HVD algorithm is finding merge vertices. Before searching for merge vertices, we have already used point location to find a subset of edges from the left and right subdiagrams on which these vertices may occur. We also know whether the edges’ endpoints are closer to $S_l$ or $S_r$. In other words, we know the side of the merge curve on which an edge’s endpoint lie. By searching an edge $e_i$ from the HVD of $S_l$ through the regions in the HVD of $S_r$, we can find the region of the HVD of $S_l$ in which the merge vertex occurs. Since the intersection of the HVD of $S_r$’s edges with $e_i$ provides the boundaries of these regions, we can do binary search among these intersections to find the merge vertices.

In other words the above problem transforms to the following problem. Input to the problem is a set of non-intersecting red segments (edges of the HVD of objects in $S_l$) and a set of non-intersecting blue segments (edges of HVD of objects in $S_r$). Let the set of 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 (or to the left of) some point. We assume that there is some aboveness relation on a red query segment’s intersection with a blue segments, which allow us to perform binary search. We call this problem the batched red-blue segment search problem. One way to solve this search problem is to compute all intersections between red and blue segments and then perform the binary search. Obviously we want to avoid this as the total number of intersections could be quadratic in the number of segments.

Chazelle et al [3] describe a sequential algorithm for solving the red-blue intersection counting and reporting problem that uses a hereditary segment tree. We will review this data structure and present modifications to show how it can used for solving our search problem. First we present a sequential algorithm and then outline a CGM algorithm.

#### 3.3.1 Sequential algorithm for the search problem

The hereditary segment tree is defined for a set of red and blue line segments [3]. The $x$-coordinates of the end-points of red and blue segments are ordered, forming a partitioning of $x$-intervals. Each interval is associated, in order, with the leaves of a balanced binary tree. Inner nodes are assigned the interval which is the union of the interval associated to its two children. 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. We compare a node’s two long catalogs, as well as short catalog with a long catalog. Note that segments within a long catalog are ordered vertically and whenever we make a comparison, we ensure that at least one of the catalogs is long. This enables us to do the binary search without actually computing all the intersections and the actual mechanics is detailed next.

Let the set of red segments be our queries. For a red segment, $e_r$, we wish to search for some blue segment $e_b$ which intersects $e_r$ directly above some point. 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 just order a node’s short blue catalog, making it difficult to efficiently search $e_r$ against a short blue catalog. However, we use a secondary structure to extend the hereditary segment tree. 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 blue 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 exactly when the blue segment intersects all of long red segments associated to that node, but not all of the red segments at that nodes 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 order the short blue catalog at each node by the order in which they cross the long red segments.

The query for a red segment $e_r$ proceeds as follows. We traverse the segment tree, looking at the $O(\log n)$ nodes where $e_r$ is stored in red catalogs. Regardless of whether $e_r$ is short or long, we 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 that we find, we perform the required binary search. The running time of the search algorithm is dominated by the time spent in searching 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 catalogs requires a total of $O(n \log^3 n)$ time.

Lemma 1 The batched red-blue segment search problem for a set of $n$ segments can be solved in $O(n \log^3 n)$ time using $O(n \log^2 n)$ space.

3.3.2 CGM Algorithm for the search problem

Our data structure is composed of a main segment tree and at each node of this tree we have associated a secondary segment tree. We distribute the trees in this structure across processors. The main segment tree is divided into a top portion, $T_0$, comprising of the top $O(\log p)$ levels of the main tree. What remains of the main segment tree after removing $T_0$ is its $p$ subtrees, $T_1, T_2, \ldots, T_p$. Each $T_i$ portion is small enough to reside at a processor and is treated as a sequential subproblem. The skeleton of the tree $T_0$ is stored at each processor, as it of size $O(p)$, and it facilitates the queries. However, the catalogs associated with $T_0$’s nodes must be distributed across processors. We sort the entries in the catalogs globally across processors (first by the node of $T_0$ and then by the rank in the catalog). As a result, the catalogs which are shared, each processor stores only contiguous portions of catalogs. Boundaries of these $O(p)$ catalog portions are copied to all processors.

To complete a description of our distributed structure, it remains to divide the $T_0$ nodes’ secondary trees. Let the $j$th node, $n_j$, in $T_0$ have a secondary tree $\tau_j$. We repeat the previous technique and split the secondary tree into a top piece, $\tau_0^j$ of depth at most $\log p$. The remaining pieces of this secondary tree, $\tau_1^j, \ldots, \tau_k^j, \ldots, \tau_p^j$ are small enough to be treated sequentially. Again, the catalogs for the upper $\tau_0^j$ portion are distributed among processors.

Next we describe how we query this distributed data structure. Each query follows a path down the main segment tree. For each node $n_j$ in this path, the query segment also follows a path through the node’s associated secondary segment tree. Let us focus our discussion on the top (shared) $\tau_0^j$ portions of the secondary trees. These trees’ catalogs can be concatenated into a global sequence, sorted by key $(n_j, s_k, e_b)$, where $n_j$ is a segment tree node, $s_k$ is a secondary node in $n_j$’s secondary tree, and $e_b$ is a short blue catalog entry at $s_k$. Then we determine for each long red segment $e_r$, 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 $e_r$ with the key $(n_j, s_k, e_b)$ is created, and then the query for the tree $\tau_0^j$ is completed by performing parallel binary search.

Now, let us briefly discuss the lower $\tau_k^j$ portions of the secondary trees. Each processor stores a set of these lower secondary subtree portions. We can first load balance these subtrees across processors and the short blue segments destined for each subtree, and then solve each search problem sequentially. This load balancing is done using similar techniques as in other CGM algorithms using distributed segment trees [7].

Queries against long blue catalogs in the main segment tree do not require use of the secondary trees. These queries in $T_0$ are treated similarly to the queries performed in the secondary trees. Note, however, that the main segment tree’s subtrees do not require load balancing, since the nodes’ intervals are based on red and blue segment endpoints. Hence, $O(n^2)$ queries are distributed to each $T_i$ subtree.

Now we analyze the complexity of our algorithm. The most complex portion of $T_0$ is the secondary catalog query-
ing. The upper $\tau_0^i$ portions of the secondary trees reduce to sequential batch binary search subproblems of total size $O(\frac{n \log^2 p}{p})$, which require $O(\frac{n \log n \log^2 p}{p})$ local computation time. The lower $\tau_1^i$ portions of the secondary trees and the subtrees in $T_I$ reduce to sequential subproblems of total size $O(\frac{n \log^2 n}{p})$ per processor, which require $O(\frac{n \log^2 n}{p})$ local computation time. Hence,

**Lemma 2** The batched red-blue segment search problem can 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 $n \geq p^3$.

### 3.3.3 Finding merge vertices

We now return to the CGM algorithm for Hausdorff Voronoi diagrams. To complete this algorithm’s description, what remains is to describe how to find the merge vertices using the batched red-blue segment search problem. Recall that we have already computed HVDs of objects in $S_l$ and $S_r$ and our problem is to merge them to obtain HVD of objects in $S$. First, for simplicity, we restrict our attention to finding merge vertices on edges from the HVD of $S_l$ since the operation is symmetric for the edge in HVD of $S_r$. Recall that we have determined a set, $E_m$, of edges to search for such merge vertices. Furthermore, we can assume that the set $E_m$ 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). Here are the main steps:

1. Construct the extended hereditary segment tree data structure for the edges in the HVD of objects in $S_l$.
2. Query the edges in the sets $E_1^l$ and $E_2^l$ in the segment tree to find merge vertices on these edges.
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 the above steps for the finding merge vertices on edges in HVD of $S_r$.

Each merge chain crossing point is a vertex in the merged diagram. Hence, it has exactly three input points associated with it. Two of these points are from the same side (they induce the Voronoi edge, $e \in E_m$, which was crossed), say $p_i, p_j$ from objects in $S_l$. The third input point is from the other side, say $r$ from $S_r$. 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 HVD of $S_r$ with $e$. If these intersecting edges are ordered along $e$, then adjacent edges will define the boundary of Voronoi regions in the HVD of $S_r$. We can compute the Hausdorff distance from each intersection to objects in $S_l$ and to objects in $S_r$ (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 $S_l$ and the other edge intersecting it closer to $S_r$.

**Figure 2. Searching an edge $q$ in $E_1^l$ for the merge vertex $v$. It is equidistant from $r, p_i$, and $p_j$. Points on the segment $vq$ are closer to an object in $S_l$, whereas points on segment $vr$, are closer to an object in $S_r$.**

For a singly-intersected edge $q = (v_1, v_r) \in E_1^l$, we have determined an endpoint closer to $S_l$, say $v_1$, and an endpoint closer to $S_r$, say $v_r$, and we are looking for a merge vertex, say $v$ on $q$ (see Figure 2 for an illustration). We note that the red blue line intersection search, will search the intersection of edges between red lines (edges in $E_1^l$) and blue lines (edges in $E^r$) using an abovenss relation. For a singly-intersected edge, $q$, and a blue edge $e$, we define this relation as follows. Given an intersection point, say $v_{qe}$, between $q$ and $e$, we determine the Hausdorff distance between $v_{qe}$ and points defining the edges $q$ (i.e., $p_i$) and $e$. If the distance to $p_i$ is smaller then we perform the search on the segment $v_{qe}v_r$, otherwise we search on the segment $v_1v_{qe}$. Notice that at the merge vertex $v$ the distance to $p_i$ is same as the distance to $r$.

For an edge $q \in E_2^l$ there may or may not be two merge vertices on $q$. We know that $q$’s two endpoints, say $v_1$ and $v_2$, are both farther from $S_l$ than $S_r$. Hence, if there exist two merge vertices, they partition $q$ into three parts, the middle part is closer to $S_l$ and the two end parts closest to $S_r$. For this case, we make use of a lemma from [14, 18].

**Lemma 3 ([14, 18])** Let $T(P_i)$ be the tree formed by the edges of the farthest point Voronoi diagram of points in an object $P_i \in S$. Let there be a point $a \in T(P_i)$. The point $a$ splits $T(P_i)$ into two subtrees. If an object $P_j \in S$ is closer
to a than $P_i$ is to $a$, then all the points in one of the two
subtrees are closer to $P_j$ than to $P_i$.

We must perform two red blue line intersection searches
on $q$, one for each potential merge vertex. Let us describe
the search for the merge vertex which is closer to $v_1$, and
let us call this merge vertex as $v$. The search for the other
merge vertex is analogous. Here, we define a slightly dif-
ferent aboveness relation on the edge $q$. For the intersection
point $v_{qe}$ between the edges $q$ and $e$, if $v_{qe}$ is closer to $S_l$
then we are in the middle region and need to search
towards $v_1$. Otherwise, we are closer to $S_r$ and we need
to determine whether we are above or below $v$. By using
Lemma 3 and the knowledge about which object in $S_r$
is closest to $v_{qe}$, we can determine whether to search towards
$v_1$ or $v_2$.

3.4 Summary

**Theorem 1** On a $p$-processor CGM, the Hausdorff Voronoi
diagram of non-crossing objects defined by $n$-points in the
plane can be constructed in $O\left(\frac{nl\log^2 n}{p}\right)$ local computation
time, in $O(\log p)$ rounds with $O\left(\frac{n\log^2 n}{p}\right)$ space per pro-
cessor, where $n \geq p^3$.

4 Improved Sequential Algorithm

The existing divide and conquer (Papadopoulou [18])
and sweepline (Papadopoulou and Lee [14]) algorithms
have worst-case running times of $O(n^2 \log n)$ for the non-
crossing case and are not optimal. In fact, Papadopoulou
and Lee poses this as an open problem. For our par-
allel 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.

Our sequential algorithm is derived as follows. We re-
peat the same divide and conquer technique as in the exist-
ing Hausdorff Voronoi divide-and conquer algorithm [18],
except we improve the merge step by doing a sequential
version of the CGM algorithm’s merge. For this, a se-
quential $O(n \log n)$ batch point location algorithm and the
$O(n \log^3 n)$ batched red blue line search algorithm from
above is used. By performing $O(\log n)$ merge steps, we
obtain the following result.

**Theorem 2** A Hausdorff Voronoi diagram for non-crossing
objects defined on $n$-points in the plane can be constructed
in $O(n \log^3 n)$ time.

5 Preliminary Experimental Results

In this section we discuss some preliminary experi-
mental results from a “first draft” implementation of our
parallel algorithm in Section 3. We stress that our paral-
lel implementation is still a work in progress and requires
considerably more fine tuning before a speedup consistent
with the theoretical analysis can be measured. We are still
working on an improved version of our code but decided to
include these preliminary results in order to meet the con-
ference deadline.

Our code includes a sequential sweepline algorithm for
HVDs, segment trees for parallel point location, segment
trees for red blue line intersection search, and a distributed
HVD data structure. We make use of two libraries: CGM-
Lib [2] and LEDA [11]. CGMLib builds on MPI to pro-
vide efficient algorithms and memory management methods
in C++ that are particularly well suited for coarse grained
parallel algorithms. All communication between proces-
sors occurs using CGMLib, and we make use of CGM-
Lib’s parallel sorting algorithms. The Library of Efficient
Data Structures and Algorithms (LEDA) implements many
fundamental algorithms, including some computational ge-
ometry methods. 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 diagram construction algorithm.
We also make use of LEDA’s data structures for graphs, pri-
ority queues and hash tables. The current implementation
of our new methods and data structures in Section 3 is still
rather straightforward and still needs a lot of optimization
work. In particular, our current implementation of the seg-
ment tree with secondary segment trees still incurs a lot of
overhead. We also discovered considerable redundancies in
our current implementation of the data movements required
for load balancing. During the merge phase, the parallel
point location and red blue line intersection modules con-
struct large trees and perform many queries on these trees.
These computations, along with the initial sequential phase
are the main contributors to running time in this preliminary
implementation. They need a better implementation with
more attention to detail. We are working on a new “version
2” of our code.

Figures 3 and 4 show speedups and wall clock run-
going times, respectively, for our current preliminary im-
plementation. These tests were performed on a Beowulf
cluster in the High Performance Virtual Computing Lab-
ory (www.HPCVL.org). The cluster consists of dual-
processor Xeon nodes (2.0GHz and 1.5GB RAM per node)
with LINUX Redhat and LAM MPI 7.1.1. For our tests,
we used only one processor per node to avoid artifacts in
our measurements from intra-node communication. For our
input data, we wrote a data generator which creates pseudo
random patterns that mimic via blocks placed on a VLSI
chip. Preliminary relative speedup results are shown in Fig-
ure 3. The speedups themselves are still far too low due
to our non optimized implementation. However, the shape
of the speedup curves is encouraging since the curves are close to linear shape for up to 16 processors. That is, they do not “drop off” at some point within this range which indicates good scalability. This is consistent with the preliminary wall clock times shown in Figure 4. We observe that the communication time decreases with increasing number of processors which indicates good scalability.

References


