UNIVERSITY OF CALIFORNIA
RIVERSIDE

Scaling Spatial Overlay Operations and Flock Pattern Discovery

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Andres Oswaldo Calderon Romero

November 2024

Dissertation Committee:

Dr. Vassilis Tsotras, Chairperson
Dr. Amr Magdy
Dr. Petko Bakalov
Dr. Ahmed Eldawy
Dr. Vagelis Hristidis

Chapter 1

Scalable Overlay Operations over DCEL Polygon Layers

1.1 Introduction

The use of spatial data structures is ubiquitous in many spatial applications, ranging from spatial databases to computational geometry, robotics, and geographic information systems [27]. Spatial data structures have been used to improve the efficiency of various spatial queries, spatial joins, nearest neighbors, Voronoi diagrams, and robot motion planning. Examples include grids [21], R-trees [15, 2], and quadtrees [10]. *Edge-list* structures are also typically utilized in applications as topological computations in computational geometry [4].

The most commonly used data structure in the edge-list family is the *Doubly Connected Edge List (DCEL)*. A DCEL [20, 23] is a data structure that collects topological information for the edges, vertices, and faces contained by a surface in the plane. The DCEL

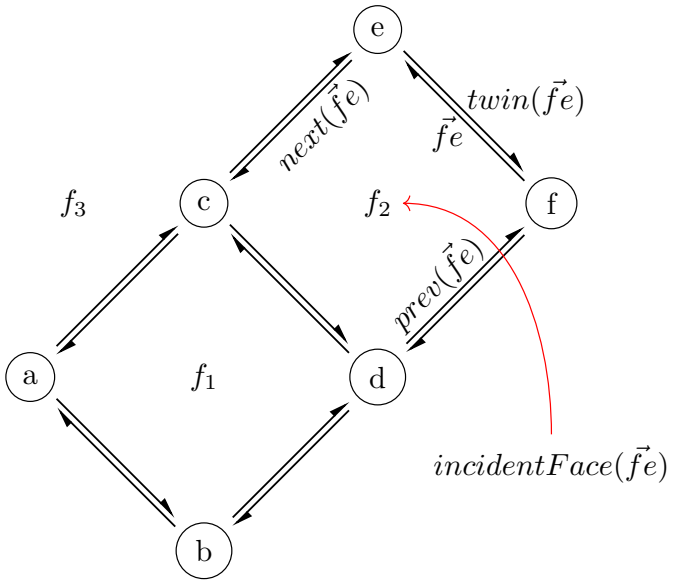


Figure 1.1: Components of the DCEL structure.

and its components represent a planar subdivision of that surface. In a DCEL, the faces (polygons) represent non-overlapping areas of the subdivision; the edges are boundaries that divide adjacent faces; and the vertices are the point endings between adjacent edges (see Figure 1.1). In addition to providing geometric and topological information, a DCEL can be enhanced to provide further information. For instance, a DCEL storing a thematic map for vegetation can also store the type and height of the trees around the area [4].

The DCEL data structure has been used in various applications. For instance, the use of connected edge lists is cardinal to support polygon triangulations and their applications in surveillance (the Art Gallery Problem [9, 22]) and robot motion planning ([4, 8]). DCELs are also used to perform polygon unions (for example, on printed circuit boards to support the simplification of connected components in an efficient manner [11]) as well as the computation of silhouettes from polyhedra [11, 3] (applied frequently in computer

vision and 3D graphics modeling [5]).

Edge-list data structures have also been utilized to create thematic *overlay maps*. In this problem, the input contains the DCELs of two polygonal layers, each capturing geospatial information and attribute data for different phenomena, and the output is the DCEL of an overlay structure that combines the two layers into one. In many application areas, such as ecology, economics, and climate change, it is important to be able to join the input layers and match their attributes in order to unveil patterns or anomalies in data that can be highly impacted by location. Several operations can then be easily computed given an overlay; for instance, the user may want to find the *intersection* between the input layers (e.g., corresponding to soil types and evapotranspiration of plants), identify their *difference* (or symmetric difference), or create their *union*.

Spatial databases use spatial indexes (R-tree [15, 2]) to store and query polygons. Such methods use the *filter and refine* approach where a complex polygon is abstracted by its Minimum Bounding Rectangle (MBR); this MBR is then inserted in the R-tree index. Finding the intersection between two polygon layers, each indexed by a separate R-tree, is then reduced to finding the pairs of MBRs from the two indexes that intersect (filter part). This is followed by the refine part, which, given two MBRs that intersect, needs to compute the actual intersections between all the polygons these two MBRs contain. While MBR intersection is simple, computing the intersection between a pair of complex real-life polygons is a rather expensive operation (a typical 2020 US census tract is a polygon with hundreds of edges). Moreover, using DCELs for overlay operations offers the additional advantage that the result is also a DCEL, which can be directly used for subsequent operations. For

example, one may want to create an overlay between the intersection of two layers with another layer, and so on.

Even though the DCEL has important advantages for implementing overlay operations, current approaches are sequential in nature. This is problematic, considering layers with thousands of polygons. For example, the layer representing the 2020 US census tracts contains around 72K polygons; the execution for computing the overlay over such a large file crashed on a stock laptop. To the best of our knowledge, there is no scalable solution for computing overlays over DCEL layers.

This chapter describes the design and implementation of a *scalable* and *distributed* approach to compute the overlay between two DCEL layers. We first present a partitioning strategy that guarantees that each partition collects the required data from each layer DCEL to work independently, thus minimizing duplication and transmission costs over 2D polygons. In addition, we present a merging procedure that collects all partition results and consolidates them in the final combined DCEL.

Implementing a distributed overlay DCEL creates novel problems. First, there are potential challenges that are not present in the sequential DCEL execution. For example, the implementation should consider *holes*, which could lay on different partitions, and they need to be connected with their components residing in other partitions so as not to compromise the combined DCEL's correctness.

Secondly, once a distributed overlay DCEL has been built, it must support a set of binary overlay operators (namely *union, intersection, difference* and *symmetric difference*) in a transparent manner. That is, such operators should take advantage of the scalability of

the overlay DCEL and be able to run also in a parallel fashion. Additionally, users should be able to apply the various operators multiple times without rebuilding the overlay DCEL data structure.

The rest of this chapter is organized as follows. Section 1.2 presents related work, while Section 1.3 discusses the basics of DCEL and the sequential algorithm. In Section 1.4, we present the partitioning schemes that enable parallel implementation of the overlay computation among DCEL layers; we also discuss the challenges presented in the DCEL computations by distributing the data and how to solve them efficiently. Two important optimizations are introduced in Section 1.5. Finally, an extensive experimental evaluation appears in Section 1.6.

1.2 Related Work

The fundamentals of the DCEL data structure were introduced in the seminal paper by Muller and Preparata [20]. The advantages of DCELs are highlighted in [23, 4]. Examples of using DCELs for diverse applications appear in [1, 6, 13].

Once the overlay DCEL is created by combining two layers, overlay operators like union, difference, etc., can be computed in linear time to the number of faces in their overlay [13]. Currently, few sequential implementations are available: LEDA [19], Holmes3D [16] and CGAL [11]. Among them, CGAL is an open-source project widely used for computational geometry research. To the best of our knowledge, there is no scalable implementation for the computation of DCEL overlay.

While there is a lot of work on using spatial access methods to support spatial

joins, intersections, unions etc. in a parallel way (using clusters, multicores or GPUs), [7, 26, 17, 12, 18, 25, 24] these approaches are different in two ways: (i) after the index filtering, they need a time-consuming refine phase where the operator (union, intersection etc.) has to be applied on each pair of (typically) complex spatial objects; (ii) if the operator changes, we need to run the filter/refine phases from scratch (in contrast, the same overlay DCEL can be used to run all operators.)

1.3 Preliminaries

The DCEL [20] structure is used to represent an embedding of a planar subdivision in the plane. It provides efficient manipulation of the geometric and topological features of spatial objects (polygons, lines, and points) using *faces*, *edges*, and *vertices*, respectively. A DCEL uses three tables (relations) to store records for the faces, edges, and vertices, respectively.

An important characteristic is that all these records are defined using edges as the main component (thus termed an edge-based structure). Examples appear in Tables 1.1-1.3, with the subdivision depicted in Figure 1.1.

An edge corresponds to a straight line segment shared by two adjacent faces (polygons). Each of these two faces will use this edge in its description; to distinguish, each edge has two *half-edges*, one for each orientation (direction). It is important to note that half-edges are oriented counter-clockwise inside each face (Figure 1.1). A half-edge is thus defined by its two vertices, one called the *origin* vertex and the other the *target* vertex,

Table 1.1: Vertex records.

vertex	coordinates	incident edge
a	(0,2)	$\vec{ba}$
b	(2,0)	$\vec{db}$
c	(2,4)	$\vec{dc}$
⋮	⋮	⋮

Table 1.2: Face records.

	boundary	hole
face	edge	list
f_1	$\vec{ab}$	*nil*
f_2	$\vec{fe}$	*nil*
f_3	*nil*	*nil*

Table 1.3: Half-edge records.

half-edge	origin	face	twin	next	prev
$\vec{fe}$	f	f_2	$\vec{ef}$	$\vec{ec}$	$\vec{df}$
$\vec{ca}$	c	f_1	$\vec{ac}$	$\vec{ab}$	$\vec{dc}$
$\vec{db}$	d	f_3	$\vec{bd}$	$\vec{ba}$	$\vec{fd}$
⋮	⋮	⋮	⋮	⋮	⋮

clearly specifying the half-edge's orientation (origin to target). Each half-edge record contains references to its origin vertex, its face, its *twin* half-edge, as well as the next and previous half-edges (using the orientation of its face); see Table 1.3. These references are used as keys to the tables that contain the referred attributes.

Figure 1.1 shows half-edge $\overrightarrow{fe}$, its *twin*($\overrightarrow{fe}$) (which is half-edge $\overrightarrow{ef}$), the *next*($\overrightarrow{fe}$) (half-edge $\overrightarrow{ec}$) and the *prev*($\overrightarrow{fe}$) (half-edge $\overrightarrow{df}$). Note the counter-clockwise direction used by the half-edges comprising face f_2. The *incidentFace* of a half-edge corresponds to the face that this edge belongs to (for example, *incidentFace*($\overrightarrow{fe}$) is face f_2).

Each vertex corresponds to a record in the vertex table (see Table 1.1) that contains its coordinates as well as one of its incident half-edges. An incident half-edge is one whose target is this vertex. Any of the incident edges can be used; the rest of a vertex's incident half-edges can be found easily following the next and twin half-edges.

Finally, each record in the faces table contains one of the face's half edges to describe the polygon's outer boundary (following this face's orientation); see Table 1.2. All other half-edges for this face's boundary can be easily retrieved following the next half-edges in orientation order. In addition to regular faces, there is one face that covers the area outside all faces; it is called the *unbounded* face (face f_3 in Figure 1.1). Since f_3 has no boundary, its boundary edge is set to *nil* in Table 1.2.

Note that polygons can contain one or more *holes* (a hole is an area inside the polygon that does not belong to it). Each such hole is described by one of its half-edges; this information is stored as a list attribute (hole list) in the faces table where each element of the list is the half-edge's id which describes the hole. Note that in Table 1.2, this list is

empty as there are no holes in any of the faces in the example of Figure 1.1.

An important advantage of the DCEL structure is that a user can combine two DCELs from different layers over the same area (e.g., the census tracts from two different years) and compute their *overlay*, which is a DCEL structure that combines the two layers into one. Other operators, like the intersection, difference, etc., can then be computed from the overlay very efficiently. Given two DCEL layers S_1 and S_2, a face f appears in their overlay $OVL(S_1, S_2)$ if and only if there are faces f_1 in S_1 and f_2 in S_2 such that f is a maximal connected subset of $f1 \cap f2$ [4]. This property implies that the overlay $OVL(S_1, S_2)$ can be constructed using the half-edges from S_1 and S_2.

The sequential algorithm [11] to construct the overlay between two DCELs first extracts the half-edge segments from the half-edge tables and then finds intersection points between half-edges from the two layers (using a sweep line approach) [4]. The intersection points found will become new vertices of the resulting overlay. If an existing half-edge contains an intersection point, it is split into two new half-edges. Using the list of outgoing and incoming half-edges for the newly added vertices (intersection points), the algorithm can compute the attributes for the records of the new half-edges. For example, the list of outgoing and incoming half-edges at each new vertex will be used to update the next, previous, and twin pointers. Finally, the records of the faces and the vertices tables are updated with the new information.

Figure 1.2 illustrates an example of computing the overlay between two DCEL layers with one face each (A_1 and B_1 respectively) overlapping the same area. First, intersection points are identified, and new vertices are created in the overlay (red vertices c_1

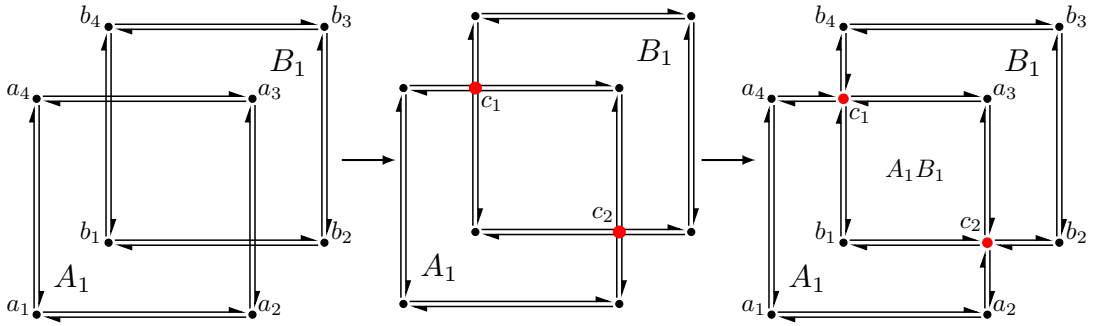


Figure 1.2: Sequential computations of an overlay of two DCEL layers.

and c_2). Then, new half-edges are created around these new vertices. As a result, face A_1 is modified (to an L-shaped boundary), as does face B_1, while a new face A_1B_1 is created. Since this new face is the intersection of the boundaries of A_1 and B_1, its label contains the concatenation of both face labels. By convention [4], even though A_1 changes its shape, it does not change its label since its new shape is created by its intersection with the unbounded face of B_1; similarly, the new shape of B_1 maintains its original label. These labels are crucial for creating the overlay (and the operators it supports) as they are used to identify which polygons overlap an existing face.

Once the overlay structure of two DCELs is computed, queries like their intersection, union, difference, etc. (Figure 1.3) can be performed in linear time to the number of faces in the overlay. The space requirement for the overlay structure remains linear to the number of vertices, edges, and faces. Since an overlay is itself a DCEL, it can support the traditional DCEL operations (e.g., find the boundary of a face, access a face from an adjacent one, visit all the edges around a vertex, etc).

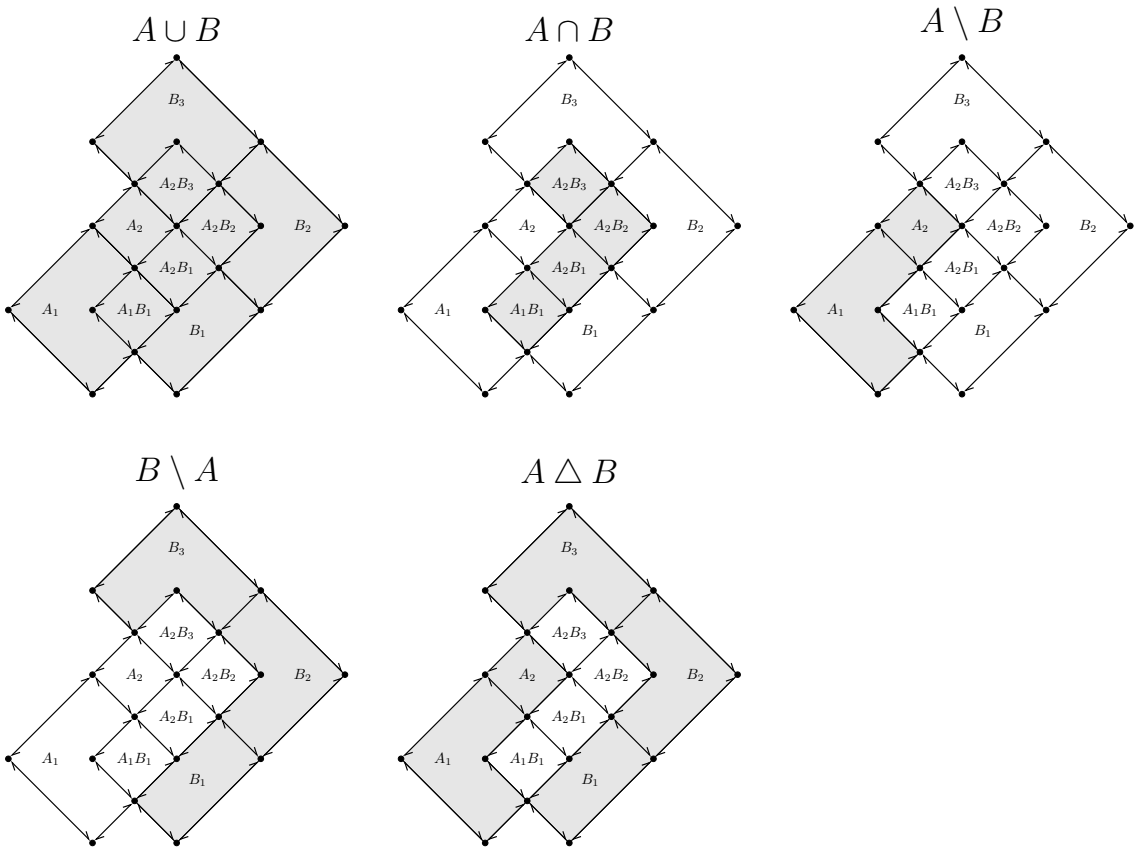


Figure 1.3: Examples of overlay operators supported by DCEL; results are shown in gray.

1.4 Scalable Overlay Construction

This section presents the construction of overlay DCELs, assuming 2D polygons as input. The overlay computation depends on the size of the input DCELs and the size of the resulting overlay. The DCEL of a planar subdivision S_1 has size $O(n_1)$ where $n_1 = \Sigma(vertices_1 + edges_1 + faces_1)$. The sequential algorithm constructing the overlay of S_1 and S_2 takes $O(n \log n + k \log n)$ time, where $n = n_1 + n_2$ and k is the size of their overlay. Note that k depends on how many intersections occur between the input DCELs, which can be very large [4].

While the sequential algorithm is efficient with small DCEL layers, it suffers when the input layers are large and have many intersections. For example, creating the overlay between the DCELs of two census tracts (from years 2000 and 2010) from California (each with 7K-8K polygons and 2.7M-2.9M edges) took about 800sec on an Intel Xeon CPU at 1.70GHz with 2GB of memory (see Section 1.6). With DCELs corresponding to the whole US, the algorithm crashed.

Nevertheless, the overlay computation can take advantage of ***partitioning*** (and thus parallelism) by observing that the edges in a given area of one input layer can only intersect with edges from the same area in the other input layer. One can thus spatially partition the two input DCELs and then compute the overlay within each cell; such computations are independent and can be performed in parallel. While this is a high-level view of our scalable approach, there are various challenges, including how to deal with edges that cross cells, how to manage the extra complexity introduced by *orphan* holes (i.e., when holes and their polygons are in different cells), how and where to combine partition overlays

into a global overlay, as well as how to balance the computation if one layer is much larger than the other.

1.4.1 Partition Strategy

The main idea of the quadtree partition strategy is to split the area covered by the input layers into non-overlapping cells, which can then be processed independently. While a simple grid could be used to divide the spatial area, our early experiments demonstrated that this approach leads to unbalanced cells, with some containing significantly more edges than others, negatively impacting overall performance. In the rest we assume that the partitioning is performed using a quadtree index wich adapts to skewed spatial distributions and helps to assign a similiar number of edges to each cell.

The overall approach can be summarized in the following steps: (i) Partition the input layers into the index cells and build local DCEL representations of them at each cell, and (ii) Compute the overlay of the DCELs at each cell. Overlay operators and other functions can be run over the local overlays, and local results are collected to generate the final answer.

Note that each input layer is given as a sequence of polygon edges, where each edge record contains the coordinates of the edge's vertices (origin and target vertex) as well as the polygon id and a hole id in the case that an edge belongs to a hole inside of a polygon. We assume there are no overlapping or stacked polygons in the dataset.

To quickly build the partitioning quadtree structure, we build a quadtree from a sample taken from the edges of each layer (1% of the total number of edges in that layer). We then use the leaves of that quadtree as the cells (partitions) of the partitioning scheme.

These cells will be used to assign the edges of each input layer. Populated cells are then distributed to the available nodes for processing the overlay operations.

To support the creation of the quadtree we use the sampling functionalities provided in the Apache Sedona, an extension available on the Apache Spark platform. It allows the user to provide a parameter for the number of quadtree leaves; using this parameter as an approximation, it builds a quadtree; it should be noted that the actual number of leaves created is typically larger than the parameter provided by the user. The number of user-requested leaves and the size of the sample are used to compute the maximum number of entries per node (capacity) during the construction of the tree. If the node capacity is exceeded, the node is divided into four child nodes with an equal spatial area, and its data is distributed among the four child nodes. If any child node has exceeded its capacity, it is further divided into four nodes recursively and so on, until each node holds at most its computed *capacity*.

After creating the quadtree from the sample, we use its leaf nodes as the partitioning cells for each layer. Each input layer file is then read from the disk, and *all* its edges are inserted into the appropriate cells of the partitioning structure. Note that the partitioning structure created from the sample is now fixed; no more cells are created when the layer edges are assigned to cells. In the rest, we use the term cell and partition interchangeably.

For this approach to work, it is important that each cell can compute its two DCELs independently. An edge can be fully contained in a cell, or it can intersect the cell's boundary. In the second case, we copy this edge to all cells where it intersects, but within each cell, we use the part of the edge that lies fully inside the cell. Figure 1.4 shows an

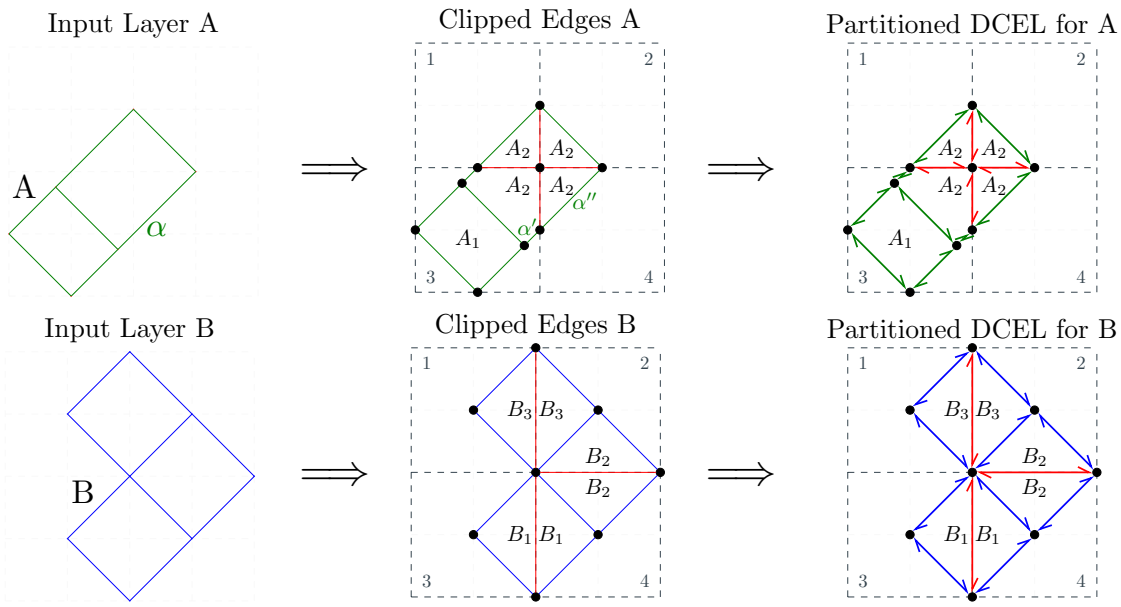


Figure 1.4: Partitioning example using input layers A and B over four cells.

example where four cells and two edges of the upper polygon from layer A cross the cell borders. Such edges are clipped at the cell borders, introducing new edges (e.g., edges α' and α'' in the Figure 1.4). Similarly, a polygon that crosses over a cell is clipped to the cell by introducing ***artificial*** edges on the cell's border (see face A_2 in cell 3 of Figure 1.4). Such artificial edges are shown in red in the figure. This allows for the creation of a smaller polygon that is contained within each cell.

For example, polygon A_2 is clipped into four smaller polygons as it overlaps all four cells. The clipping of edges and polygons ensures that each cell has all the needed information to complete its DCEL computations. As such computations can be performed independently, they are sent to different worker nodes to be processed in parallel. The assignment is delegated to the distributed framework (i.e., Apache Spark).

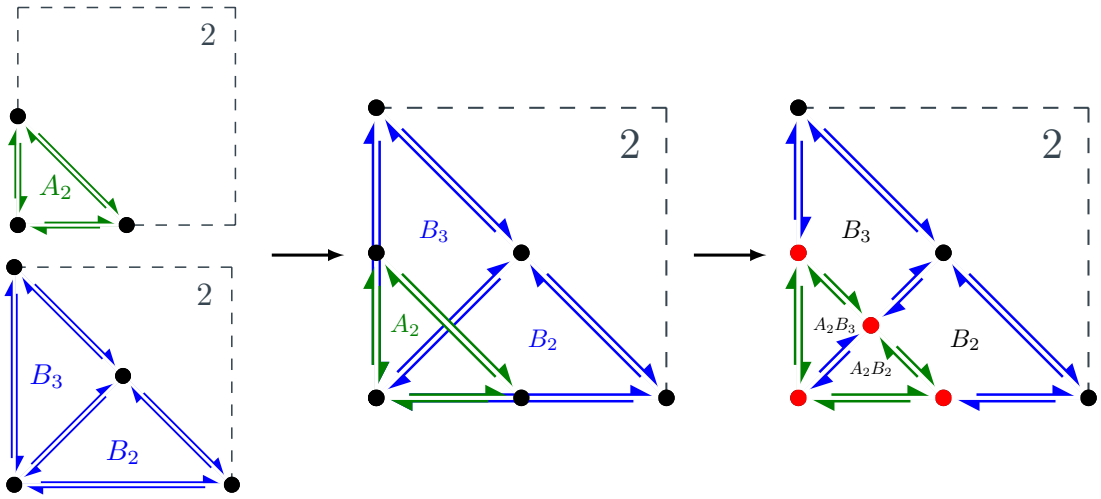


Figure 1.5: Local overlay DCEL for cell 2.

Once a cell is assigned to a worker node, the sequential algorithm is used to create a DCEL for each layer (using the cell edges from that layer and any artificial edges, vertices, and faces created by the clipping procedures above) and then compute the corresponding (local) overlay for this cell. Using the example from Figure 1.4, Figure 1.5 depicts an overview of the process for creating a local overlay DCEL inside cell 2. Similarly, Figure 1.6 shows all local overlay DCELs computed at each cell (artificial edges are shown in red).

Nevertheless, the partitioning creates two problems (not present in the sequential environment) that need to be addressed. The first is the case where a cell is empty; it does not intersect with (or contain) any regular edge from either layer. A regular edge is not part of a hole. This empty cell does not contain any label, and thus, we do not know which face it may belong to. We term this as the ***orphan cell*** problem. An example is shown in Figure 1.7, which depicts a face (from one of the input layers) whose boundary goes over

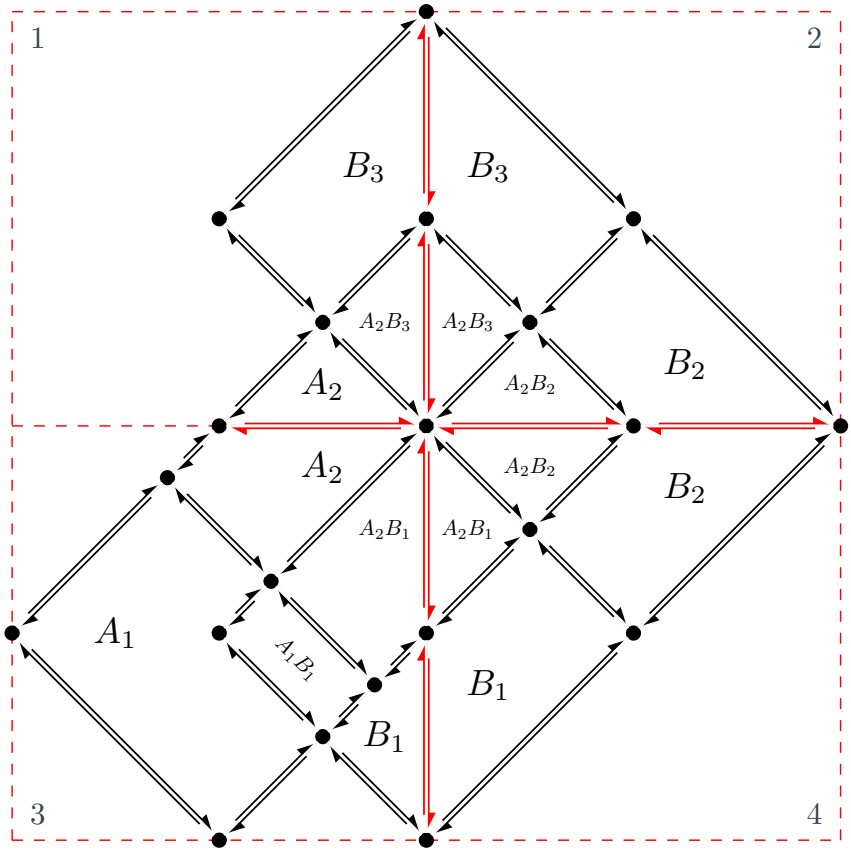


Figure 1.6: Result of the local overlay DCEL computations.

many quadtree cells; orphan cells are shown in grey.

Note that an orphan cell may contain a hole (see Figure 1.7). In this case, the original label of the face where the hole belongs (and reported in the hole's edges) may have changed during the overlay computation (because it overlapped with a face from the other layer). However, this new label has not been propagated to the hole edges. We term this as the ***orphan hole*** problem. For simplicity, we focus on the case where a hole is within one orphan cell, but in the general case, a hole can split among many such cells.

The issue with both 'orphan' problems is the missing labels. In section 1.4.2, we propose an algorithm that correctly labels an orphan cell. If this cell contains a hole, the new label is also used to update the hole edges.

1.4.2 Labeling Orphan Cells and Holes

Assuming a quadtree-based partitioning, to find the label of an orphan cell, we propose an algorithm that recursively searches the space around the orphan cell until it identifies a nearby cell that contains an edge(s) of the face that includes the orphan cell and thus acquire the appropriate label information. The quadtree index accommodates this search. Two observations are in order: (1) each cell is a leaf of the quadtree index (by construction), and (2) each cell has a unique id created by the way this cell was created; this id effectively provides the *lineage* (unique path) from the quadtree root to this leaf.

Recall that the root has four possible children (typically numbered as 0,1,2,3 corresponding to the four children NW, NE, SW, and SE). The lineage is the sequence of these numbers in the path to the leaf. For example, the lineage for the shaded orphan cell in Figure 1.7(a) is 03. Further, note that the quadtree is an unbalanced structure, having more

deep leaves where there are more edges. Thus, higher leaves correspond to larger areas, and deeper leaves correspond to smaller areas (since a cell split is created when a cell has more edges than a threshold).

After identifying an orphan cell, the question is where to search for a cell containing an edge. The following Lemma applies:

Lemma 1 *Given an orphan cell, one of its siblings at the same quadtree level must contain a regular edge (directly or in its subtree).*

This lemma arises from the simple observation that if all three siblings of an orphan cell are empty, then there is no reason for the quadtree to make this split and create these four siblings. Based on the lemma, we know that at least one of the three siblings of the orphan cell can lead us to a cell with an edge. However, these siblings may not be cells (leaves). Instead of searching each one of them in the quadtree until we reach their leaves, we want a way to quickly reach their leaves. To do so, we pick the centroid point of the orphan cell's parent (which is also one of the corners of the orphan cell).

For example, the parent centroid for the orphan cell 03 is the green point in Figure 1.7(b). We then query the quadtree to identify which cells (leaves, one from each sibling) contain this point. We check whether these cells contain an edge; if we find such a cell, we stop (and use the label in that cell). If all three cells are orphans, we need to continue the search. An example appears in Figure 1.7(b), where all three cells (green in the figure) are also orphans.

We first check if any of these orphan cells is a sibling (has the same parent) of the original cell. In this case that sibling is also a leaf (i.e. it does not have a subtree) and

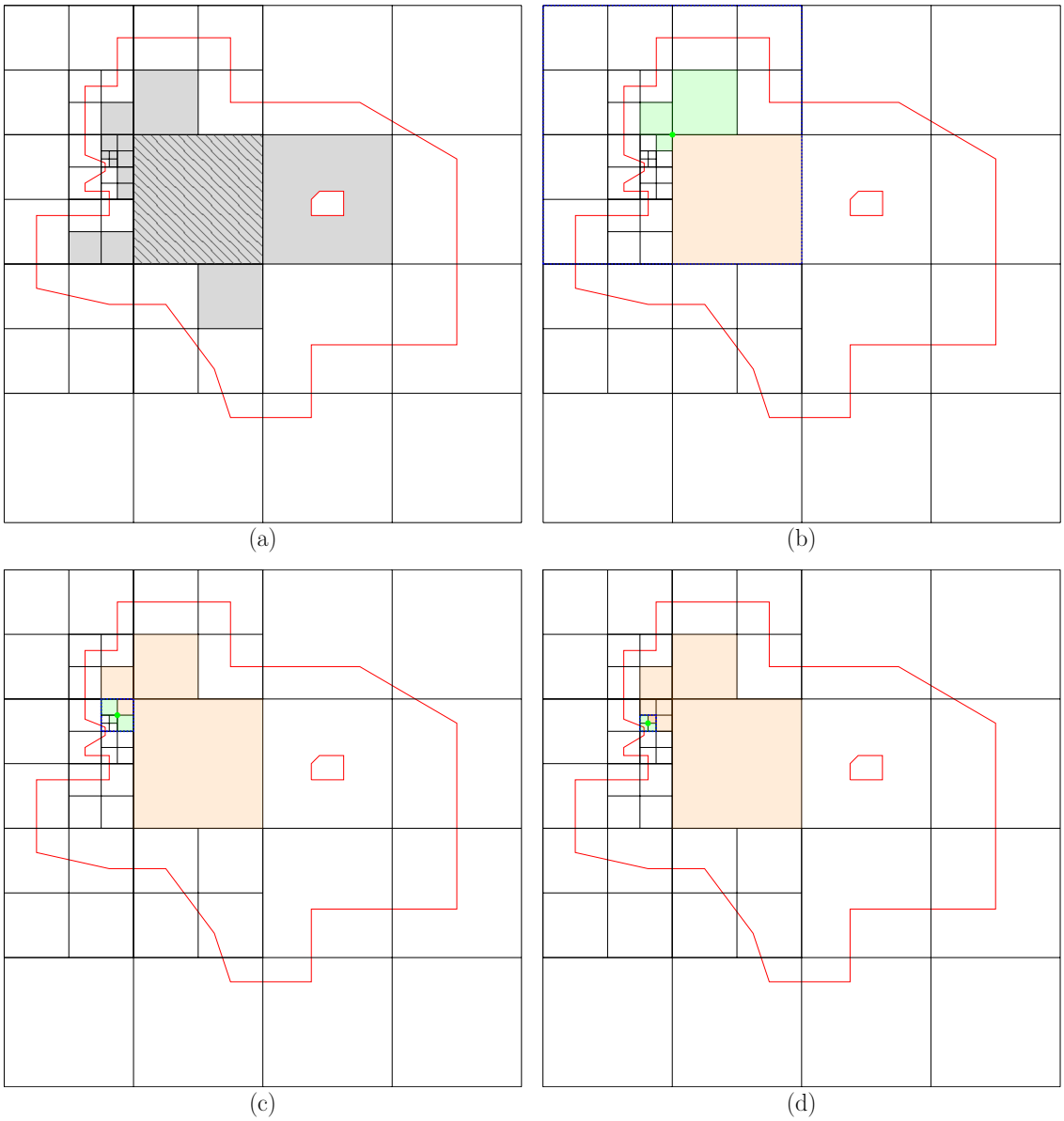


Figure 1.7: (a) Empty cell and hole examples; (b)-(c)-(d) show three iterations of the proposed solution.

does need to be explored. The remaining orphans are therefore at a lower level than the original orphan cell, which means they come from a sibling that has been split because of some edge. The algorithm picks any of the remaining orphan cells to continue. In Figure 1.7(b) all three leaves (green orphan cells) are at a lower level than the original orphan cell.

One can use different heuristics to pick which of the remaining leaves to use. Below, we consider the case where we use the deepest cell (i.e., the one with the longest lineage) among the leaves. This is because we expect this to lead us to the denser areas of the quadtree index, where there is more chance to find cells with edges. Figure 1.7 shows a three-iteration run of the algorithm.

During the search process, we keep any orphan cells we discover; after a cell with an edge (non-orphan cell) is found, the algorithm stops and labels the original orphan cell and any other orphan cells retrieved in the search with the label found in the non-orphan cell. Note that if the non-orphan cell contains many labels (because different faces pass through it), we assign the label of the face that contains the original centroid.

The pseudo-code of the search process can be seen in Algorithms 1 and 2. Another heuristic we used that is not described here is to follow the highest among the three orphan cells; i.e. the one with the shorter lineage since this has a larger area and will thus help us cover more empty space and possibly reach the border of the face faster.

To determine the worst-case performance of the search algorithm, consider that for an orphan cell, the algorithm performs three point quadtree queries to find the sibling leaves containing the centroid. It then selects one of these leaves and repeats the process, querying three points for a new centroid within the siblings of the selected leaf. This causes

Algorithm 1 getNextCellWithEdges algorithm

Require: a quadtree $\mathcal{Q}$ and a list of cells $\mathcal{M}$.

```
1: function getNextCellWithEdges ( Q, M )
2:     C ← orphan cells in M
3:     for each orphanCell in C do
4:         initialize cellList with orphanCell
5:         nextCellWithEdges ← nil
6:         referenceCorner ← nil
7:         done ← false
8:         while ¬done do
9:             c ← last cell in cellList
10:            cells, corner ← getCellsAtCorner(Q, c)
11:            for each cell in cells do
12:                nedges ← get edge count of cell in M
13:                if nedges > 0 then
14:                    nextCellWithEdges ← cell
15:                    referenceCorner ← corner
16:                    done ← true
17:                else
18:                    if cell.level < orphanCell.level then
19:                        add cell to cellList
20:                    end if
21:                end if
22:            end for
23:        end while
24:        for each cell in cellList do
25:            output(cell,
26:                nextCellWithEdges, referenceCorner)
27:            remove cell from C
28:        end for
29:    end for
30: end function
```

Algorithm 2 GETCELLSATCORNER algorithm

Require: a quadtree $\mathcal{Q}$ and a cell c.
  function GETCELLSATCORNER ($\mathcal{Q}$, c)
    $region \leftarrow$ quadrant region of c in $c.parent$
    switch $region$ **do**
      case 'SW'
        $corner \leftarrow$ left bottom corner of $c.envelope$
      case 'SE'
        $corner \leftarrow$ right bottom corner of $c.envelope$
      case 'NW'
        $corner \leftarrow$ left upper corner of $c.envelope$
      case 'NE'
        $corner \leftarrow$ right upper corner of $c.envelope$
    $cells \leftarrow$ cells which intersect $corner$ in $\mathcal{Q}$
    $cells \leftarrow cells - c$
    $cells \leftarrow$ sort $cells$ on basis of their depth
    return ($cells$, $corner$)
  end function

the algorithm to explore progressively deeper into the quadtree. In the worst case, the longest path in the quadtree could result in a time complexity of $O(N)$. However, in the average case, when the quadtree is balanced, the complexity is logarithmic.

1.4.3 Answering global overlay queries

Using the local overlay DCELs, we can easily compute the global overlay DCEL; for that, we need a reduce phase, described below, to remove artificial edges, and concatenate split edges from all the faces. Using the local overlay DCELs, we can also compute in a scalable way global operators like intersection, difference, symmetric difference, etc. For these operators, there is first a map phase that computes the specific operator on each local DCEL, followed by a reduce phase to remove artificial edges/added vertices. Figure 1.8 shows how the intersection overlay operator ($A \cap B$) is computed, starting with the local

DCELs for four cells in Figure 1.8(a). First, each cell computes the intersection using its local overlay DCEL as shown in Figure 1.8(b). This is a map operation to identify overlay faces that contain both labels from layer A and layer B. Each cell can then report every such face that does not include any artificial edges, like face A_1B_1 in Figure 1.8(b); note that these faces are fully included in the cell.

Using a reduce phase, the remaining faces are sent to a master node; in our implementation, it would be the driver node of the spark application that will (i) remove the artificial edges, shown in red in the figure and (ii) concatenate edges that were split because they were crossing cell borders. This is done by pairing faces with the same label and concatenating their geometries by removing the artificial edges and vertices added during the partition stage, for example, the two faces with label A_2B_1 from two different cells in Figure 1.8(b) were combined into one face in Figure 1.8(c). While the extra vertex was also removed. In section 1.5.1, we discuss techniques to optimize the reduce process of combining faces.

For symmetric difference, $A \triangle B$, the map phase filters faces whose label is a single layer (A or B). For the difference, $A \setminus B$, it filters faces with label A. For union $A \cup B$, all faces in the overlay structure are retrieved.

1.5 Overlay evaluation optimizations

We now focus on the different optimization aspects regarding the best approach to compute the boundaries of faces that span over different cells and how to mitigate the issues of layers with an unbalanced number of edges.

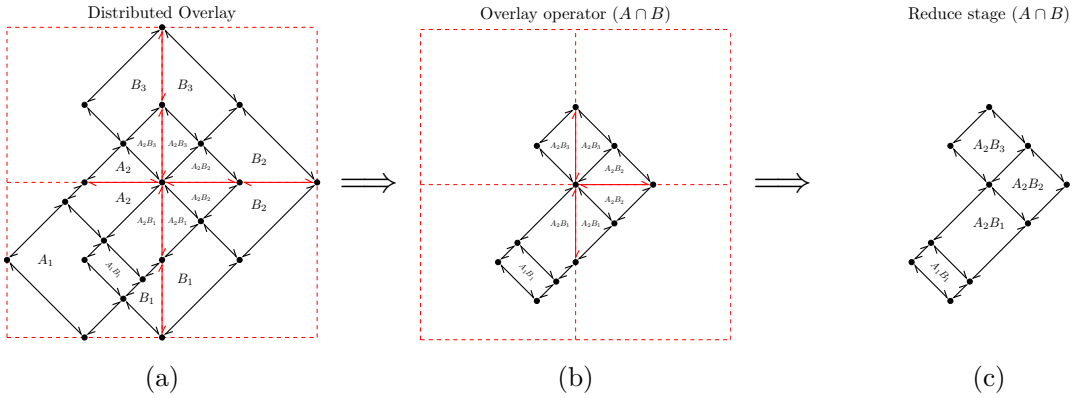


Figure 1.8: Example of an overlay operator querying the distributed DCEL.

1.5.1 Optimizations for faces spanning multiple cells

The naive reduce phase described above has the potential for a bottleneck since all faces, which can be a very large number, are sent to one worker node. From a distributed perspective, this process follows a typical MapReduce pattern. In the map phase, each worker node identifies and reports faces that are fully contained within its boundaries, as well as segments of faces that may need to be concatenated with segments reported by other nodes. These face segments are then sent to a master node, incurring communication costs as the master must wait for all nodes to report their segments. In the reduce phase, the master node groups the segments by face ID, sorts them, and concatenates the parts to form complete, closed faces. One observation is that faces from different concatenated cells are in contiguous cells. This implies that faces from a particular cell will be combined with faces from neighboring cells. We will use this spatial proximity property to reduce the overhead in the central node.

We thus propose an alternative where an intermediate reduce processing step is

introduced. In particular, the user can specify a level in the quadtree structure, measured as the depth from the root, that can be used to combine cells together. While it may be challenging to predetermine an optimal level, it can be estimated based on the input size or the number of partitions. Moreover, Section 1.6.2 offers recommendations for suitable values and alternative approaches. Given level i, the quadtree nodes in that level (at most 4^i) will serve as intermediate reducers, collecting the faces from all the cells below that node. Note: level 0 corresponds to the root, which is the naive method where all the cells are sent to one node.

By introducing this intermediate step, it is expected that much of the reduce work can be distributed in a larger number of worker nodes. Nevertheless, there may be faces that cannot be completed by these intermediate reducers because they span the borders of the level i nodes. Such faces still have to be evaluated in a master/root node. From a Map-Reduce standpoint, this alternative functions similarly to the previous approach but introduces additional reduce operations at an intermediate level. However, this also introduces new synchronization points, as each intermediate reducer must wait for its workers to report potential face segments before processing them. The reducer then either reports completed faces or sends incomplete segments to the driver for further processing.

Clearly, picking the appropriate level is important. Choosing a level i, i.e., going to nodes lower in the quadtree structure, implies a larger number of intermediate reducers and, thus, higher parallelism. However, simultaneously, it increases the number of faces that would need to be evaluated by the master/root node. On the other hand, lowering i reduces parallelism, but fewer faces will need to go to the master/root node.

We also examine another approach to deal with the bottleneck in the naive reduce phase. This approach re-partitions the faces using the label as the key. Such partitions represent small independent amounts of work since they only combine faces with the same label that are typically few. Partitions are then shuffled among the available nodes. The second approach effectively avoids the reduce phase; it has to account for the cost of the re-partitioning; however, as we will show in the experimental section, this cost is negligible. From a distributed computing perspective, this alternative introduces a shuffle stage at the beginning, eliminating the need for a reduce operation. The shuffle ensures that all segments with the same face ID are placed in the same worker, allowing them to be processed and reported directly.

1.5.2 Optimizing for unbalanced layers

During the overlay computation, finding the intersections between the half-edges is the most critical task. In many cases, the number of half-edges from each layer within a cell can be unbalanced; that is, one of the layers has many more half-edges than the other.

In our initial implementation, the input sets of half-edges within each cell were combined into a single dataset, initially ordered by the x-origin of each half-edge. Then, a sweep-line algorithm is performed, scanning the half-edges from left to right (in the x-axis). This scanning takes time proportional to the total number of half-edges. However, if one layer has much fewer half-edges, the running time will still be affected by the cardinality of the larger dataset.

An alternative approach is to scan the larger dataset only for the x-intervals where we know that there are half-edges in the smaller dataset. To do so, we order the two input

sets separately. We scan the smaller dataset in x-order and identify x-intervals occupied by at least one half-edge. For each x-interval, we then scan the larger dataset using the sweep-line algorithm. This focused approach avoids unnecessary scanning of the large dataset, for example, areas with no half-edges from the smaller dataset.

1.6 Experimental Evaluation

For our experimental evaluation, we used a 12-node Linux cluster (kernel 3.10) and Apache Spark 2.4. Each node has 9 cores (each core is an Intel Xeon CPU at 1.70GHz) and 2G memory.

The scalable approach was implemented over the Apache Spark framework. From a Map-Reduce point of view the stages described in Section 1.4 were implemented using several transformations and actions supported by Apache Spark. For example, the partitioning and load balancing described in Section 1.4.1 was implemented using a quadtree, where its leaves were used to map and balance the number of edges that have to be sent to the worker nodes. Mostly, map operations were used to process and locate the edges in the corresponding leaf to exploit proximity among them while at the same time dividing the amount of work among worker nodes.

Similarly, the edges at each partition were processed using chains of transformations at local level (see Section 1.4) followed by reducer actions to post-process incomplete faces which could span over multiples partitions and have to be combined or re-distributed to obtain the final answer. In addition, the reduce actions were further optimized as described in Section 1.5.

Table 1.4: Evaluation Datasets

Dataset	Layer	Number of polygons	Number of edges
MainUS	Polygons for 2000	64983	35417146
	Polygons for 2010	72521	36764043
GADM	Polygons for Level 2	160241	64598411
	Polygons for Level 3	223490	68779746
CCT	Polygons for 2000	7028	2711639
	Polygons for 2010	8047	2917450

1.6.1 Evaluation datasets

The details of the real datasets of polygons that we use are summarized in Table 1.4. The first dataset (MainUS) contains the complete Census Tracts for all the states on the US mainland for the years 2000 (layer A) and 2010 (layer B). It was collected from the official website of the United States Census Bureau [28]. The data was clipped to select just the states inside the continent. Something to note with this dataset is that the two layers present a spatial gap (which was due to improvements in the precision introduced for 2010). As a result, there are considerably more intersections between the two layers, thus creating many new faces for the DCEL.

The second dataset, GADM - taken from Global Administration Areas [14], collects the geographical boundaries of the countries and their administrative divisions around the

globe. For our experiments, one layer selects the States (administrative level 2), and the other has Counties (administrative level 3). Since GADM may contain multi-polygons, we split them into their individual polygons.

Since these two datasets are too large, a third, smaller dataset was created for comparisons with the sequential algorithm. This dataset is the California Census Tracts (CCT), a subset from MainUS for the state of California; layer A corresponds to the CA census tracts from the year 2000, while layer B corresponds to 2010. Below, we also use other states to create datasets with different numbers of faces. To test the scalable approach, a sequential algorithm for DCEL creation was implemented based on the pseudo-code outlined in [4].

1.6.2 Overlay face optimizations

We first examine the optimizations in Section 1.5.1. To consider different distributions of faces, for these experiments, we used 8 states from the MainUS dataset with different numbers of tracts (faces). In particular, we used, in decreasing order of number of tracts, CA, TX, NC, TN, GA, VA, PA, and FL. For each state, we computed the distributed overlay between two layers (2000 and 2010). For each computation, we compared the baseline; master at the root node, with intermediate reducers at different levels: i varied from 4 to 10.

Figure 1.9 shows the results for the distributed overlay computation stage; after the local DCELs were computed at each cell. Note that for each state experiment, we tested different numbers of cells for the quadtree and reported the configuration with the best performance. To determine this, we sampled 1% of the edges for each state and evaluated

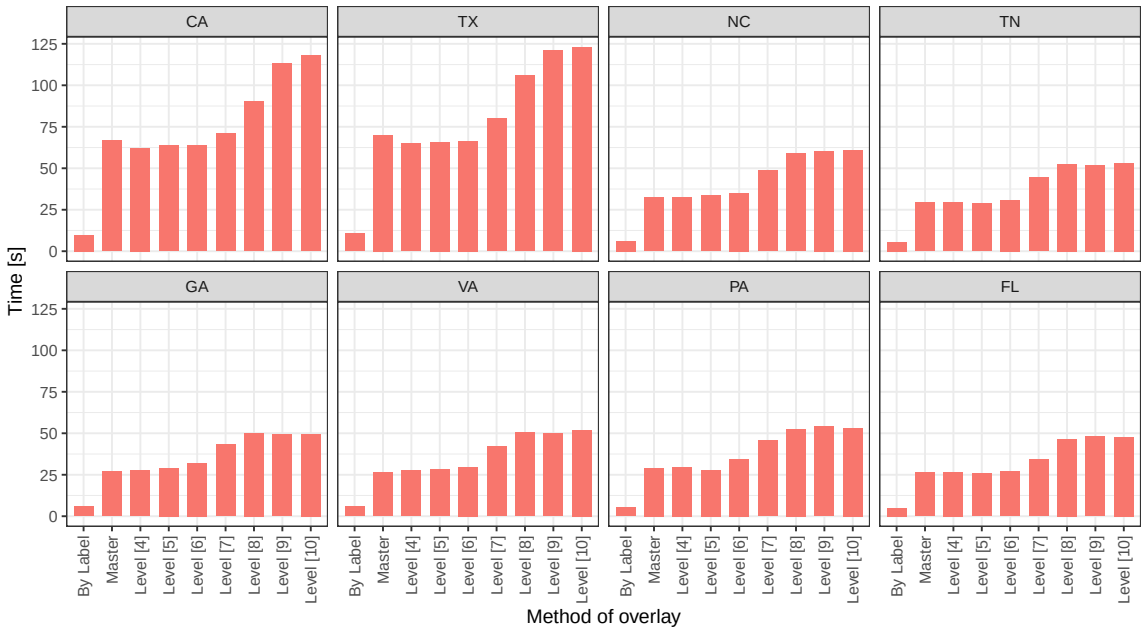


Figure 1.9: Overlay methods evaluation.

the best number of cells ranging from 200 to 2000. In most cases, the best number of cells was around 3000. As expected, there is a trade-off between parallelism and how much work is left to the final reduce job. For different states, the optimal i varied between levels 4 and 6. The figure also shows the optimization that re-partitions the faces by label id. This approach has actually the best performance. This is because few faces with the same label can be combined independently. This results in smaller jobs better distributed among the cluster nodes, and no reduce phase is needed. As a result, we use the label re-partition approach for the rest of the experiments to implement the overlay computation stage.

Finally we note that the overlay face optimizations involve shuffling of the incomplete faces. Table 1.5 shows the percentage of incomplete faces for three states, assuming 3000 cells. As it can be seen, the incomplete faces is small (in average 12.89%) and moreover,

Table 1.5: Percentages of edges in incomplete faces for three states

Dataset	Number of edges	Edges in incomplete faces	Percentage
CA	47834	6339	13.25%
TX	41227	4436	10.75%
FL	24152	3547	14.68%

for the *By-Label* approach, this shuffling is parallelized.

1.6.3 Unbalanced layers optimization

For these experiments, we compared the traditional sweep approach with the 'filtered-sweep' approach that considers only the areas where the smaller layer has edges (Section 1.5.2). To create the smaller cell layer, we picked a reference point in the state of Pennsylvania, from the MainUS dataset, and added 2000 census tracts until the number of edges reached 3K. We then varied the size of the larger cell layer in a controlled way: using the same reference point but using data from the 2010 census, and we started adding tracts to create a layer that had around 2x, 3x, ..., 7x the number of edges of the smaller dataset.

Since this optimization occurs per cell, we used a single node to perform the overlay computation within that cell. Figure 1.10(a) shows the behavior of the two methods (filtered-sweep vs. traditional sweep) under the above-described data for the overlay computation stage. Clearly, as the data from one layer grows much larger than the other layer,

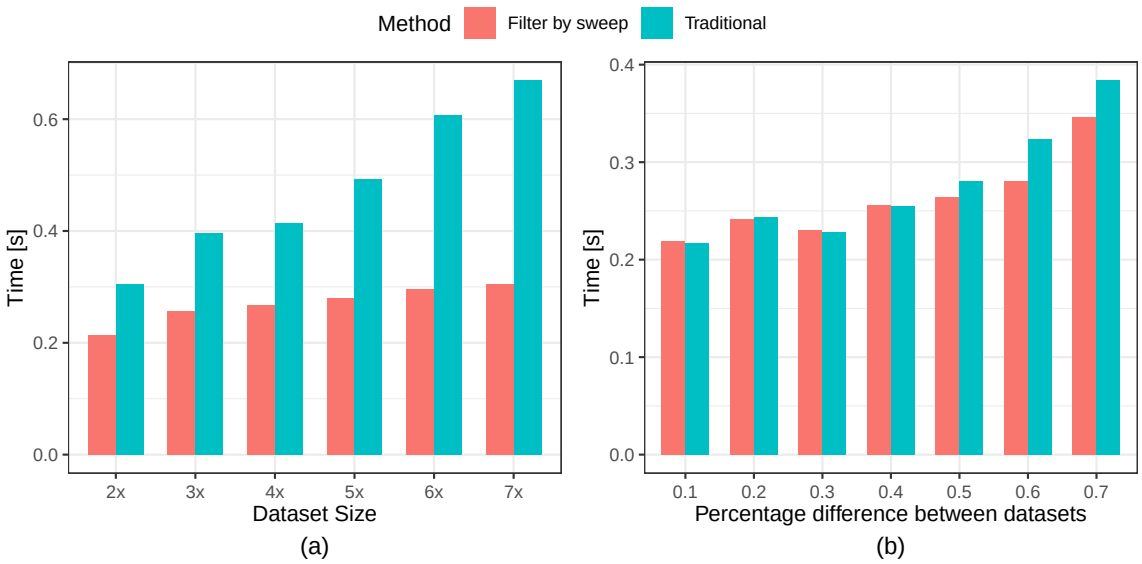


Figure 1.10: Evaluation of the unbalanced layers optimization.

the filtered-sweep approach overcomes the traditional one.

We also performed an experiment where the difference in size between the two layers varies between 10% and 70%. For this experiment, we first identified cells from the GADM dataset where the smaller layer had around 3K edges. Among these cells, we then identified those where the larger layer had 10%, 20%, ... up to 70% more edges. In each category, we picked 10 representative cells and computed the overlay for the cells in that category.

Figure 1.10(b) shows the results; in each category, we show the average time to compute the overlay among the 10 cells in that category. The filtered-sweep approach shows better performance as the percentage difference between layers increases. Based on these results, one could apply the optimization on those cells where the layer difference is significant (more than 50%). We anticipate that this optimization will be particularly

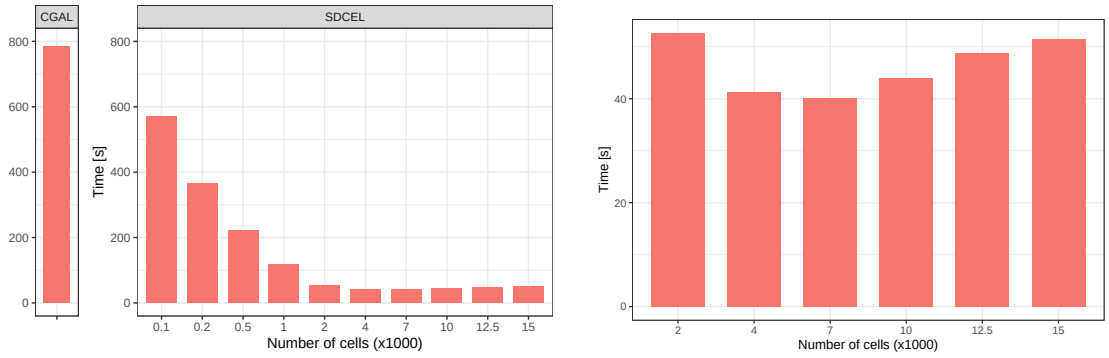
beneficial for datasets where the two input layers contain many cells with significantly different edge counts.

1.6.4 Varying the number of cells

The quadtree configuration allows for performance tuning by setting the *maximum capacity* of a cell. The quadtree continues splitting until this capacity is reached. There is an inverse relationship between the capacity and the number of leaf cells: a lower capacity results in more cells, while a higher capacity leads to fewer leaf cells. In skewed datasets, the quadtree may become unbalanced, with some branches splitting more frequently. As a result, the final number of partitions is not necessarily a multiple of four. In the figures, we round the number of leaf cells to the nearest thousand.

The number of cells affects the performance of our scalable overlay implementation, termed as SDCEL, since it relates to the average cell capacity given by the number of edges it could contain. As it was said before, a fewer number of cells implies larger cell capacity and thus more edges to process within each cell. Complementary, creating more cells increases the number of jobs to be executed.

Figure 1.11(a) shows the SDCEL performance using the two layers of the CCT dataset while varying the number of cells from 100 to 15K (by multiple of 1000). Each bar corresponds to the time taken to create the DCEL for each layer and then combine them to create the distributed overlay. Clearly, there is a trade-off: as the number of cells increases, the SDCEL performance improves until a point where the larger number of cells adds an overhead. Figure 1.11(b) focuses on that area; the best SDCEL performance was around 7K cells.



(a) (b)

Figure 1.11: SDCEL performance while varying the number of cells in the CCT dataset.

In addition, Figure 1.11(a) shows the performance of the sequential solution (CGAL library) for computing the overlay of the two layers in the CCT dataset using one of the cluster nodes. Clearly, the scalable approach is much more efficient as it takes advantage of parallelism. Note that the CGAL library would crash when processing the larger datasets (MainUS and GADM).

Figure 1.12 shows the results when using the larger MainUS and GADM datasets, while again varying the number of cells parameter from 8K to 18K and from 16K to 34K, respectively. In this figure, we also show the time taken by each stage of the overlay computation. This is, the time to create the DCEL for layer A, for layer B, and for their combination to create their distributed overlay. We can see a similar trade-off in each of the stages. The best performance is given when setting the number of cells parameter to 12K for the MainUS and 22K for the GADM dataset. Note that in the MainUS dataset, the two layers have a similar number of edges; as can be seen, their DCEL computations

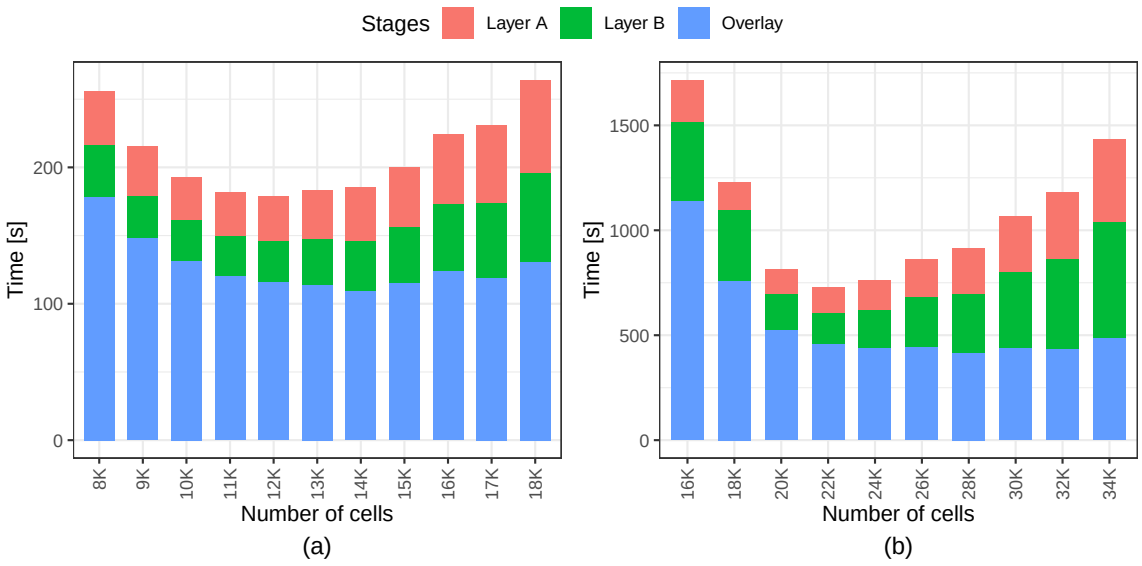


Figure 1.12: Performance with (a) MainUS and (b) GADM datasets.

are similar.

Interestingly, the overlay computation is expensive since as mentioned earlier there are many intersections between the two layers. An interesting observation from the GADM plots is that layer B takes more time than layer A; this is because there are more edges in the counties than in the states. Moreover, county polygons are included in the (larger) state polygons. When the size of cells is small (i.e., a larger number of cells like in the case of 34K cells), these cells mainly contain counties from layer B. As a result, there are not many intersections between the layers in each cell, and the overlay computation is thus faster. On the other hand, with large cell sizes (smaller number of cells), the area covered by the cell is larger, containing more edges from states and thus increasing the number of intersections, resulting in higher overlay computation.

Additionally, Table 1.6 provides statistics on the cells. It shows that in larger

Table 1.6: Cell size statistics.

Dataset	Min	1st Qu.	Median	Mean	3rd Qu.	Max
GADM	0	0	2768	3141	5052	16978
MainUS	0	1538	2582	2853	3970	10944
CCT	0	122	324	390	546	1230

Table 1.7: Orphan cells and orphan holes description

Dataset	Number of cells	Number of holes	Number of orphans (cell/holes)
GADM	21970	1999	4310
MainUS	12343	850	1069
CCT	7124	40	215

datasets, an average cell size of approximately 3000 edges produces the best results. This cell size ensures a relatively small amount of data to transmit, which minimizes the impact on data shuffling and processing. Table 1.7 presents the number of cells, original holes, and the orphan cells and holes generated after partitioning.

1.6.5 Speed-up and Scale-up experiments

The speed-up behavior of SDCEL appears in Figure 1.13(a) (for the MainUS dataset) and in Figure 1.14(a) (for the GADM dataset); in both cases, we show the per-

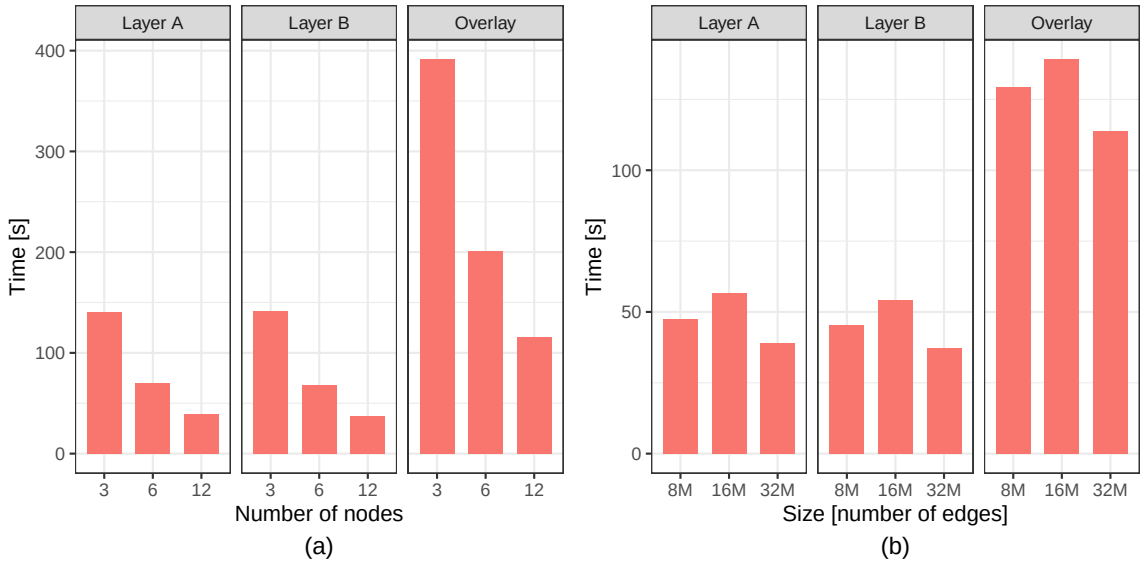


Figure 1.13: Speed-up and Scale-up experiments for the MainUS dataset.

formance for each stage. For these experiments, we varied the number of nodes to 3, 6, and 12 while keeping the input layers the same. Clearly, as the number of nodes increases, the performance improves. SDCEL shows good speed-up characteristics: as the number of nodes doubles from 3 to 6 and then from 6 to 12, the performance improves by almost half.

To examine the scale-up behavior, we created smaller datasets out of the MainUS and similarly out of the GADM so that we could control the number of edges. To create such a dataset, we picked a centroid and started increasing the area covered by this dataset until the number of edges was closed to a specific number. For example, from the MainUS, we created datasets of sizes 8M, 16M, and 32M edges for each layer. We then used two layers of the same size as input to a different number of nodes while keeping the input-to-node ratio fixed. That is, the layers of size 8M were processed using 3 nodes, the layers of size 16M using 6 nodes, and the 32M using 12 nodes. We used the same process for

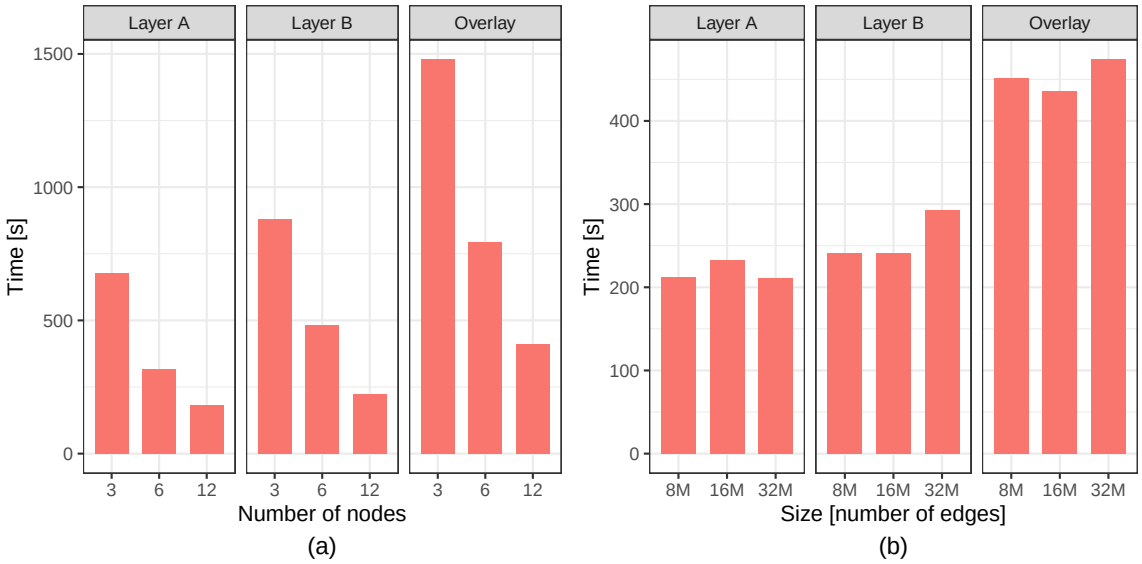


Figure 1.14: Speed-up and Scale-up experiments for the GADM dataset.

the scale-up experiments with the GADM dataset. The results appear in Figure 1.13(b) and Figure 1.14(b). Overall, SDCEL shows good scale-up performance; it remains almost constant as the work per node is similar (there are slight variations because we could not control perfectly the number of edges and their intersection).

Bibliography

[1] G. Barequet. DCEL - A Polyhedral Database and Programming Environment. *Ijcga*, 08(05n06):619–636, 1998.

[2] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *Acm Sigmod Pods*, pages 322–331, New York, NY, USA, 1990. Association for Computing Machinery.

[3] E. Berberich, E. Fogel, D. Halperin, M. Kerber, and O. Setter. Arrangements on Parametric Surfaces. *Mathematics in Computer Science*, 4(1):67–91, 2010.

[4] M. Berg, O. Cheong, M. Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer, TU Eindhoven, P.O. Box 513, 2008.

[5] P. Boguslawski, C. Gold, and H. Ledoux. Modelling and analysing 3D buildings with a primal/dual data structure. *Isprs*, 66(2):188–197, 2011.

[6] D. Boltcheva, J. Basselin, C. Poull, H. Barthélemy, and D. Sokolov. Topological-based roof modeling from 3D point clouds. In *Wscg*, volume 28, pages 137–146, CZ 301 00 Plzen, 2020. Union Agency, Science Press.

[7] J. Challa, P. Goyal, S. Nikhil, A. Mangla, S. Balasubramaniam, and N. Goyal. DD-Rtree: A dynamic distributed data structure for efficient data distribution among cluster nodes for spatial data mining algorithms. In *IEEE Big Data*, pages 27–36, 222 Rosewood Drive, Danvers, MA 01923., 2016. Ieee.

[8] L. Chew and K. Kedem. A convex polygon among polygonal obstacles. *Computational Geometry*, 3(2):59–89, 1993.

[9] V. Chvátal. A combinatorial theorem in plane geometry. *Combinatorial Theory*, 18(1):39–41, 1975.

[10] Raphael Finkel and Jon Bentley. Quadtrees: A Data Structure for Retrieval on Composite Keys. *Acta Inf.*, 4:1–9, March 1974.

[11] E. Fogel, D. Halperin, and R. Wein. *CGAL Arrangements and Their Applications*. Springer Berlin, Heidelberg, 2012.

[12] W. Franklin, S. Magalhães, and M. Andrade. Data Structures for Parallel Spatial Algorithms on Large Datasets. In *ACM BigSpatial*, pages 16–19, Seattle, WA, USA, 2018. Acm.

[13] W. Freiseisen. Colored DCEL for boolean operations in 2D, 1998.

[14] GADM maps and data. https://gadm.org/.

[15] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Acm Sigmod Icmd*, pages 47–57, New York, NY, United States, 1984. Association for Computing Machinery.

[16] R. Holmes. The DCEL Data Structure for 3D Graphics, 2021.

[17] Y. Li, A. Eldawy, J. Xue, N. Knorozova, M. Mokbel, and R. Janardan. Scalable computational geometry in Map-Reduce. *VLDB*, 28(1):523–548, 2019.

[18] S. Magalhães, M. Andrade, W. Franklin, and W. Li. Fast exact parallel map overlay using a two-level uniform grid. In *ACM BigSpatial*, pages 45–54, New York, NY, USA, 2015. Association for Computing Machinery.

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

[20] D. Muller and F. Preparata. Finding the intersection of two convex polyhedra. *Theoretical Computer Science*, 7(2):217–236, 1978.

[21] J. Nievergelt, H. Hinterberger, and K. Sevcik. The Grid File: An Adaptable, Symmetric Multikey File Structure. *ACM Trans. Database Syst.*, 9(1):38–71, 1984.

[22] J. O'Rourke. *Art Gallery Theorems and Algorithms*. Oxford University Press, United States, 1987.

[23] F. Preparata and M. Shamos. *Computational Geometry: An Introduction*. Springer, New York, NY, 1985.

[24] S. Puri, D. Agarwal, X. He, and S. Prasad. MapReduce Algorithms for GIS Polygonal Overlay Processing. In *Ieee Ipdps*, pages 1009–1016, Cambridge, MA, USA, 2013. Ieee.

[25] S. Puri and S. Prasad. Efficient Parallel and Distributed Algorithms for GIS Polygonal Overlay Processing. In *Ieee Ipdps*, pages 2238–2241, Usa, 2013. IEEE Computer Society.

[26] I. Sabek and M. Mokbel. On Spatial Joins in MapReduce. In *Acm Sigspatial*, pages 1–10, New York, NY, USA, 2017. Association for Computing Machinery.

[27] H. Samet. *The Design and Analysis of Spatial Data Structures*. Wesley, 75 Arlington Street, Suite 300 Boston, MA, United States, 1990.

[28] United States Census Data. https://www2.census.gov/geo/tiger/TIGER2010/TRACT/.